

NPS-SE-18-002



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

**METHODOLOGIES AND PATTERNS FOR MODEL
CENTRIC SYSTEMS ENGINEERING**

by

Kristin Giammarco
Bruce Allen

March 2018

Approved for public release. Distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, 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 Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY)		2. REPORT TYPE		3. DATES COVERED (From — To)	
03-02-2018		Technical Report		10-01-2016 to 01-31-2018	
4. TITLE AND SUBTITLE METHODOLOGIES AND PATTERNS FOR MODEL CENTRIC SYSTEMS ENGINEERING				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Kristin Giammarco, Bruce Allen				5d. PROJECT NUMBER	
				5e. TASK NUMBER W7A36	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943				8. PERFORMING ORGANIZATION REPORT NUMBER NPS-SE-18-002	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Research Program				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited					
13. SUPPLEMENTARY NOTES The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.					
14. ABSTRACT This research aims to improve the development and analysis of architecture models using the Unified Modeling Language (UML) Profile-based Integrated Architecture (UPIA)-inspired Generic Model guidance published by the Architecture Modeling for Systems Engineering Working Group (AMSEWG). To test its tool-neutrality, the Generic Model language is modeled in several other tools. This language was successfully reproduced in Alloy, a formal language, and in Innoslate, a model-based systems engineering (MBSE) tool. The results of this research include new patterns, templates and style guidance for architecture-level model views, which can be used to inform the continued evolution of modeling guidance. We provide graphical templates for structuring executable behavior models, and discuss the benefits and limitations of these templates. The results also found that Monterey Phoenix (MP) tools MP-Firebird and MP-Python are capable of representing aspects of the Generic Model language that pertain to behavior. The graphical templates and MP modeling in this research lay the groundwork for follow-on work to automate the translation of graphical behavior models into MP to leverage its scope-complete scenario generation, emergent behavior identification, verification, and validation capabilities.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (include area code)
Unclassified	Unclassified	Unclassified	UU	159	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

THIS PAGE INTENTIONALLY LEFT BLANK

**NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000**

Ronald A. Route
President

Steven R. Lerman
Provost

The report entitled “Methodologies and Patterns for Model Centric Systems Engineering” was prepared for the Naval Air Systems Command and funded by the Naval Research Program.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Kristin Giammarco

Bruce Allen

Reviewed by:

Ronald E. Giachetti, Chairman
Department of Systems Engineering

Peter Denning, Chairman
Department of Computer Science

Released by:

Jeffrey D. Paduan
Dean of Research

ABSTRACT

This research aims to improve the development and analysis of architecture models using the Unified Modeling Language (UML) Profile-based Integrated Architecture (UPIA)-inspired Generic Model guidance published by the Architecture Modeling for Systems Engineering Working Group (AMSEWG). To test its tool-neutrality, the Generic Model language is modeled in several other tools. This language was successfully reproduced in Alloy, a formal language, and in Innoslate, a model-based systems engineering (MBSE) tool. The results of this research include new patterns, templates and style guidance for architecture-level model views, which can be used to inform the continued evolution of modeling guidance. We provide graphical templates for structuring executable behavior models, and discuss the benefits and limitations of these templates. The results also found that Monterey Phoenix (MP) tools MP-Firebird and MP-Python are capable of representing aspects of the Generic Model language that pertain to behavior. The graphical templates and MP modeling in this research lay the groundwork for follow-on work to automate the translation of graphical behavior models into MP to leverage its scope-complete scenario generation, emergent behavior identification, verification, and validation capabilities.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Objectives	2
1.3	Research Methodology	2
1.4	Student Involvement	3
1.5	Chapter Summary	4
2	The Conceptual Data Model (CDM) as a Language for Architecture Patterns	5
2.1	Architecture Modeling Software Analytics	5
2.2	Translation of Analytics into Other CDM Languages	18
2.3	Examples of Well-formed Architecture Models	25
2.4	Chapter Summary	53
3	A Tool-Agnostic Methodology for Deriving and Documenting Architecture-Level Patterns	54
3.1	Introduction	54
3.2	Related Work	55
3.3	Method	56
3.4	Implementation	84
3.5	Discussion and Summary	85
4	Graphical Templates for Executable Behavior Models	88
4.1	Actions	88
4.2	Branches	91
4.3	Interaction Types	92
4.4	Templates	93
5	Virtual Testing of System Behavior using Executable Models in Monterey Phoenix	104

5.1	Method for Using MP	104
5.2	Introduction to MP-Python	105
5.3	HADR scenario UAV behaviors modeled using MP-Python	109
5.4	Relationship of MP Events to Generic Model Classes	130
6	Conclusions and Proposed Way Ahead	132
6.1	Reproducibility of the Generic Model in other tools.	132
6.2	Tool-agnostic patterns and style guidance.	133
6.3	Executability of behavior views in Innoslate and Monterey Phoenix	133
6.4	Tool-agnostic instructions for creating and analyzing executable activity models	133
6.5	Recommendations for Further Research	134
	List of References	135
	Initial Distribution List	140

List of Figures

Figure 2.1	Four simple CDM examples.	7
Figure 2.2	Main Lifecycle Modeling Language (LML) classes and relationships (reciprocal relationships hidden for readability), after Figure 3-1 in [1].	8
Figure 2.3	Example mapping between the Unified Modeling Language (UML) Profile-based Integrated Architecture (UPIA) and LML CDMs. .	9
Figure 2.4	Pattern for a high level operational concept graphic. Two assets (boxes) are connected to each other by a conduit (depicted here in the style of a yellow lightning bolt). The background is an image of Mars that has been color-filtered for an Earth-like atmosphere. Image source: NASA.	26
Figure 2.5	Example of a high level operational concept graphic. Assets (boxes) are connected to each other by conduits (yellow lightning bolts or bidirectional arrows, in this case). The environment is explicitly modeled as an entity that can have interactions with systems. . .	27
Figure 2.6	Pattern for a physical hierarchy. This view shows the composition of the system and the environment assets depicted in the high level operational graphic. No asset in this chart has more than one parent or exactly one child.	29
Figure 2.7	Example of a physical hierarchy. This view shows the composition of the Military Assets, Other Assets, Unmanned Aerial System (UAS), and even the UAV depicted in the high level operational graphic. No asset in this chart has more than one parent or exactly one child. .	29
Figure 2.8	Pattern for a functional hierarchy. This view shows the corresponding functions (actions) for the System and the Environment assets and their subsystems. No action in this chart has more than one parent or exactly one child.	30
Figure 2.9	Example of a functional hierarchy. This view shows the hierarchy of corresponding functions (actions) for the HADR assets. No action in this chart has more than one parent or exactly one child.	30

Figure 2.10	Pattern for a spider traceability graph. This view shows the function-to-form mapping for a top level system. Each action is performed by an asset (blue arrows).	31
Figure 2.11	Example of a spider traceability graph. This view shows the function-to-form mapping for the UAV employed in the HADR mission. Each action is performed by an asset (blue arrows).	32
Figure 2.12	Pattern for a use case diagram. The use case diagram shows which actors participate in which use cases. Use cases (ovals) are actions that are performed by actors (assets), shown here as stick figures.	33
Figure 2.13	Example of a use case diagram. Each action (use case) is performed by at least one asset (actor). The maintenance use case is shown as including two different types of maintenance: scheduled and unscheduled, each of which involves different procedures.	34
Figure 2.14	Pattern for an action diagram. Each asset performs at least one action; each input/output is both generated and received by an action. . .	35
Figure 2.15	Example of an action diagram. Each asset performs at least one action; each input/output is both generated and received by an action.	36
Figure 2.16	Pattern for an event trace -style sequence diagram. The arrows are labeled with verb-oriented events, or actions. (Perform B and Perform C are not shown because they are internal actions of the system, from Figure 2.14.)	37
Figure 2.17	Pattern for an input/output-style sequence diagram. Here, the arrows are labeled with noun-oriented input/outputs.	37
Figure 2.18	Example of an event trace -style sequence diagram. The arrows are labeled with verb-oriented events, or actions, from the corresponding action diagram.	37
Figure 2.19	Example of an input/output-style sequence diagram. The arrows are labeled with noun-oriented input/outputs from the corresponding action diagram.	38
Figure 2.20	Patterns for an "Integrated Computer Aided Manufacturing" or Icam DEfinition for Functional Modeling (IDEF0)-A0 diagram. Each action (function) has at least one control and one output (an IDEF0 heuristic from [2]).	40

Figure 2.21	Example of an IDEF0-A0 diagram, top level. The action has at least one control and one output (an IDEF0 heuristic from [2]).	41
Figure 2.22	Example of an IDEF0-A0 diagram, first level. The action has at least one control and one output (an IDEF0 standard heuristic from [2]).	41
Figure 2.23	Pattern for a context-level IDEF0 diagram. The main system function has at least one control and one output.	43
Figure 2.24	Example of a context-level IDEF0 diagram. Interactions between the UAS top level function (OA.0) and the external system functions are shown, and some interactions among external system functions are shown. The figure also demonstrates that six functions is about the maximum that humans would typically want to look at at once (with fewer being preferred), and pushes the limit of legibility on standard page dimensions used for reports.	44
Figure 2.25	Pattern for a first-level IDEF0 diagram. Each function has at least one control and one output.	45
Figure 2.26	Pattern for an IDEF0 decomposing external functions. Each function has at least one control and one output, except for the Physical Environment, whose controls are outside the scope of the model.	46
Figure 2.27	Example of an IDEF0 diagram, first level. The input/outputs from the top level (black arrows) are carried down and allocated to first level functions of the UAS, and new internal input/outputs (blue arrows) are created to provide interaction among the first level functions that provide continuity for the top level input/outputs.	47
Figure 2.28	Example of an IDEF0 diagram, second level. We can distinguish input/outputs originating from the top level (black arrows), the first level (blue arrows), and new internal input/outputs (orange arrows) created to provide interaction among the second level functions that support first level and top level interactions.)	48
Figure 2.29	Pattern for a context-level asset diagram, showing a call-out of input/outputs transferred by one of the conduits.	49
Figure 2.30	Pattern for a first-level asset diagram of the system, showing a call-out of input/outputs transferred by one of the conduits.	49
Figure 2.31	Pattern for an asset diagram of assets making up the environment, showing a call-out of input/outputs transferred by one of the conduits.	50

Figure 2.32	Example of a context-level asset diagram, showing a call-out of input/outputs transferred by one of the conduits.	50
Figure 2.33	Example of a first-level asset diagram of the UAS system, showing a call-out of input/outputs transferred by one of the conduits. . . .	51
Figure 2.34	Example of a second-level asset diagram of the UAV system, showing a call-out of input/outputs transferred by one of the conduits. . .	52
Figure 3.1	(a) Architecture metamodel graph; (b) Architecture metamodel Alloy code.	57
Figure 3.2	OperationalTask0 has three looping information elements.	59
Figure 3.3	InformationElement2 does not have a source operational task specified.	60
Figure 3.4	InformationElement0 does not have a destination operational task specified.	61
Figure 3.5	CapabilityRole0 does not have an operational task specified. . . .	62
Figure 3.6	Operational tasks that do not interact through exchange of information elements.	63
Figure 3.7	An operational task that is not performed by any capability role. .	64
Figure 3.8	Modified Alloy code containing constraints.	65
Figure 3.9	Two capability roles are not interoperable with one another. . . .	66
Figure 3.10	Architecture metamodel graph, extended to include needlines. . .	69
Figure 3.11	Extensions for Needline class in architecture metamodel Alloy code.	70
Figure 3.12	Architecture metamodel graph, extended to include standards. . .	72
Figure 3.13	Implementation of standards constraints in Alloy code, part 1 (signatures).	75
Figure 3.14	Implementation of standards constraints in Alloy code, part 2 (global constraints).	76
Figure 3.15	Implementation of standards constraints in Alloy code, part 3 (reciprocal relations and run statement).	77

Figure 3.16	Architecture metamodel graph, extended to include more Generic Model classes and relationships.	78
Figure 3.17	Alloy model for the Generic Model, part 1 (signatures and local constraints for Vision, StrategicMission, Capability, CapabilityRequirement, OrganizationalResource, and CapabilityUsage). . . .	79
Figure 3.18	Alloy model for the Generic Model, part 2 (signatures and local constraints for CapabilityRole, OperationalTask, InformationElement, Needline, and Standard).	80
Figure 3.19	Alloy model for the Generic Model, part 3 (global constraints). .	81
Figure 3.20	Alloy model for the Generic Model, part 4 (reciprocal relations and LML mappings).	82
Figure 3.21	Generic model UPIA metamodel graph, with mappings to LML classes and relationships.	83
Figure 4.1	An action is denoted by a rectangular box. Actions can follow and precede other actions. Icon source: Innoslate.	88
Figure 4.2	An "or" action is denoted by a box leading with a diamond shape on the left, and a trailing branch for each of its possible outcomes. Icon source: Innoslate.	89
Figure 4.3	An "or" action, named in question format. Outcomes are of a yes or no nature.	89
Figure 4.4	An "or" action, named in activity format. Outcomes are the possible decisions.	89
Figure 4.5	A "loop" action is denoted by a box leading with a diamond shape and a trailing branch that loops back into the top of the box. A second branch exits the loop from the bottom of the box. Icon source: Innoslate.	90
Figure 4.6	A "sync" action is denoted by a box trailing with a diamond shape, and a leading branch for each of its possible predecessors. Icon source: Innoslate.	91
Figure 4.7	Time proceeds from the beginning to the end of the branch (here, left to right).	91

Figure 4.8	Parallel branches denote logical concurrency. Icon source: Innoslate.	92
Figure 4.9	If all the actions on a given branch are performed by the same asset, then the branch can be labeled with the performing asset. Icon source: Innoslate.	92
Figure 4.10	Actions may depend on the presence of sufficient resources. Icon source: Innoslate.	93
Figure 4.11	Actions may depend on input/outputs such as information or instructions. Triggering input/outputs (required dependencies) are green; optional input/outputs are grey. Icon source: Innoslate.	93
Figure 4.12	A one-branch sequence.	94
Figure 4.13	A multi-branch sequence.	95
Figure 4.14	A single decision point.	96
Figure 4.15	A separation of actors, where the behaviors of each are partitioned onto different diagrams.	97
Figure 4.16	Separation of phases applied to the HADR mission example. . . .	98
Figure 4.17	Separation of phases applied to the template examples.	98
Figure 4.18	Continuation of loops until a failure mode occurs.	99
Figure 4.19	Continuation of loops until a resource is in low supply.	101
Figure 4.20	Continuation of loops until a counter reaches zero.	103
Figure 5.1	MP-Python Main Window.	106
Figure 5.2	MP-Python File Menu, expanded to show pre-loaded examples. .	107
Figure 5.3	UAV Ingress, Scope 1 Trace 1, Proceed.	112
Figure 5.4	UAV Ingress, Scope 1 Trace 2, Abort.	113
Figure 5.5	UAV On Station, Scope 1 Trace 7.	117
Figure 5.6	UAV On Station, Scope 2 Trace 33.	118
Figure 5.7	UAV On Station, Scope 4 Trace 363, illustrating four targets. . . .	119

Figure 5.8	UAV Egress, Scope 1 Trace 81.	125
Figure 5.9	UAV Postflight, Scope 1 Trace 1.	129
Figure 5.10	UAV Postflight, Scope 1 Trace 1, MP-Firebird version 2 view, for comparison.	130
Figure 5.11	Root events in MP typically correspond to components of a system, systems, or the environment, and so can be typed as UPIA capability roles. High level composite and atomic events may be typed as UPIA operational tasks.	131
Figure 5.12	Root events can alternatively be typed as LML assets. Composite and atomic events may be typed as LML actions.	131

List of Tables

Table 2.1	Hierarchy Analytics.	18
Table 2.2	Functional/Physical Allocation Analytics.	20
Table 2.3	Functional Interaction Analytics.	20
Table 2.4	Physical Interaction Analytics.	21
Table 2.5	Traceability Queries.	22
Table 2.6	Standards Analytics.	23
Table 3.1	First iteration heuristics and corresponding analytics supporting Question 1.	67
Table 3.2	First iteration heuristics and corresponding analytics supporting Question 2.	68
Table 3.3	Second iteration heuristics and corresponding analytics supporting Question 1.	71
Table 3.4	Second iteration heuristics and corresponding analytics supporting Question 2.	71
Table 3.5	Third iteration heuristics and corresponding analytics supporting Question 1.	72
Table 3.6	Third iteration heuristics and corresponding analytics supporting Question 2.	73
Table 3.7	Additional heuristics and corresponding analytics derived from the extended UPIA-based Generic Model.	83

List of Acronyms and Abbreviations

AMSEWG	Architecture Modeling for Systems Engineering Working Group
CDM	Conceptual Data Model
DM2	DoDAF Meta Model
DOD	Department of Defense
DoDAF	Department of Defense Architecture Framework
DRM	Design Reference Mission
EMMI	Energy, Matter, Material wealth, or Information
GCS	Ground Control Station
HADR	Humanitarian Assistance / Disaster Relief
ICOMs	Inputs, Outputs, Controls and Mechanisms
IDEF0	"Integrated Computer Aided Manufacturing" or Icam DEFinition for Functional Modeling
IT	Information Technology
ITEA	International Test and Evaluation Association
JGF	JSON Graph Format
JTF C2	Joint Task Force Command and Control
LML	Lifecycle Modeling Language
MASC	Mission-Based Architecture for Swarm Composability
MBE	Model Based Engineering
MBSE	Model-Based Systems Engineering

MP	Monterey Phoenix
NAVAIR	Naval Air Systems Command
NPS	Naval Postgraduate School
NRP	Naval Research Program
OV	Operational View
PD21	Project Development for the 21st Century
PI	Principal Investigator
PEO	Program Executive Office
PEO U&W	Program Executive Office Unmanned Aviation and Strike Weapons
RSA	Rational Software Architect
SAT	Satisfiability
SERC	Systems Engineering Research Center
SDL	System Description Language
SE	Systems Engineering
SoS	System of Systems
SysML	Systems Modeling Language
TD	Tactical Disc
UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
UPIA	UML Profile-based Integrated Architecture
USN	U.S. Navy
V&V	Verification and Validation

Acknowledgments

The authors would like to acknowledge with much appreciation the sponsorship of the Naval Research Program. We express gratitude to Chris Wolfgeher, who conducted much of the work underlying the translation of the Generic Model and reproduction in the LML and Innoslate. We are grateful to Mikhail Auguston for the MP event trace generator and model development enabling the testing of executable behavior models in MP-Python, and to Marianna Jones for the program management and editing contributions to the project deliverables. Finally, we are grateful to our NAVAIR sponsor, in particular to Fran Chamberlain and David Fields.

CHAPTER 1:

Introduction

Kristin Giammarco

The purpose of this report is to communicate the results of a Naval Research Program (NRP)-sponsored research study that took place for Naval Air Systems Command (NAVAIR) between October 2016 and January 2018. This introductory chapter explains the motivation for this research, provides the research objectives and tasks, and describes the research methodology. The chapters that follow focus on each respective aspect of the project, including the use of the Conceptual Data Model (CDM) as a language for architecture patterns, a tool-agnostic methodology for deriving and documenting architecture-level patterns, graphical templates for executable behavior models, and virtual testing of system behavior using Monterey Phoenix (MP). Throughout the chapters, architecture patterns are demonstrated and style guidance is provided to facilitate recognition of architectural patterns and concepts.

1.1 Motivation

In 2016, a model reference architecture and development guide (Generic Model) was created by the Architecture Modeling for Systems Engineering Working Group (AMSEWG) for use by Program Executive Office Unmanned Aviation and Strike Weapons (PEO U&W) programs of record at NAVAIR. This reference model provides examples of Department of Defense Architecture Framework (DoDAF) views that seek to normalize the structure of views throughout the PEO, regardless of which program they are developed in, so that all PEO U&W architecture models share a common look and feel, minimizing ambiguity and facilitating stakeholder reviews. Two needs have been recognized to promote the broader adoption of the Generic Model: 1) the Generic Model needs to be tool-agnostic, so that it may be used as guidance by architects using various tools, and 2) the Generic Model needs to contain guidance for executable architecture modeling, which is increasingly being recognized as an important capability for the early verification and validation of static DoDAF models being used as a specification for detailed design.

Naval Postgraduate School (NPS) has a competency in architecture modeling research that will be used to address these needs. Identification of tool-agnostic patterns in architecture models was the subject of the Principal Investigator (PI)'s PhD dissertation in 2012 [3], which identifies recurring patterns of entities and relationships using a high-level DoDAF and Unified Profile for DoDAF and MoDAF (UPDM) –based conceptual data model, culminating in the definition of non-interoperability patterns against which architecture

models may be tested. Preliminary and follow on publications [4] [5] [6] [7] [8] describe the approach in general terms. The tool-independent Lifecycle Modeling Language (LML) is used for its ability to express generic patterns that can be reused throughout different system architectures. The MP language is also relevant in that it promotes the early exposure of modeling errors and unwanted behaviors. MP has a demonstrated ability to expose incorrect, hazardous, unsecure, or otherwise undesirable behaviors in processes and system designs so that they can be removed or mitigated before they manifest in an actual system [9] [8] [10] [11].

1.2 Research Objectives

In response to the needs expressed by the topic sponsor, NPS has established the following research objectives:

1. Assess the reproducibility of the RSA-native Generic Model in other tools, specifically in Innoslate and Monterey Phoenix
2. Produce tool-agnostic patterns and style guidance
3. Test the executability of successfully reproduced Generic Model behavior views in Innoslate and Monterey Phoenix
4. Provide tool-agnostic instructions for creating and analyzing executable activity models

1.3 Research Methodology

The research employed a methodology starting with data collection on the Generic Model, and a request for a NAVAIR architecture model already expressed using the Generic Model for use as a Verification & Validation (V&V) case study. Exports from Rational Software Architect (RSA) tool were provided by NAVAIR and used as the primary source data to reproduce the Generic Model. Upon inspection of the source data, it became clear to the research team that the Generic Model is comprised of a collection of example templates for an architecture model using the Unified Modeling Language (UML) Profile-based Integrated Architecture (UPIA) conceptual data model as a language. The team then proceeded with modeling classes of the UPIA present in the Generic Model in Alloy, to create a single, integrated formal model of the Generic Model schema since only example instances exist. The Alloy language was chosen because the Alloy Analyzer tool can generate example instances of the Generic Model automatically, supporting the research objective to test tool neutrality of the Generic Model. By representing the Generic Model in Alloy, we effectively re-generated the contents of the provided source data (minus some classes that were not modeled due to schedule constraints), plus additional possible scenarios that were not present in the original file. This work is described in Chapter 3.

The next step was to review existing patterns and style guidance for architecture models, which led to the formalization and publication of new and existing heuristics and corresponding analytics expressed at the level of a conceptual data model language. The preliminary work used the LML since it has a deliberately high level of abstraction. As the project progressed, we translated the LML expressions into UPIA (Generic Model), DoDAF Meta Model (DM2), and System Description Language (SDL) CDMs. We then developed some examples of well-formed architecture models that adhere to the defined heuristics. This work is described in Chapter 2.

Since a model of an actual system expressed in Generic Model language was not available, an existing Unmanned Aerial Vehicle (UAV) model from a different project supporting the same sponsor was substituted. The UAV model is represented in Innoslate (a commercial Model-Based Systems Engineering (MBSE) tool) and MP-Python (a Monterey Phoenix tool). Since Innoslate uses LML, the Innoslate entities were labeled according to UPIA conceptual data model equivalents to provide a mapping to the Generic Model. These mappings are available natively in the Innoslate model accompanying the deliverables.

Innoslate was used to develop most of the architecture views (connectivity, hierarchy, traceability, functionality, behavior). In order to facilitate analysis of graphical behavior models, executable behavior models were also developed in Innoslate. A number of generic behavior model templates showing good modeling practices were developed to supplement the Generic Model. This work is described in Chapter 4.

MP-Python was used to test the executability of system behavior in particular. Four Humanitarian Assistance / Disaster Relief (HADR) scenario phases employing a UAV were modeled in MP-Python. This work is described in Chapter 5.

Finally, we evaluated the findings from the research and summarized these as conclusions and recommendations in Chapter 6.

1.4 Student Involvement

NPS Master's and PhD students participated throughout the research, bringing valuable insights to the project in exchange for gaining modeling experience. Chris Wolfgeher, a Systems Engineering (SE) PhD student, was instrumental in the digestion and modeling of the Generic Model source data. Students taking courses SE4151 System Integration and Development (Curricula 311, 721, 581), SE4935 Formal Methods for System Architecting (Curricula 581, 721), and SI4022 System Architecture (Curriculum 721) developed, verified and/or validated the UAV model used as examples for the templates. Two Project Development for the 21st Century (PD21) students in particular, Ernest Lemmert and William Carlson, have adopted thesis topics that supported the templates and patterns aspect of this research. Lemmert is summarizing the strengths and weaknesses found in various

languages. Taking a domain-specific bottom-up approach, Carlson [12] has explored process model structure impact on the decision process for the operation of an acoustic range conducting an underwater vehicle test, which generalizes to systems of interest to NAVAIR.

1.5 Chapter Summary

This chapter introduced the motivation for this research, the research objectives, the research methodology, student involvement, and described where each research objective is addressed in the chapters that follow.

CHAPTER 2:

The CDM as a Language for Architecture Patterns

Kristin Giammarco

This chapter provides a discussion on Conceptual Data Models (CDMs) and several examples, including the UML Profile-based Integrated Architecture (UPIA) used for the Generic Model. Tool-agnostic patterns are provided in Lifecycle Modeling Language (LML), DoDAF Meta Model (DM2), UML Profile-based Integrated Architecture (UPIA), and System Description Language (SDL) as heuristics and corresponding analytics. Examples of well-formed architecture models and some style guidance for some typical architecture views are presented.

2.1 Architecture Modeling Software Analytics

This section contains a version of work authored by Kristin Giammarco appearing as the paper entitled “Architecture Modeling Software Analytics: Model quality and maturity assessment using automated tools” in proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.

This section presents a new collection of architecture modeling software analytics for identification of common errors and omissions committed by both new and experienced users of model-based engineering tools. These analytics are based on architecture modeling heuristics that have been gained through the author’s past 15 years of modeling experience and practice, and are novel in that they can be expressed in a multitude of conceptual modeling languages making them modeling tool-agnostic. The analytics enable students and practitioners to quickly and automatically identify typical architecture model properties impacting model quality and maturity and correct them, enabling time formerly spent on the labor-intensive and error-prone chore of manually finding and fixing the issues to be reinvested in more advanced tasks, such as conducting verification and validation of the design described by the model. The section presents examples expressed in the LML and translates some of them into the UPIA to demonstrate language-neutrality. Deployment of these analytics is recommended to relieve human architects of the manually intensive labor of checking models for these properties that impact model quality and maturity.

2.1.1 Introduction

Architecture is defined by [13] as the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.” An architecture model is an abstraction of the high level design that is used to test

and assess proposed concepts prior to their implementation. The architecture model bridges the expanse between stakeholder expectations (both implicit and explicit) and detailed system design, where the latter is intended to satisfy the former. Experienced architects craft their models from knowledge, experience, and the skilled use of general principles and concepts such as abstraction, refinement, scoping, partitioning [14], coupling, cohesion, modularity, and composability [15]. The architecture model undergoes iteration and inspection with the stakeholders until the stakeholders are satisfied that their expectations are captured. The validated model is a formal specification of requirements, and becomes the starting point for a system's detailed design. Experience has shown that the quality of the detailed design depends on the quality of the high level design described in the architecture model. Model Based Engineering (MBE) practitioners seek cost and time savings through use of automated tools to do the heavy lifting of architecting tasks that are manually intensive, error-prone, and expensive to change in a document-based approach. A significant problem faced by leaders operating in an MBE environment today is the ability to achieve a predictable and consistent level of model quality and maturity across a range of commercially available architecting tools, any one of which may be in use by their hired system architect(s). Even in the case where the leadership or customer can dictate the tool to be used by their architects, there are still no standards that provide model quality and maturity criteria to be used by that tool. This paper provides managers, leaders, and architects operating in an MBE environment with over 40 experience-based analytics that enable them to specify expectations for, then assess, the quality of any model developed for them in any tool that uses a compatible conceptual data model.

The analytics and the heuristics upon which they are based are sorted into categories representing areas in which modeling quality and maturity concerns commonly occur: hierarchy, functional/physical allocation, functional interaction, physical interaction, traceability, and standardization.

The Conceptual Data Model as a language for human experience

Building a model in any tool requires a way to organize the descriptive content. A structured language provides “parking places” for different kinds of information about the system or System of Systems (SoS), its components, the general functions and specific behaviors performed by the components, the interactions among the components, and all the constraints governing the proper interactions within the system and between the system and other systems or components in its environment. This language groups related elements (such as components, or functions, or requirements) into classes, and defines attributes (such as name, number, and description) and allowable relations among the classes (such as component *performs* function, and function *satisfies* requirement). In modeling tools that do not have a native conceptual language like this, system architects often define their own class names and relationships to give the model of the system a predictable and reusable

structure. The product of this definition is a model of the model referred to as a *meta-model* or a *conceptual data model* (CDM). A common CDM provides a standard for use by different organizations and systems so that the same data has the same meaning to everyone involved. Reference [14] describes a data model as that which “specifies data that a system retains, and the relationships among the data.”

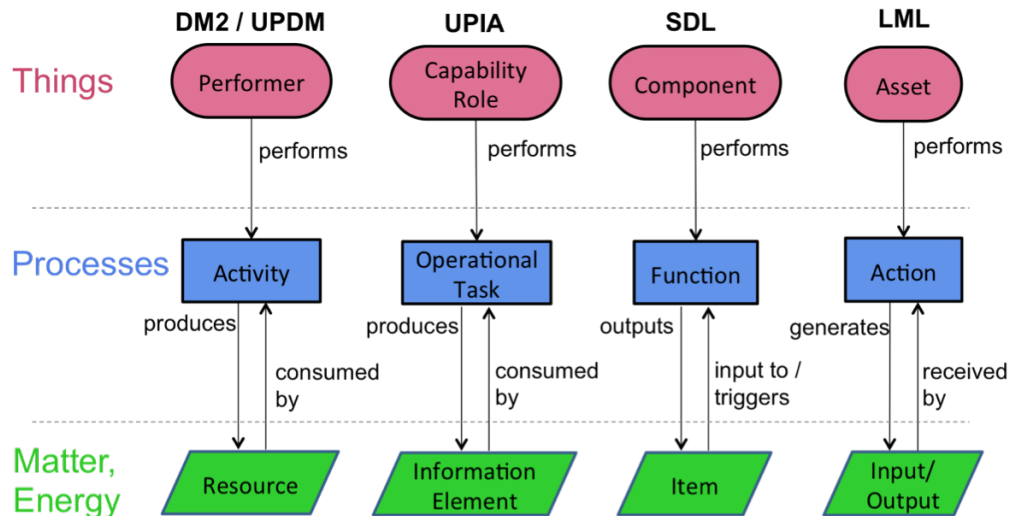


Figure 2.1. Four simple CDM examples. Things doing processes exchanging matter and energy, in the taxonomy of DM2, UPIA, SDL, and LML.

Figure 2.1 shows four small conceptual data model examples used for architecture modeling: the Department of Defense Architecture Framework Meta Model (DM2) [16], the Unified Modeling Language (UML) UML Profile-based Integrated Architecture (UPIA) [17], the System Description Language (SDL) [18], and the Lifecycle Modeling Language (LML) [1]. It demonstrates the conceptual similarities in the terms used in each language that are *things* or physical objects that perform *processes* that exchange *matter or energy*. Because any system can be described in these terms, data models are powerful languages for reasoning about common entities and relationships at a very high level of abstraction. Each CDM has many more classes and relationships than those shown in 2.1. As an example, Figure 2.2 shows a more extensive depiction of LML classes and relationships. CDMs can also be mapped to one another to permit exchange of data between tools using different data models, as shown in Figure 2.3, where a subset of the LML CDM is mapped to a subset of the UPIA CDM in a small example.

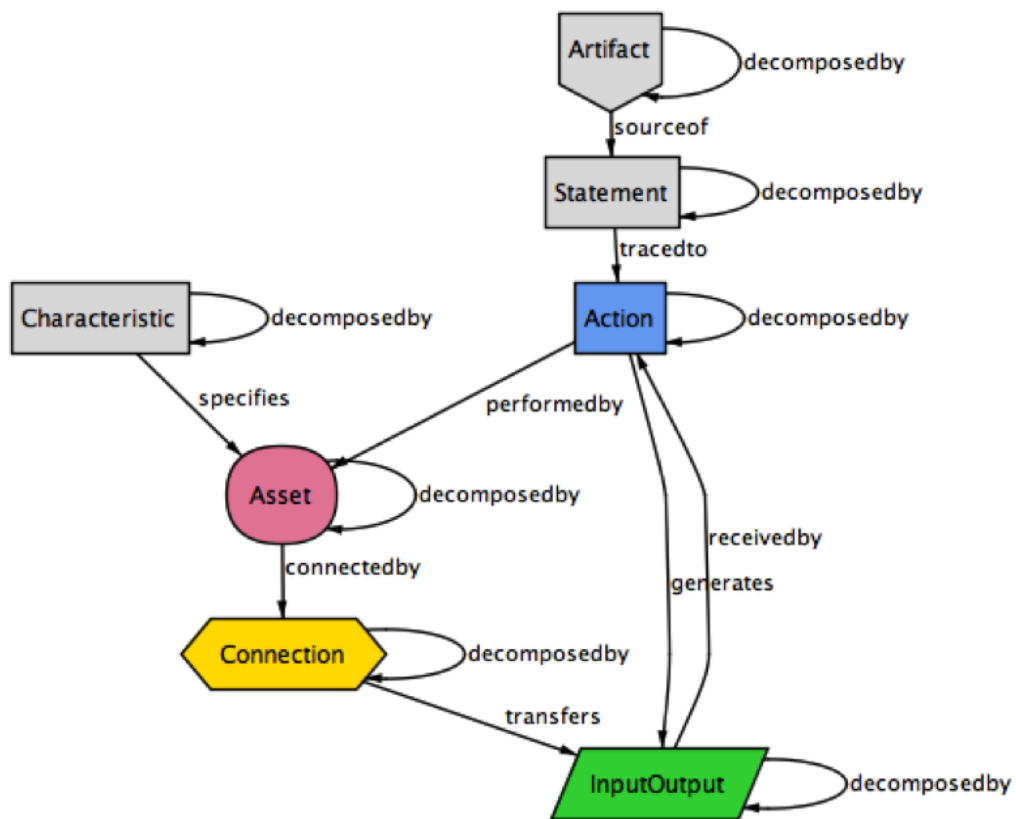


Figure 2.2. Main LML classes and relationships (reciprocal relationships hidden for readability), after Figure 3-1 in [1].

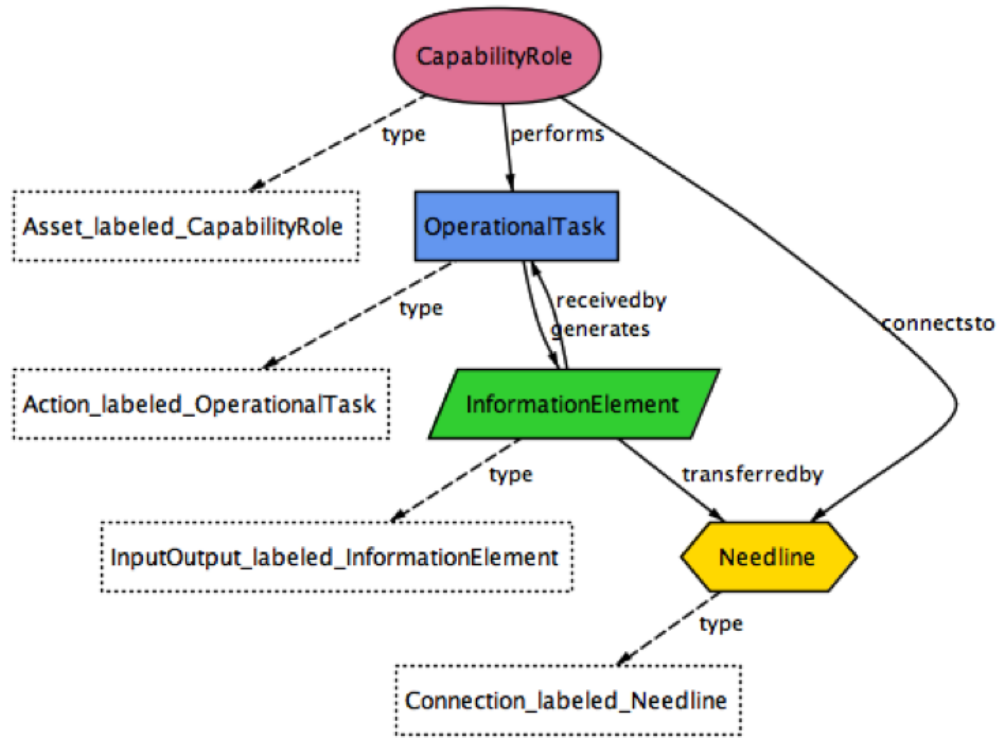


Figure 2.3. Example mapping between the UPIA and LML CDMs.

Related work

While many tools contain automated queries, published work on heuristics for composing analytics across multiple architecture modeling languages and tools appears to be sparse. Logically precise formulas for some heuristics are available in [4] [6] [3]. The following list of heuristics is the most comprehensive to date, expressed in natural language using the LML for its high level concepts, a proposed standard defined in [1].

2.1.2 Architecture Modeling Heuristics and Corresponding Analytics

This section introduces a set of modeling quality and maturity heuristics, as they appear expressed in the LML. The entity terms referred to in the heuristics (e.g., action, asset, input/output) are defined in [1]. The heuristics are general best practices that are experienced-based, some having certain exception cases included with their description. Some heuristics are more or less restrictive than others, allowing the user to specify a combination that best fits the user's intent. The method used for developing CDM-based heuristics is described in [4] and [3].

Each heuristic is followed by one or more corresponding analytics that can be used to measure compliance with the heuristic. Since the heuristics are composed using CDM entities and relationships, they can be codified as desired or undesired properties for any given system model in a tool that has a compatible CDM (see Figures 2.1 and 2.3 for example compatible CDMs). The presence or absence of the codified properties can then be checked using automated queries in models of many different systems. The goal of these queries is to gain insight into typical patterns in model structure that is often difficult or tedious to do without automation. Such queries can be used in the education of new architects who may not be familiar with all the heuristics, as well as provide experienced architects with a means to double-check their work.

A precondition to using the analytic associated with each heuristic is that a relevant entity must be present in the model. For example, "Actions with no child and no parent" should only be checked if there are actions present in the model. A default priority color code of amber is suggested unless otherwise noted. Automated tools implementing these analytics may offer the user the opportunity to fix, ignore once, or ignore all violations of the corresponding heuristic. Decision makers can specify expectations for each in terms of the number goal and/or listed entities (e.g., zero entities listed).

Hierarchy:

1. *Each action / asset / requirement has a child or a parent.* As the model matures, actions, assets and requirements typically will not stand on their own, but appear as part of a hierarchy (either decomposing a parent entity, or being decomposed by a child entity). A hierarchy is useful for grouping and organizing entities of the same class.

No. Number and list of:

- H.1.1 Actions with no child and no parent
- H.1.2 Assets with no child and no parent
- H.1.3 Requirements with no child or parent

2. *No assets / actions / input/outputs / requirements have more than one parent.* Entities with more than one parent belong to multiple hierarchies. This may be desired in some cases, such as when entities are being reused or mapped to multiple taxonomies. Sometimes, however, it is accidental, and results in ambiguity about which is the correct path of parentage.

No. Number and list of:

- H.2.1 Assets with more than one parent
- H.2.2 Actions with more than one parent
- H.2.3 Input/outputs with more than one parent
- H.2.4 Requirements with more than one parent

3. *No asset / action has exactly one child.* Assets, actions, and in general any entity, are usually decomposed into two or more children, or no children at all (if they are the lowest level entity intended). Decomposing an entity into just one other entity implies an equivalency that is often redundant. This flag may be ignored if this is a work in progress (more children will be added), or the one-to-one decomposition relationship is intentional for style reasons.

No. Number and list of:

- H.3.1 Assets with exactly one child
- H.3.2 Actions with exactly one child

4. *No asset / action has more than [userdefined#] children.* For model comprehension, it is strongly recommended to limit the number of entities that appear at a given level. Buede [19] recommends the optimum number is between 3 and 6 actions; Miller [20] uses a 7 plus or minus 2 heuristic. Regrouping a large number of entities at one level in a hierarchy often helps break up the description into manageable chunks. For example, decompose an asset with 24 child assets into 4 new assets, each with 6 child assets. The 4 new assets are used to group the otherwise large number of entities.

No. Number and list of:

- H.4.1 Assets with more than [userdefined#] children
- H.4.2 Actions with more than [userdefined#] children

5. *No asset / action / input/output / requirement is its own parent or child.* To provide a clear definition of hierarchy, an entity may not be included inside itself. It may only include separate entities of the same kind.

No. Number and list of:

- H.5.1 Assets having itself as a child
- H.5.2 Actions having itself as a child
- H.5.3 Input/outputs having itself as a child
- H.5.4 Requirements having itself as a child

Functional/Physical Allocation:

1. *Each action is performed by some asset.* Actions that have not been allocated to an asset lack a physical embodiment. The physical object that will perform the action must eventually be specified. If an action cannot be mapped to a single asset, consider regrouping the actions / assets to support the assignment of actions to specific assets. This regrouping will enable a clear work breakdown structure that delineates "who or what" (asset) is responsible for "doing what" (action).

No. Number and list of:

FPA.1.1 Actions that are not performed by any asset

2. *Each asset performs at least one action.* Assets that have not been related to any action lack functionality. If an asset exists without an associated action, it may be unnecessary, or it may be that its action has been overlooked. Each physical form should be allocated to a corresponding function.

No. Number and list of:

FPA.2.1 Assets that do not perform any action

Functional Interaction:

1. *If any action generates an input/output, it also receives an input/output.* To preserve equilibrium, an action should not generate an output without having received some input at some point in the past. A couple of recognized exception cases in modeling are 1) the use of "stub" actions, during executable modeling development or for simulation debugging, and 2) on a top-level context diagram where the unmodeled input is assumed, and not explicitly shown because it is beyond the scope of the model.

No. Number and list of:

FI.1.1 Actions that generate an input/output without receiving any input/output

2. *If any action receives an input/output, it also generates an input/output.* As a corollary to the previous heuristic, an action should not be able to receive an input without eventually generating some output. Recognized exceptions in modeling are 1) the use of "stub" actions, during executable modeling development or for simulation debugging, and 2) on a top-level context diagram where the unmodeled output is assumed, and not explicitly shown because it is beyond the scope of the model.

No. Number and list of:

FI.2.1 Actions that receive an input/output without generating any input/output

3. *Each action generates or receives at least one input/output.* An action that does not generate or receive any input/outputs is either idle, or intended to define a closed process. More realistically, all actions will interact with at least one other action at some point. Even undesired interactions should be modeled, to help the specification of counteractions.

No. Number and list of:

FI.3.1 Actions that do not generate or receive any input/outputs

4. *No action generates and receives the same input/output.* A “looping” input/output is one that is generated by the same action that consumes it. In an abstract model this may not be a problem, but at a simulation level this is not executable. The looping input/output should be moved one level down into the decomposed view of the action it is leaving and re-entering, to show distinct sub-actions that generate and receive it.
Suggested priority: Red

No. Number and list of:

FI.4.1 Actions that generate and receive the same input/output

5. *Each input/output is generated by some action.* An input/output that is not generated by any action has an unspecified source. If there is no source for the input/output, the implication is that it appears from nowhere or is otherwise never generated by any source action.

No. Number and list of:

FI.5.1 Input/outputs not generated by any action

6. *Each input/output is received by some action.* An input/output that is not received by any action has an unspecified destination. If there is no destination for the input/output, the implication is that it disappears to nowhere or is otherwise never received by any destination action.

No. Number and list of:

FI.6.1 Input/outputs not received by any action

7. *Each input/output is generated or received by some action.* No input/output should be isolated; it should have at least one relationship to an action.

No. Number and list of:

FI.7.1 Input/outputs that are not generated or received by any action

Physical Interaction:

1. *Each asset is connected by at least one conduit.* To support interactions with other assets of any sort, an asset needs to be connected to at least one conduit. (In LML, a Conduit is a type of Connection [1].)

No. Number and list of:

PI.1.1 Assets that are not connected by any conduits

2. *Each conduit connects to at least two disjoint assets.* A conduit is a point-to-point concept that applies a pairwise relationship between connected assets. If a conduit is connected to less than two assets, its specification is incomplete.

No. Number and list of:

PI.2.1 Conduits that connect to fewer than two different assets

3. *Each conduit connects to no more than two disjoint assets.* A conduit is a point-to-point concept that applies a pairwise relationship between connected assets. If a conduit seems to need more than two connection points, consider modeling the conduit as an asset instead.

Suggested priority: Red

No. Number and list of:

PI.3.1 Conduits that connect to more than two different assets

4. *Each conduit transfers at least one input/output.* A conduit with no input/outputs assigned to it may be unnecessary or incomplete.

No. Number and list of:

PI.4.1 Conduits that do not transfer any input/outputs

5. *Each input/output is transferred by at least one conduit.* An input/output that has not been assigned to a conduit may be forgotten in the requirements for that conduit.

No. Number and list of:

PI.5.1 Input/outputs that are not transferred by any conduits

6. *If any two assets exchange some input/output, those assets are connected by at least one common conduit.* Assets that interact through their performed actions should have at least one conduit in common.

No. Number and list of:

PI.6.1 Assets that exchange some input/output, but are not connected by any common conduits

7. *Every input/output exchanged between any two assets is transferred by at least common conduit that connects the two assets.* All input/outputs exchanged between assets need to be assigned to a logical or physical connection between those assets.

No. Number and list of:

PI.7.1 Input/outputs exchanged between assets that are not transferred by a conduit that connects the two assets

8. *Each asset generates an input/output to or receives an input/output from at least one other disjoint asset.* Assets that do not have any interactions are either idle, or intended to define a closed system. Most realistically, all assets will interact with at least one other asset at some point. Even undesired interactions should be modeled, to help the specification of counteractions.

No. Number and list of:

PI.8.1 Assets that do not interact with any other assets through exchange of input/outputs

Traceability:

1. *Every entity is related to some other entity.* Entities that have no relations to other entities may be artifacts of early editing that are often unnecessary to retain. Deletion after verifying there is no longer any need for them is usually recommended.

No. Number and list of:

T.1.1 Entities that are not related to any other entity

2. *Each action is related to some requirement.* Any actions that do not satisfy, verify, or trace to some requirement may be unnecessary or outside the scope of what is needed. Exceptions may apply to modeled actions or functions of external systems outside the scope of the requirement specification for the system under design. Another exception case is made for a root action at the top of an action or function hierarchy, since such an entity is typically used for context only.

Suggested priority: Red

No. Number and list of:

T.2.1 Actions that do not satisfy/verify/trace to any requirement

3. *Each asset that performs an action is traced from each requirement that is satisfied by that performed action.* Assets are related to requirements through the actions they perform.

No. Number and list of:

T.3.1 Assets that are not traced from any requirements

4. *Each input/output satisfies some requirement.* Any input/outputs that do not satisfy some requirement may be unnecessary or outside the scope of what is needed.
Suggested priority: Red

No. Number and list of:

T.4.1 Input/outputs that are not satisfied by any requirement

5. *Each conduit satisfies some requirement.* Any conduits that do not satisfy some requirement may be unnecessary or outside the scope of what is needed.

No. Number and list of:

T.5.1 Conduits that do not satisfy some requirement

6. *All leaf-level requirements are satisfied by at least one entity in the Action, Conduit, Asset, or Input/output class.* A requirement that is not satisfied by any modeled entity is either unnecessary or has not yet been specified in the architecture model.
Suggested priority: Red

No. Number and list of:

T.6.1 Leaf-level requirements that are not satisfied by any action, asset, input/output or conduit

Standards:

1. *Each asset references some standard-labeled artifact.* Assets should reference at least one standard-labeled artifact to denote rules, policies, guidelines or protocols to which the asset is expected to adhere. (In LML, standards may be attached to an Artifact entity and assigned a “Standard” label.)

No. Number and list of:

S.1.1 Assets that do not reference any Standard-labeled artifact

2. *Each action references some standard-labeled artifact.* Actions should reference at least one standard-labeled artifact to denote rules, policies, guidelines or protocols to which the action is expected to adhere.

No. Number and list of:

S.2.1 Actions that do not reference any Standard-labeled artifact

3. *Each input/output references some standard-labeled artifact.* Input/outputs should reference at least one standard-labeled artifact to denote rules, policies, guidelines or protocols to which the input/output is expected to adhere.

No. Number and list of:

S.3.1 Input/outputs that do not reference any Standard-labeled artifact

4. *Each conduit references some standard-labeled artifact.* Conduits should reference at least one standard-labeled artifact to denote rules, policies, guidelines or protocols to which the conduit is expected to adhere.

No. Number and list of:

S.4.1 Conduits that do not reference any Standard-labeled artifact

5. *If any two assets exchange some input/output through performed actions, those assets satisfy some common standardizing requirement.* Assets that interact through their performed actions should have at least one standard in common, so that they are consistently constrained to support the interaction.

No. Number and list of:

S.5.1 Assets that interact with each other through exchange of input/outputs, but satisfy no common standardizing requirement

6. *If any input/output generated by any action a_A of any asset p_A is received by some action a_B of another asset p_B , then a_A and a_B reference at least one common standard-labeled artifact that is also referenced by both p_A and p_B .* Actions that interact through exchange of input/output(s) should have at least one standard in common with the assets that perform them, so that they are consistently constrained to support the interaction.

No. Number and list of:

S.6.1 Actions that exchange some input/output that reference no common standard-labeled artifact with the assets performing those actions

7. *Every exchanged input/output between any two assets reference some standard-labeled artifact that is also referenced by those assets.* An input/output exchanged between interacting assets should have at least one standard in common with its

source and destination assets, so that the input/output and the assets are consistently constrained to support the interaction.

No. Number and list of:

S.7.1 Input/outputs exchanged between assets that reference no common standard-labeled artifact with those assets

8. *Every conduit that connects any two assets and transfers an input/output between those assets references some standard-labeled artifact that is also referenced by those assets.* A conduit that connects interacting assets should have at least one standard in common with the assets that they connect, so that the conduit and connected assets are consistently constrained to support the interaction.

No. Number and list of:

S.8.1 Conduits that connect assets transferring some input/output that reference no common standard-labeled artifact with those assets

2.2 Translation of Analytics into Other CDM Languages

This section contains translations for the analytics presented in the previous section into DM2/UPDM, UPIA, and SDL to demonstrate their conceptual universality. Due to some ambiguity and/or variability in source documents for some of the languages, some of the translations may need to be fine tuned in follow on work. Translations may exist for additional CDMs that are beyond the scope of the current study.

Table 2.1. Hierarchy Analytics.

No.	LML	DM2 / UPDM	UPIA	SDL
H.1.1	Actions with no child and no parent	Activities with no child and no parent	Operational tasks with no child and no parent	Functions with no child and no parent
H.1.2	Assets with no child and no parent	Performers with no child and no parent	Capability roles with no child and no parent	Components with no child and no parent
H.1.3	Requirements with no child or parent	-	-	Requirements with no child and no parent
H.2.1	Assets with more than one parent	Performers with more than one parent	Capability roles with more than one parent	Components with more than one parent

Continuation of Table 2.1				
H.2.2	Actions with more than one parent	Activities with more than one parent	Operational tasks with more than one parent	Functions with more than one parent
H.2.3	Input/outputs with more than one parent	Resources with more than one parent	Information elements with more than one parent	Items with more than one parent
H.2.4	Requirements with more than one parent	-	-	Requirements with more than one parent
H.3.1	Assets with exactly one child	Performers with exactly one child	Capability roles with exactly one child	Components with exactly one child
H.3.2	Actions with exactly one child	Activities with exactly one child	Operational tasks with exactly one child	Functions with exactly one child
H.4.1	Assets with more than [userdefined#] children	Performers with more than [userdefined#] children	Capability roles with more than [userdefined#] children	Components with more than [userdefined#] children
H.4.2	Actions with more than [userdefined#] children	Activities with more than [userdefined#] children	Operational tasks with more than [userdefined#] children	Functions with more than [userdefined#] children
H.5.1	Assets having itself as a child	Performers having itself as a child	Capability roles having itself as a child	Components having itself as a child
H.5.2	Actions having itself as a child	Activities having itself as a child	Operational tasks having itself as a child	Functions having itself as a child
H.5.3	Input/outputs having itself as a child	Resources having itself as a child	Information elements having itself as a child	Items having itself as a child
H.5.4	Requirements having itself as a child	-	-	Requirements having itself as a child
End of Table				

Table 2.2. Functional/Physical Allocation Analytics.

No.	LML	DM2 / UPDM	UPIA	SDL
FPA.1.1	Actions that are not performed by any asset	Activities that are not performed by any performer	Operational tasks that are not performed by any capability role	Functions that are not performed by any component
FPA.2.1	Assets that do not perform any action	Performers that do not perform any activity	Capability roles that do not perform any operational task	Components that do not perform any function
End of Table				

Table 2.3. Functional Interaction Analytics.

No.	LML	DM2 / UPDM	UPIA	SDL
FI.1.1	Actions that generate an input/output without receiving any input/output	Activities that produce a resource without consuming any resource	Operational tasks that produce an information element without consuming any information element	Functions that produce an item without consuming any item
FI.2.1	Actions that receive an input/output without generating any input/output	Activities that consume a resource without producing any resource	Operational tasks that consume an information element without producing any information element	Functions that consume an item without producing any item
FI.3.1	Actions that do not generate or receive any input/outputs	Activities that do not produce or consume any resources	Operational tasks that do not produce or consume any information elements	Functions that do not produce or consume any items
FI.4.1	Actions that generate and receive the same input/output	Activities that produce and consume the same resource	Operational tasks that produce and consume the same information element	Functions that produce and consume the same item

Continuation of Table 2.3				
FI.5.1	Input/outputs not generated by any action	Resources not produced by any activity	Information elements not produced by any operational task	Items not produced by any function
FI.6.1	Input/outputs not received by any action	Resources not consumed by any activity	Information elements not consumed by any operational task	Items not consumed by any function
FI.7.1	Input/outputs that are not generated by or received by any action	Resources that are not produced by or consumed by any activity	Information elements that are not produced by or consumed by any operational task	Items that are not produced by or consumed by any function
End of Table				

Table 2.4. Physical Interaction Analytics.

No.	LML	DM2 / UPDM	UPIA	SDL
PI.1.1	Assets that are not connected by any conduits	Performers that are not connected to any connectors	Capability Roles that are not connected to any needlines	Components that are not connected to any links
PI.2.1	Conduits that connect fewer than two different assets	Connectors that connect fewer than two different performers	Needlines that connect fewer than two different capability roles	Links that connect fewer than two different components
PI.3.1	Conduits that connect more than two different assets	Connectors that connect more than two different performers	Needlines that connect more than two different capability roles	Links that connect more than two different components
PI.4.1	Conduits that do not transfer any input/outputs	Connectors that do not transfer any resources	Needlines that do not transfer any information elements	Links that do not transfer any items
PI.5.1	Input/outputs that are not transferred by any conduits	Resources that are not transferred by any connectors	Information elements that are not transferred by any needlines	Items that are not transferred by any links

Continuation of Table 2.4				
PI.6.1	Assets that exchange some Input/output, but are not connected by any common conduits	Performers that exchange some resource, but are not connected to any common connectors	Capability Roles that exchange some information element, but are not connected to any common needlines	Components that exchange some item, but are not connected to any common links
PI.7.1	Input/outputs exchanged between assets that are not transferred by a conduit that connects the two assets	Resources exchanged between performers that are not transferred by a connector that connects the two performers	Information elements exchanged between capability roles that are not transferred by a needline that connects the two capability roles	Items exchanged between components that are not transferred by a link that connects the two components
PI.8.1	Assets that do not interact with any other assets through exchange of input/outputs	Performers that do not interact with any other performers through exchange of resources	Capability roles that do not interact with any other capability roles through exchange of information elements	Components that do not interact with any other components through exchange of items
End of Table				

Table 2.5. Traceability Queries.

No.	LML	DM2 / UPDM	UPIA	SDL
T.1.1	Entities that are not related to any other entity	Entities that are not related to any other entity	Entities that are not related to any other entity	Entities that are not related to any other entity
T.2.1	Actions that do not satisfy/verify/trace to any requirement	-	-	Functions that are not based on any requirement

Continuation of Table 2.5				
T.3.1	Assets that are not traced from any requirement	-	-	Components that are not specified by any requirement
T.4.1	Input/outputs that do not satisfy any requirement	-	-	Items that are not specified by any requirement
T.5.1	Conduits that do not satisfy any requirement	-	-	Links that are not specified by any requirement
T.6.1	Leaf-level requirements that are not satisfied by any action, asset, input/output, or conduit	-	-	Leaf-level requirements that are not the basis of or do not specify any function, component, item, or link
End of Table				

Table 2.6. Standards Analytics.

No.	LML	DM2 / UPDM	UPIA	SDL
S.1.1	Assets that do not reference any standard-labeled artifact	Performers that are not subject to any standard	Capability roles that are not subject to any standard	Components that are not specified by any standard-labeled requirement
S.2.1	Actions that do not reference any standard-labeled artifact	Activities that are not subject to any standard	Operational tasks that are not subject to any standard	Functions that are not based on any standard-labeled requirement
S.3.1	Input/outputs that do not reference any standard-labeled artifact	Resources that are not subject to any standard	Information elements that are not subject to any standard	Items that are not specified by any standard-labeled requirement

Continuation of Table 2.6				
S.4.1	Conduits that do not reference any standard-labeled artifact	Connectors that are not subject to any standard	Needlines that are not subject to any standard	Links that are not specified by any standard-labeled requirement
S.5.1	Assets that interact with each other through exchange of input/outputs, but satisfy no common standardizing requirement	Performers that interact with each other through exchange of resources, but are not subject to a common standard	Capability roles that interact with each other through exchange of information elements, but are not subject to a common standard	Components that interact with each other through exchange of items, but are not specified by a common standard-labeled requirement
S.6.1	Actions that exchange some input/output that reference no common standard-labeled artifact with the assets performing those actions	Activities that exchange some resource that are not subject to any common standard with the performers performing those activities	Operational tasks that exchange some information element that are not subject to any common standard with the capability roles performing those operational tasks	Functions that exchange some item that are not the basis of any common standard-labeled requirement with the components performing those functions
S.7.1	Input/outputs exchanged between assets that reference no common standard-labeled artifact with those assets	Resources exchanged between performers that are not subject to any common standard with those performers	Information elements exchanged between capability roles that are not subject to any common standard with those capability roles	Items exchanged between components that are not specified by any common standard-labeled requirement with those components

Continuation of Table 2.6				
S.8.1	Conduits that connect assets transferring some input/output that reference no common standard-labeled artifact with those assets	Connectors that connect performers transferring some resource that are not subject to any common standard with those performers	Needlines that connect capability roles transferring some information element that are not subject to any common standard with those capability roles	Links that connect components transferring some item that are not specified by any common standard-labeled requirement with those components
End of Table				

2.3 Examples of Well-formed Architecture Models

This section shows some examples of architecture models in both a pattern format and a case study format. The purpose of this section of the report is to provide examples of some well-formed models and a brief description of the utility of each view. The patterns are generic, whereas the case study examples were developed from the Humanitarian Assistance / Disaster Relief (HADR) Design Reference Mission (DRM) in Appendix G of [21]. The Innoslate tool and the LML were used to produce the examples in this section, but as discussed elsewhere in the report, the same guidelines apply in other tools and languages.

2.3.1 High Level Concept Graphic

A high level operational concept graphic (referred to in the DoDAF as an Operational View (OV)-1) is often one of the first views to be developed, because it highlights the main operational nodes and interesting or unique aspects of operational capabilities. Its main utility is as a facilitator of communication between key stakeholders and decision-makers over a broad depiction of the operational concept. It is intended to convey the system's operational purpose.

Figure 2.4 shows a pattern for a high level operational concept graphic. Because the Navy, other services and industry are moving to using MBSE approaches to capture architecture descriptions, this graphic was developed in an architecture tool that stores the entities shown in the graphic and their relationships formally. The boxes in Figure 2.4 are assets, and the lightning bolt connecting them is a conduit that connects both assets. At the highest notional level, a system interacts with its environment. In an actual system design, the environment is typically characterized along with external systems to show other objects with which the

system interacts. In a SoS design, there may be many systems shown interacting on an OV-1.



Figure 2.4. Pattern for a high level operational concept graphic. Two assets (boxes) are connected to each other by a conduit (depicted here in the style of a yellow lightning bolt). The background is an image of Mars that has been color-filtered for an Earth-like atmosphere. Image source: NASA.

Figure 2.5 shows the systems of a SoS interacting with each other and with an environment. It is important to show the system interacting with a physical environment, as the physical environment of the system may present many constraints on the system design. Pictured in this figure are the operational nodes enumerated in the DRM:

- Environment
- Joint Task Force Command and Control (JTF C2)
- US Navy Ships
- Helicopters
- Ground Control Station (GCS) Operator
- Ground Crew
- Host Nation
- Non-Government Organizations
- UAV
- Launcher
- GCS

- Recoverer

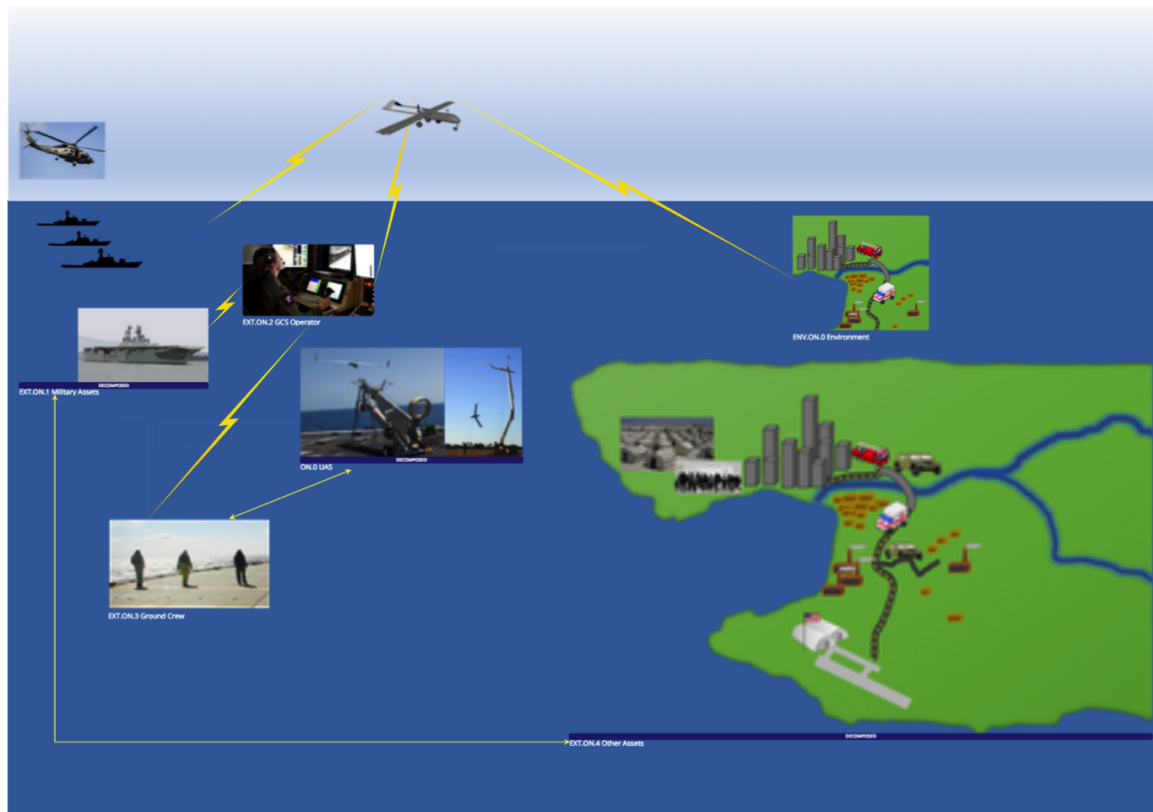


Figure 2.5. Example of a high level operational concept graphic. Assets (boxes) are connected to each other by conduits (yellow lightning bolts or bidirectional arrows, in this case). The environment is explicitly modeled as an entity that can have interactions with systems.

Because there are quite a few assets involved in conveying the operational concept, some of them are shown grouped together in Figure 2.5 with a blue "Decomposed" bar beneath the name of each group. For this architecture, the following groupings have been made:

- Environment
- Military Assets
 - JTF C2
 - US Navy Ships
 - Helicopters
- GCS Operator
- Ground Crew
- Other Assets

- Host Nation
 - Non-Government Organizations
- Unmanned Aerial System (UAS)
 - UAV
 - Launcher
 - GCS
 - Recoverer

The image used for each group in Figure 2.5 is a composition of images of the members of each group, to enable this high level graphic to include all the entities that are relevant to discuss at the very highest level of the architecture.

2.3.2 Hierarchy Charts

A hierarchy chart is a simple and familiar way to view the structure of an architecture. Two common hierarchies are functional and physical hierarchies. The figures below illustrate good hierarchy structures.

The system and environment shown in Figure 2.6 are the same system and environment shown in Figure 2.4. Likewise, the first row of assets under "0 Physical Context" in Figure 2.7 are the same assets depicted in Figure 2.5. Numbering conventions differ based on organizational standards or architecture preferences. The examples here use "X" or "EXT" as a preface for entities external to the system under design. The terms "internal" and "external" are relative, however, depending on whether a system or a SoS perspective is taken. A special prefix "ENV" is used to indicate the physical environment. Prefix "C" is used here to indicate an asset is a component, prefix "ON" to mean an asset is an operational node, prefix "F" to mean an action is a function, and prefix "OA" to mean an action is an operational activity. These prefixes illustrate how numbering conventions can be used to give a quick clue about the type of entity it labels without requiring separate classes for storing solution-neutral and solution-oriented entities, respectively.

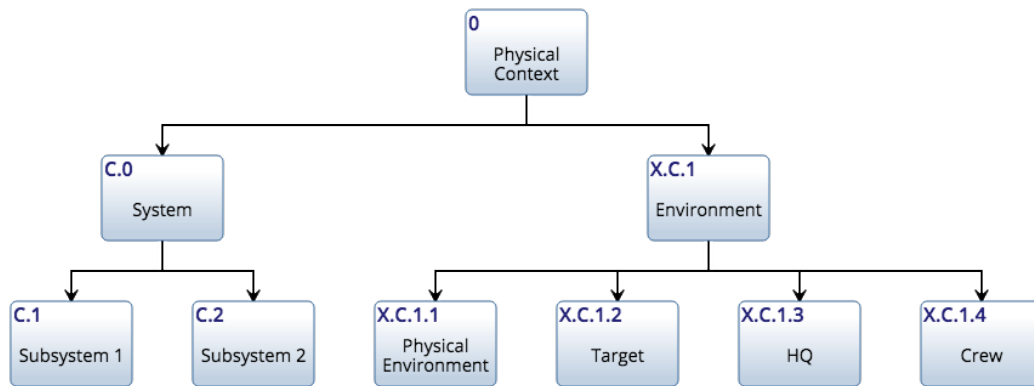


Figure 2.6. Pattern for a physical hierarchy. This view shows the composition of the system and the environment assets depicted in the high level operational graphic. No asset in this chart has more than one parent or exactly one child.

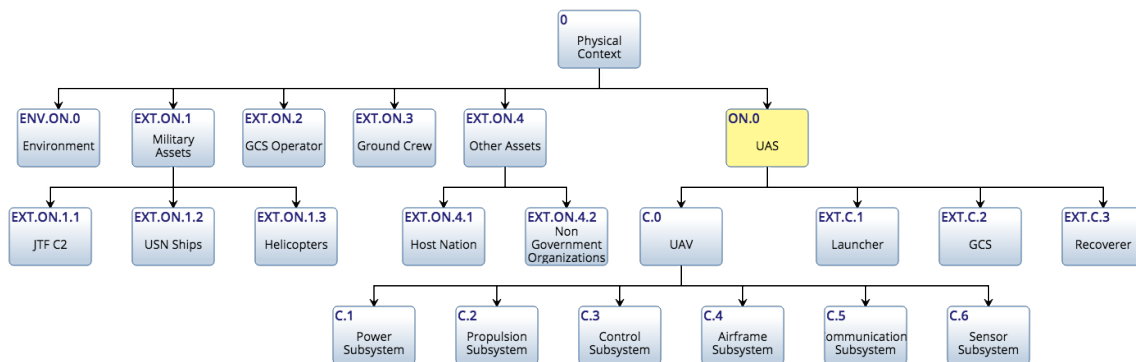


Figure 2.7. Example of a physical hierarchy. This view shows the composition of the Military Assets, Other Assets, UAS, and even the UAV depicted in the high level operational graphic. No asset in this chart has more than one parent or exactly one child.

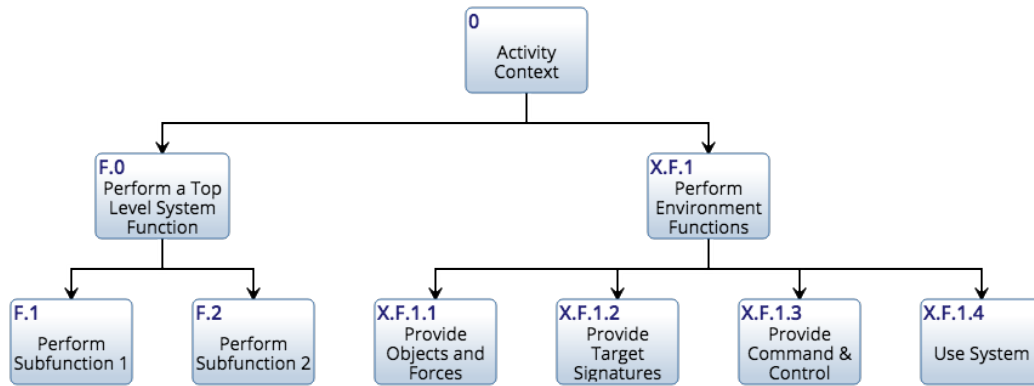


Figure 2.8. Pattern for a functional hierarchy. This view shows the corresponding functions (actions) for the System and the Environment assets and their subsystems. No action in this chart has more than one parent or exactly one child.

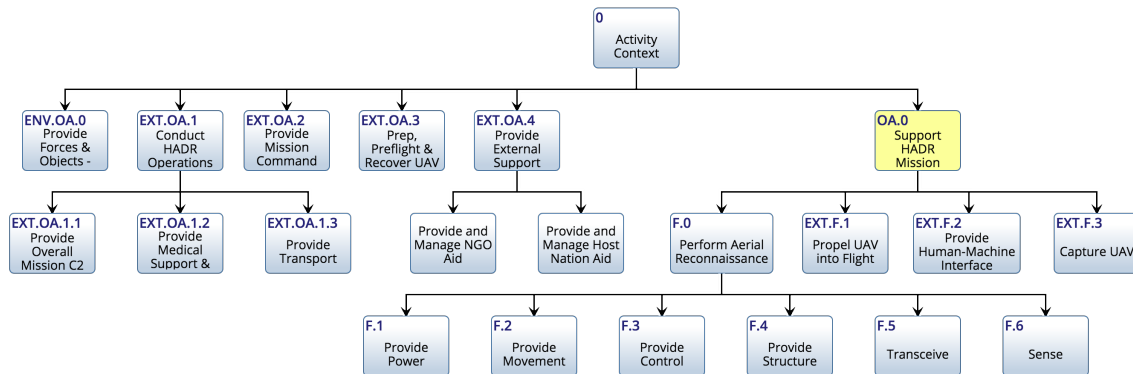


Figure 2.9. Example of a functional hierarchy. This view shows the hierarchy of corresponding functions (actions) for the HADR assets. No action in this chart has more than one parent or exactly one child.

2.3.3 Traceability Spider Diagrams

Spider diagrams can be used to show hierarchy as well, using the "decomposed by" relations. The following figures show an additional use for spider diagrams - showing traceability. In this case, we use them to show the mapping of form to function (assets to corresponding actions). To do this, we graph both "decomposed by" and "performed by" relations for an action of interest, as shown in the figures below. If we have an asset of interest, then we

would graph with "decomposed by" and the inverse relation "performs" to arrive at a similar diagram.

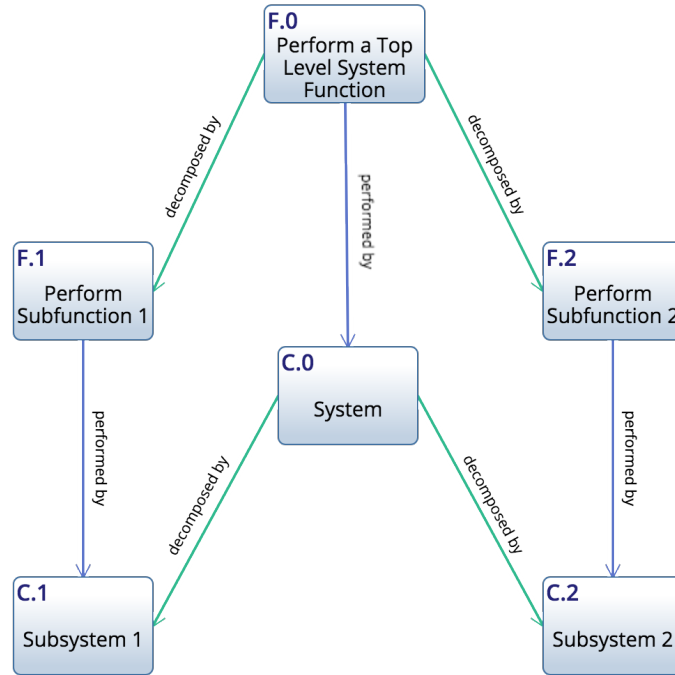


Figure 2.10. Pattern for a spider traceability graph. This view shows the function-to-form mapping for a top level system. Each action is performed by an asset (blue arrows).

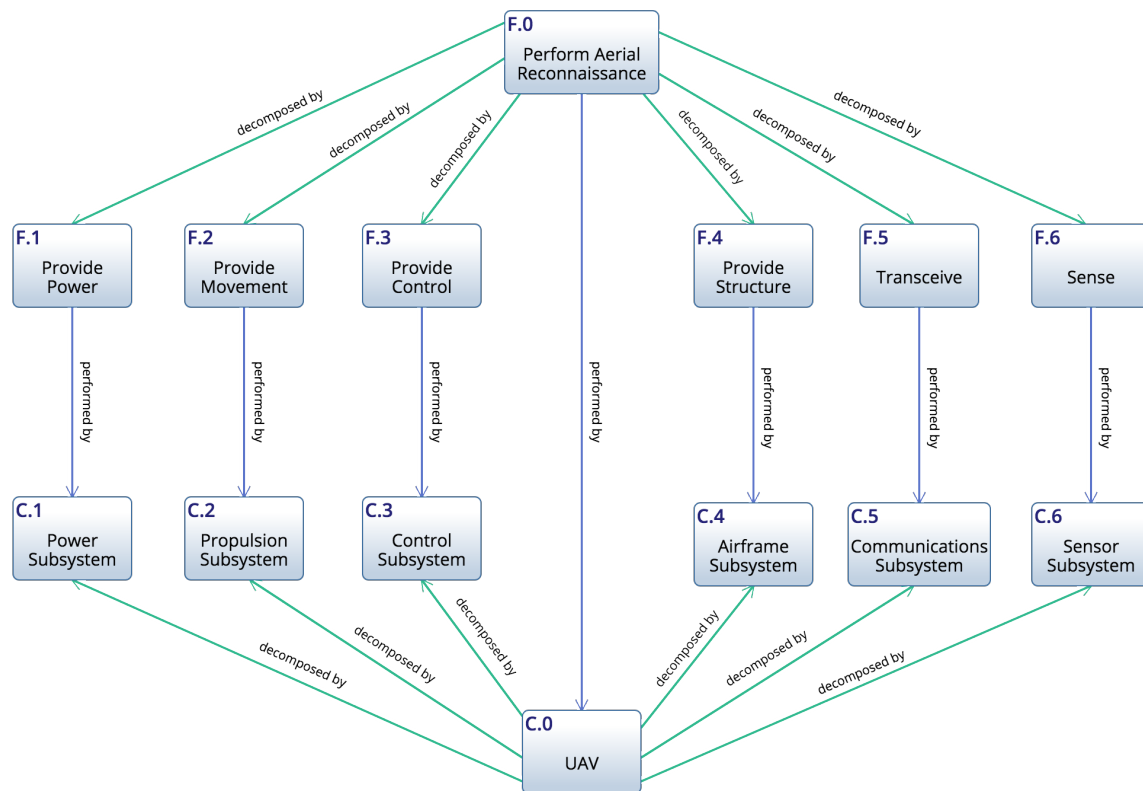


Figure 2.11. Example of a spider traceability graph. This view shows the function-to-form mapping for the UAV employed in the HADR mission. Each action is performed by an asset (blue arrows).

2.3.4 Use Case Diagrams

Use case diagrams provide an overview of the main behavioral functions provided by the system to external users and other actors. System use cases are named from the actor's perspective, rather than from the system's perspective. The external actors that participate in each use case are connected by lines or arrows to the use case.

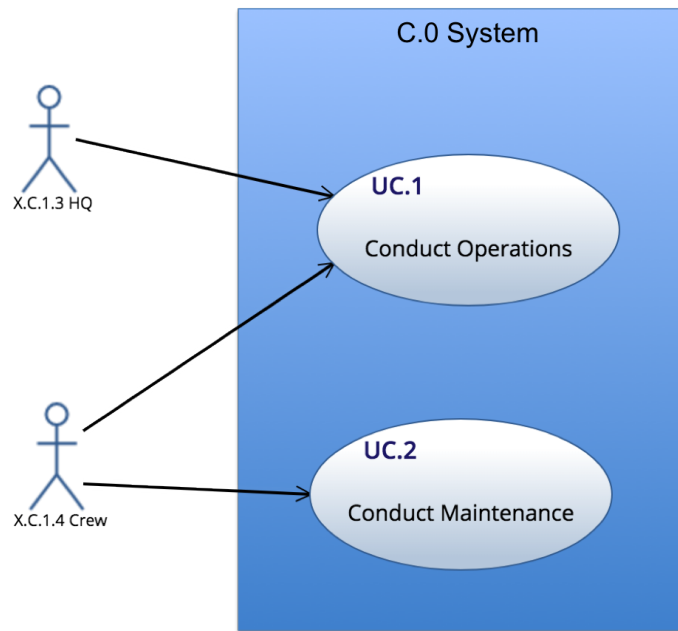


Figure 2.12. Pattern for a use case diagram. The use case diagram shows which actors participate in which use cases. Use cases (ovals) are actions that are performed by actors (assets), shown here as stick figures.

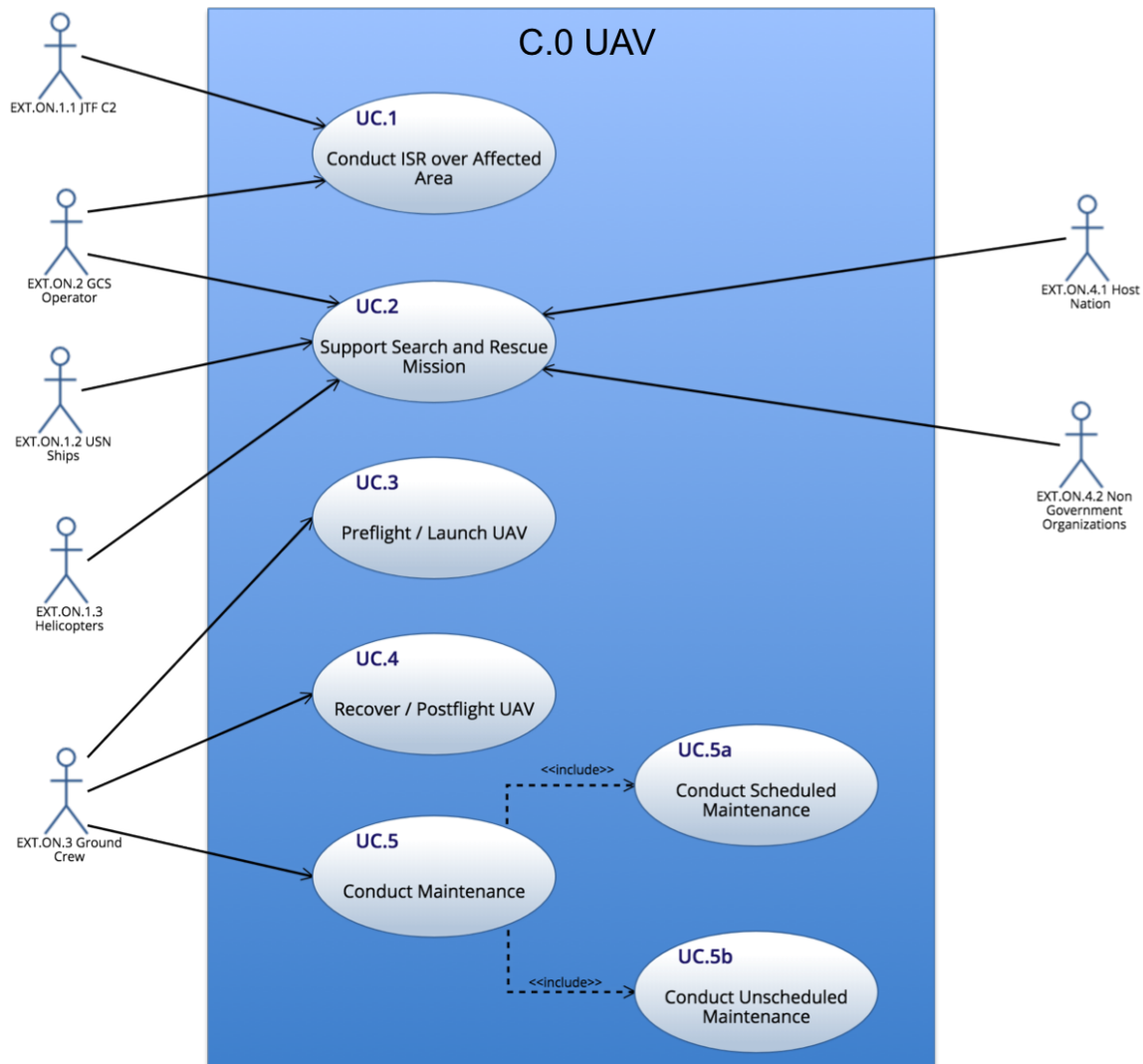


Figure 2.13. Example of a use case diagram. Each action (use case) is performed by at least one asset (actor). The maintenance use case is shown as including two different types of maintenance: scheduled and unscheduled, each of which involves different procedures.

2.3.5 Action Diagrams

In action diagrams, each branch (or swim lane) is allocated an asset. The allocated asset name appears as the branch label. Each branch contains a sequence of actions (blue boxes) performed by the branch asset. Interactions among actions on different branches are defined using input/outputs (green parallelograms). In general, each branch (asset) should have at

least one action, and each input/output should be generated and received by an action. Notice, however, in the figures that follow, that not all actions receive an input/output, and not all actions generate an input/output. These figures use the minimum number of input/outputs needed for specifying the order of the actions. If it is desired to follow these heuristics strictly, then more input/outputs would be added to these diagrams. One reason to do this is to thoroughly show data/information flow by the names of the input/outputs. One reason not to do this is to simplify the diagram for behavioral analysis of just one segment of a design. If the very first action in an executed sequence has a triggering input/output as part of its specification, this will prevent its execution in simulation and the model should be run from a broader scope (including the preceding action that generated the triggering input/output), or by inserting a stub action to generate that input/output. In general, every input/output received by an action must be generated by some action in order for the receiving action to complete in simulation. Adherence to a well-defined set of heuristics can therefore help an architect avoid common errors that prevent executable models from running to completion.

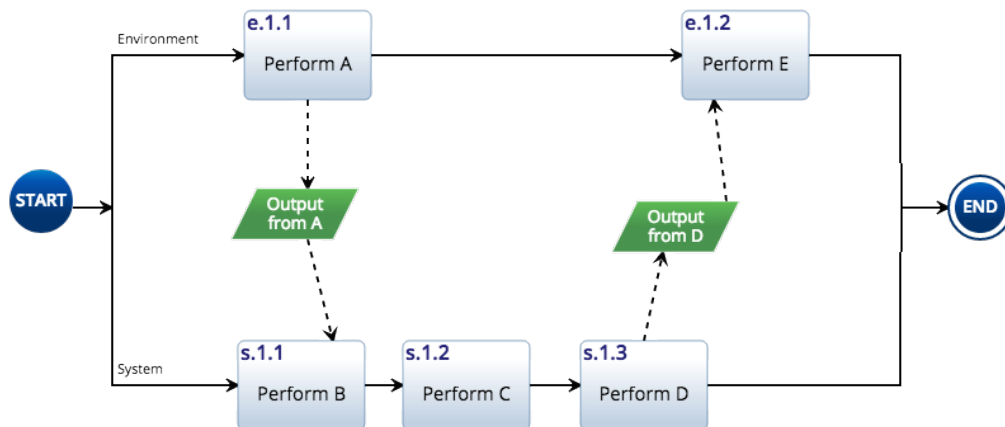


Figure 2.14. Pattern for an action diagram. Each asset performs at least one action; each input/output is both generated and received by an action.

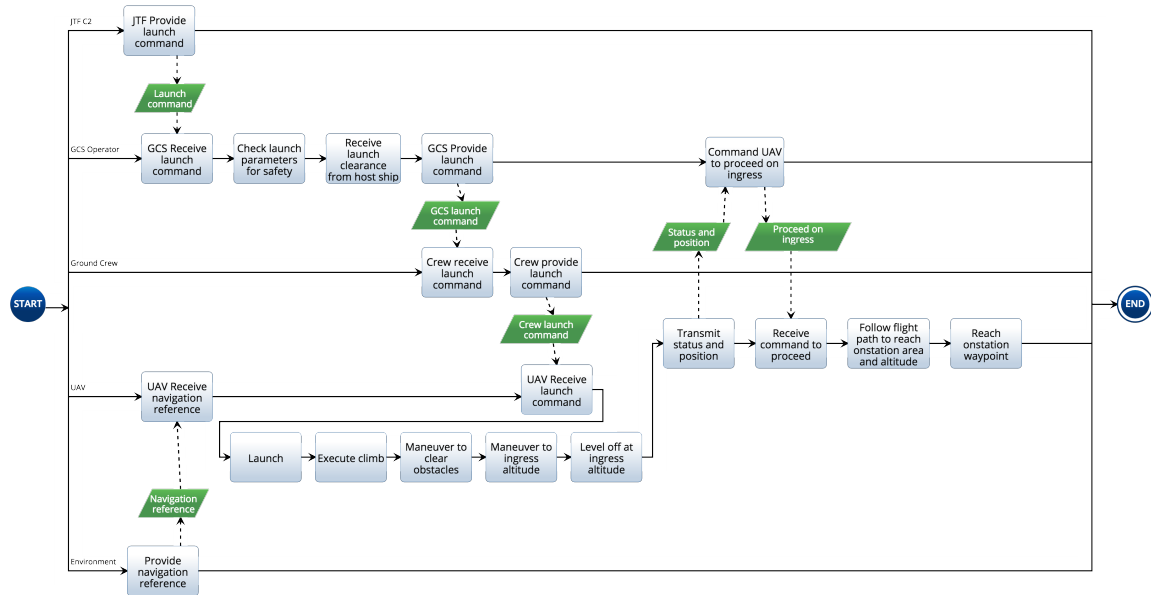


Figure 2.15. Example of an action diagram. Each asset performs at least one action; each input/output is both generated and received by an action.

2.3.6 Sequence Diagrams

A straightforward way to graphically visualize a sequence of interactions is with a sequence diagram, which traces through the sequence of events or input/outputs in a particular use case. Sequence diagrams can be elaborate, showing concurrent events and more than one possible outcome of a use case, but the convention used here employs a separate sequence diagram for each possible use case scenario outcome. The reason for this convention is mainly because the corresponding action diagrams already have concurrency, alternate, and other logical constructs, but do not as easily show each separate path as clearly and legibly as can a sequence diagram.

In a sequence diagram, time proceeds from top to bottom along the lifelines of the actors involved in the use case. The branch assets (actors) from the action diagram are shown in this view as boxes atop each lifeline. The interactions are shown as arrows between lifelines. These interactions may be labeled with verb-oriented events or noun-oriented input/outputs, depending on the preferred convention and notation.

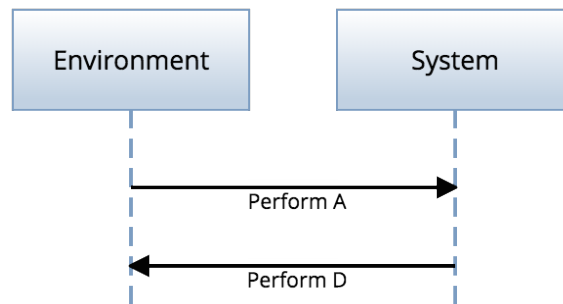


Figure 2.16. Pattern for an event trace -style sequence diagram. The arrows are labeled with verb-oriented events, or actions. (Perform B and Perform C are not shown because they are internal actions of the system, from Figure 2.14.)

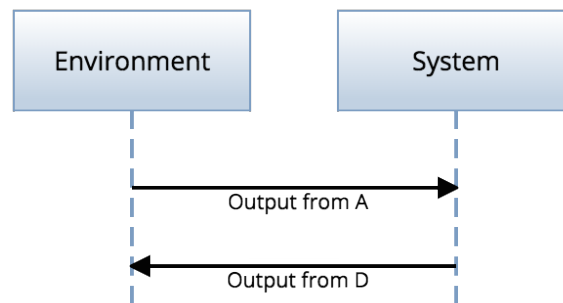


Figure 2.17. Pattern for an input/output-style sequence diagram. Here, the arrows are labeled with noun-oriented input/outputs.

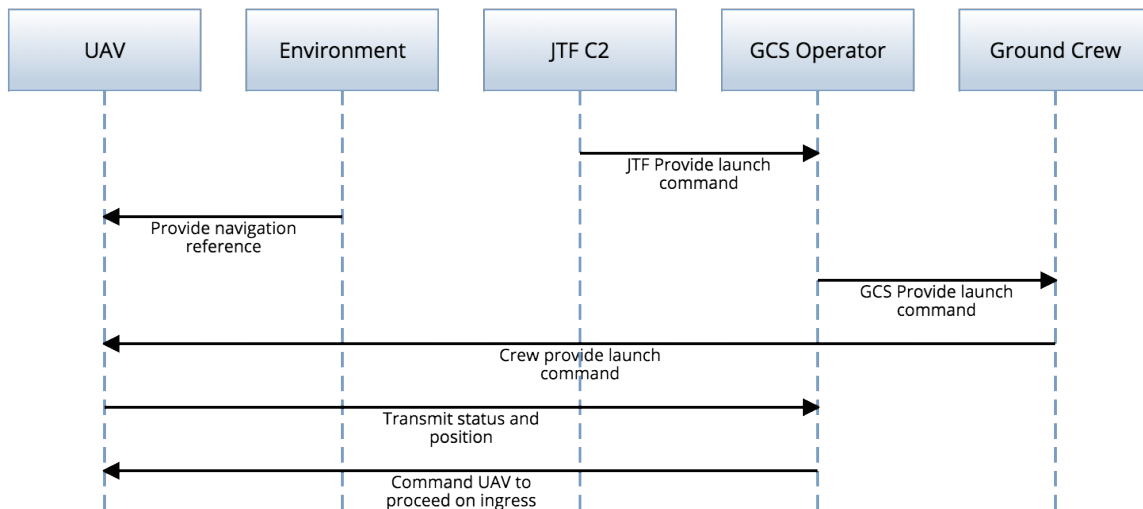


Figure 2.18. Example of an event trace -style sequence diagram. The arrows are labeled with verb-oriented events, or actions, from the corresponding action diagram.

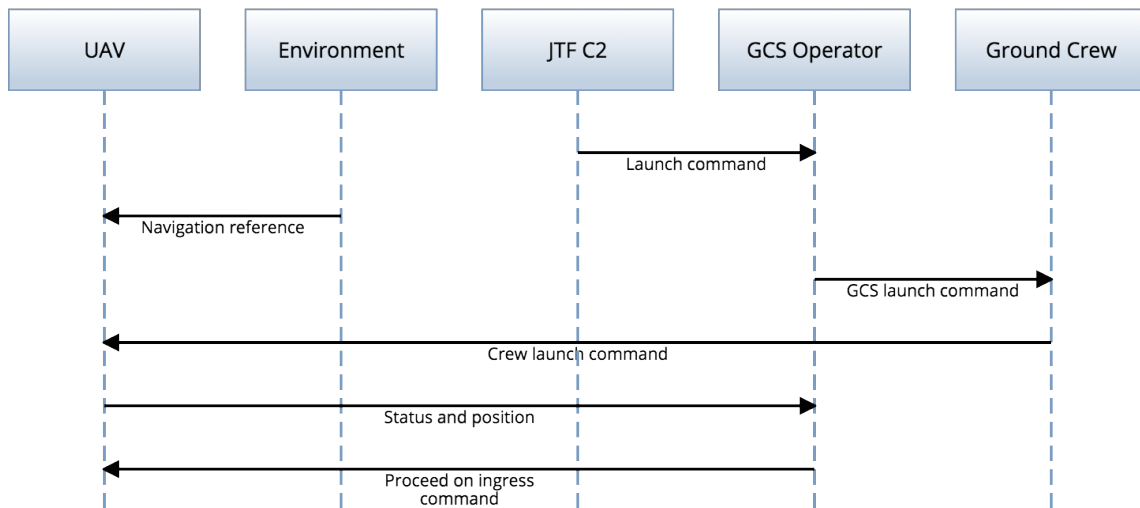


Figure 2.19. Example of an input/output-style sequence diagram. The arrows are labeled with noun-oriented input/outputs from the corresponding action diagram.

2.3.7 IDEF0 A-0 Diagrams

As its namesake suggests, the IDEF0 [2] is a standard that was originally developed for describing manufacturing processes ("Integrated Computer Aided Manufacturing" or Icam DEFinition for Functional Modeling (IDEF0)). Other disciplines, including systems engineering, adopted the notation for modeling other types of systems, using its function-oriented, decomposable views to map function to function through interactions, as well as function to form. The IDEF0 diagram, applied with certain style conventions, is useful for showing high level functional interactions within a system and between a system and its environment. While IDEF0 functions and interactions are unsequenced, activity models show the time-sequenced order of activities as they interact. The IDEF0 shows the architecture at a higher level of abstraction than do activity models like the action diagram.

Following the modeling conventions herein, an IDEF0 diagram has boxes representing *functions* (actions) that are performed by *mechanisms* (assets). The functions transform *inputs* governed by *controls* into *outputs* (input/outputs). The inputs, outputs, controls and mechanisms for a function are often referred to as ICOMs for brevity. Input arrows always point into a function box from the left, control arrows always point down into a function box from the top, output arrows always point away from a function box to the right, and mechanism arrows always point up into a function box from the bottom. From the IDEF0 standard:

- *function*: "A transformation of inputs to outputs, by means of some mechanisms, and subject to certain controls, that is identified by a function name and modeled by a

box.”

- *input*: “In an IDEF0 model, that which is transformed by a function into output.”
- *output*: “In an IDEF0 model, that which is produced by a function.”
- *control*: “In an IDEF0 model, a condition or set of conditions required for a function to produce correct output.”
- *mechanism*: “In an IDEF0 model, the means used by a function to transform input into output.”

We differentiate an input from a control in that an input is comprised of Energy, Matter, Material wealth, or Information (EMMI) and a control is generally a condition such as a law, policy, guidance or other rule that governs/constrains the function. EMMI may be shown as a control when it governs the behavior of the destination function.

The simplest IDEF0 diagram is the A0 diagram, which is a black box -style view of a system’s function and its Inputs, Outputs, Controls and Mechanisms (ICOMs). The diagram at the top of Figure 2.20 is a model for a topmost system function IDEF0-A0. The bottom two diagrams are notional IDEF0-A0 views for first level functions. Input/outputs that are internal to the system are color-coded blue in these figures. For instance, the "internal system instructions" output from Subsystem 1’s function is a control on Subsystem 2’s function, and "object data" output from Subsystem 2’s function is an input to Subsystem 1’s function. Normally, internal interactions are discovered/derived after some iteration, rather than knowing them all at the outset.

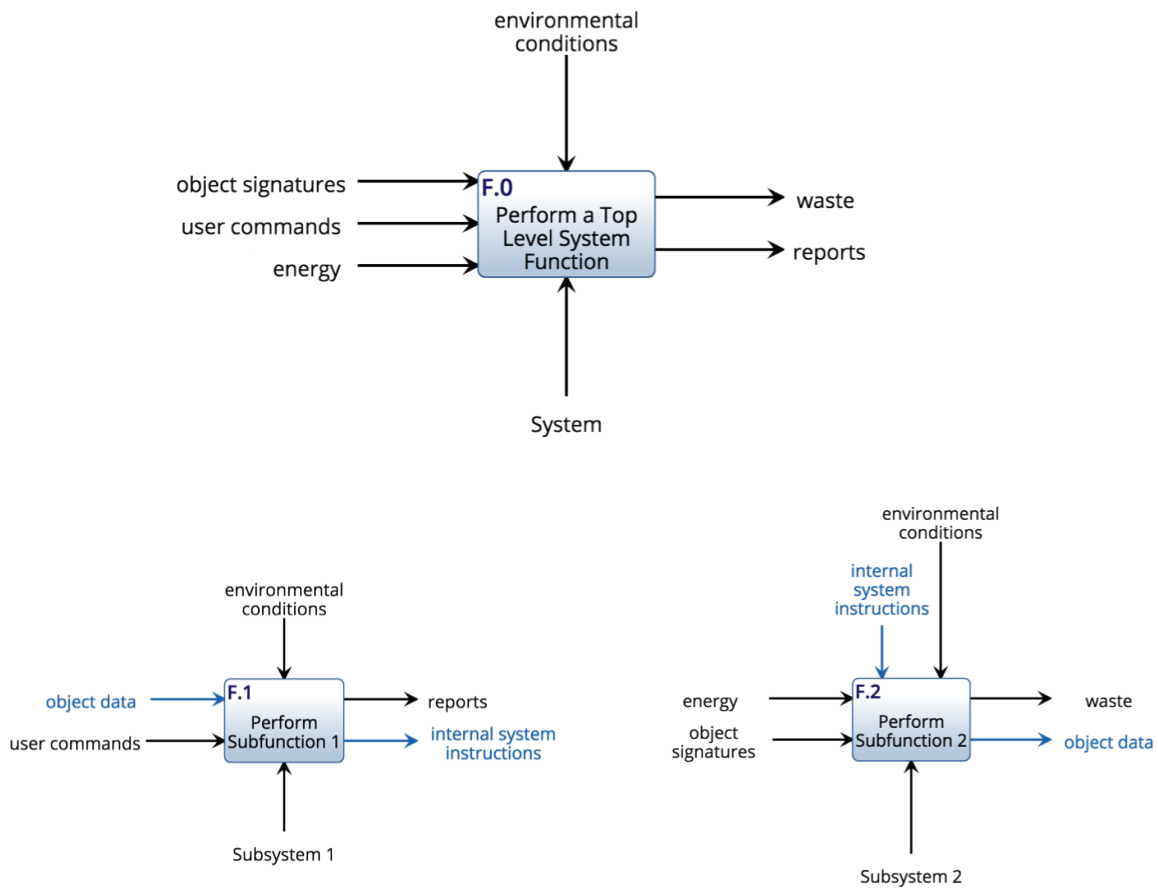


Figure 2.20. Patterns for an IDEF0-A0 diagram. Each action (function) has at least one control and one output (an IDEF0 heuristic from [2]).

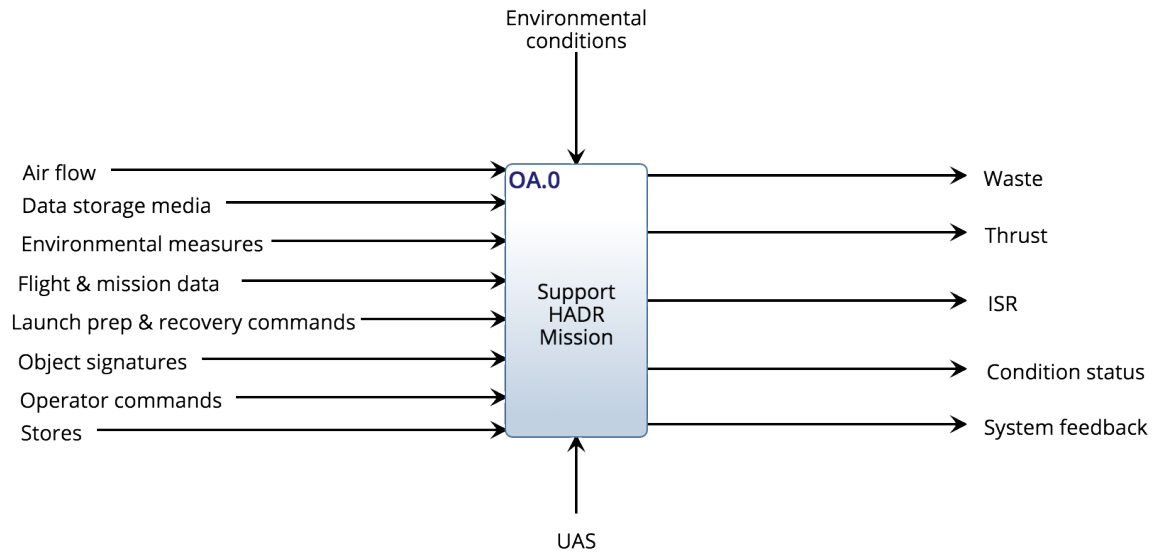


Figure 2.21. Example of an IDEF0-A0 diagram, top level. The action has at least one control and one output (an IDEF0 heuristic from [2]).

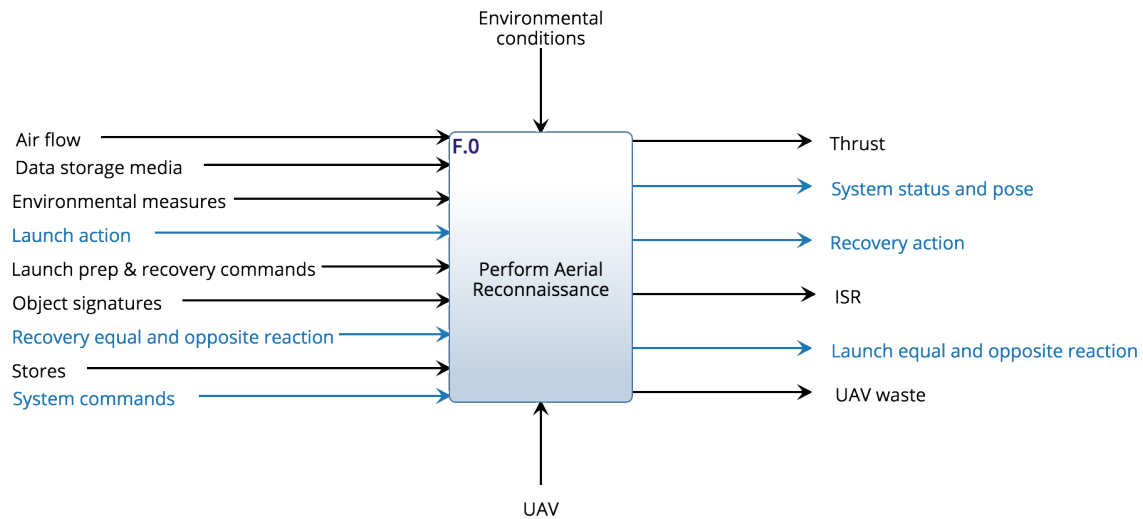


Figure 2.22. Example of an IDEF0-A0 diagram, first level. The action has at least one control and one output (an IDEF0 standard heuristic from [2]).

2.3.8 IDEF0 Diagrams

The functions on the IDEF0 diagrams are abstractions of actions found in the more detailed activity models. The value of the IDEF0 diagrams lies in the way they aggregate many

individual actions sequenced out on activity models into high level functions, and then project a sort of executive summary of all use cases and activity flows.

IDEF0 diagrams place multiple function boxes on one diagram to show function-function interactions. Information from separate IDEF0-A0 diagrams is woven together to show sources and destinations for the arrow heads and tails pointing towards and away from each function box. IDEF0 diagrams can be decomposed many layers down, typically to the second, third, or even fourth level, or until the architecture is modeled sufficiently to begin detailed design work.

As specific usage instances of the system, action diagrams and sequence diagrams can be traced through an IDEF0 at any level, like a single thread weaves through a completed fabric. More abstract than use case and other behavior models, the IDEF0 is intended to describe the overall functionality of the system as an aggregation of the use cases. Input, output, and control names may be more abstract on higher level IDEF0 diagrams, and more detailed on lower level diagrams through the use of input/output decomposition. Each abstract input/output may be used to represent many different instances of more specific interactions among the systems that are shown in the behavior models. If every interaction from the behavior models were to be shown in full detail on one IDEF0, that diagram would be too busy to be useful. Employing abstraction on all ICOMs reduces IDEF0 diagram complexity.

As a general guideline, IDEF0 diagrams should follow the 7 plus or minus 2 rule [20] or better yet, contain between 3-6 functions [19], to aid sense-making of the presented information. Exceeding this guideline risks sending human reviewers into sensory overload, so functional and physical grouping is encouraged so long as it aids rather than hinders comprehension. Aggregation and partitioning can be used to dial each entity of the diagram to an appropriate level of abstraction to achieve this guideline.

The diagram at the top of the IDEF0 hierarchy is a context-level IDEF0 diagram that should show interactions (inputs, controls, outputs) between systems (mechanisms) at the highest level (on par with the high level operational concept graphic). Elements of the operational concept, both internal or external to the design, are shown as the mechanisms on this diagram, with corresponding top-level functions. Figures 2.23 and 2.24 show notional IDEF0 diagrams that are at the context level. They both meet heuristic guidelines, with the exception of the environment's function lacking a control in each diagram. The IDEF0 specification itself calls for each function having at least one control, and one output [2]. The environment's function is a part that is outside the scope of design (the parts of the environment over which we do not have design control). Even if we wanted to show the control on the environment's function, we encounter the problem of a potentially infinitely long or undefined specification (what were the conditions for that, and what were the conditions for that, and so on back to the beginning of time). We therefore need to assume

this particular control is present and doing its thing to enable the environment to do its thing.

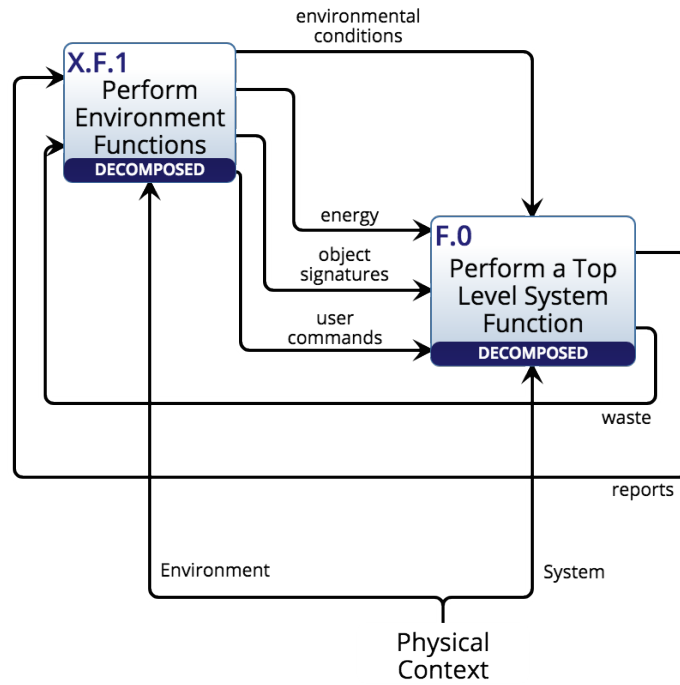


Figure 2.23. Pattern for a context-level IDEF0 diagram. The main system function has at least one control and one output.

ment, and provides a means to gradually dial up the level of detail without overcrowding the high level diagrams. Each diagram should be some abstraction of one beneath it and an elaboration on one above it.

The first decomposition of a system's topmost function may be called a first-level IDEF0 diagram. This is generally how we refer to an IDEF0 diagram that is just beneath the context-level diagram. Depending on the system size and complexity, the first-level IDEF0 may show sub-systems of a system (as in Figure 2.25), or the composite systems of a SoS (as in Figures 2.26 and 2.27).

Figure 2.26 shows a notional breakout of a generic environment, which includes external systems along with the physical environment itself. This alternate approach would simplify the diagram in Figure 2.24 showing the physical environment, external systems, and the SoS under design (the UAS) all on the same diagram, which has benefits and drawbacks. The approach taken is usually the architect's discretion, but it is a good idea to use feedback from those consuming the diagrams to dial the resolution at each level according to what will make it easiest to find flaws in the model. A complicated diagram can hide many errors, but so can an over-partitioned one, if reviewers need the integrity of seeing the pieces they care about in one place.

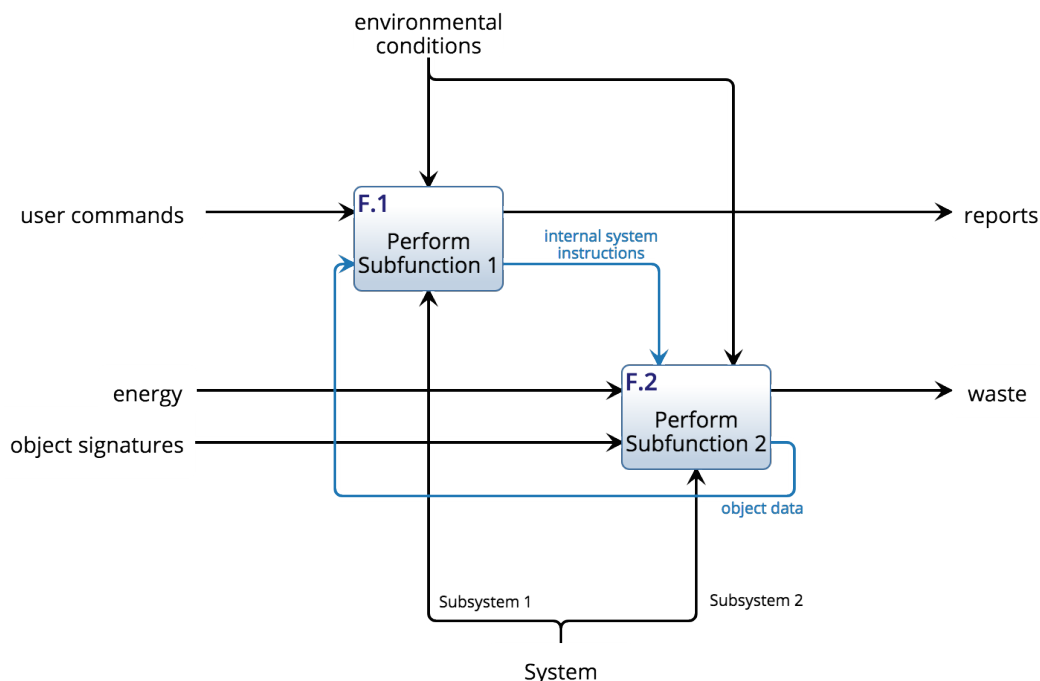


Figure 2.25. Pattern for a first-level IDEF0 diagram. Each function has at least one control and one output.

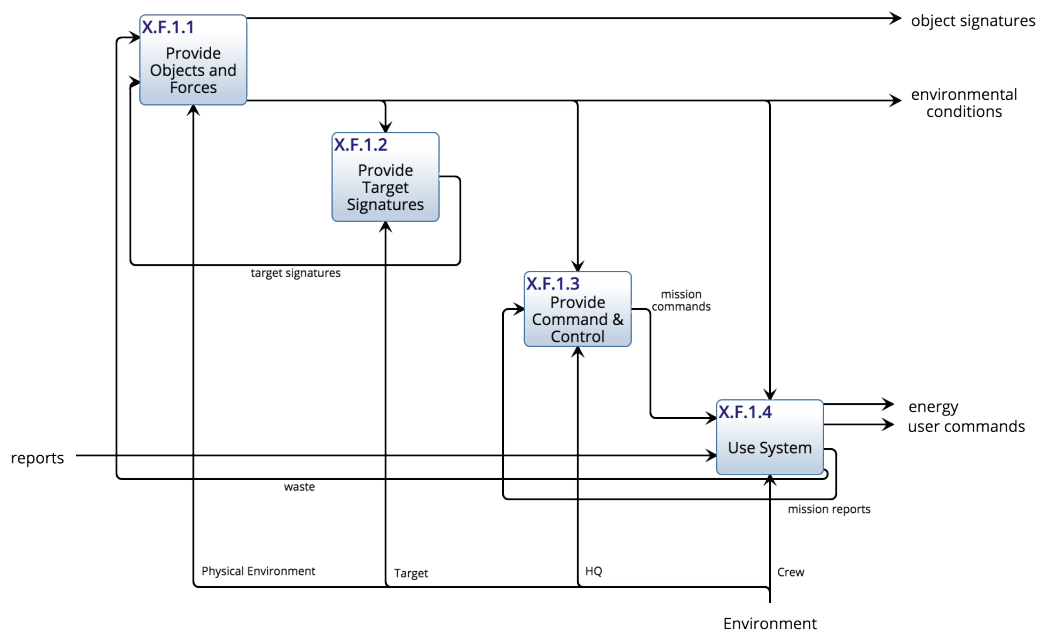


Figure 2.26. Pattern for an IDEF0 decomposing external functions. Each function has at least one control and one output, except for the Physical Environment, whose controls are outside the scope of the model.

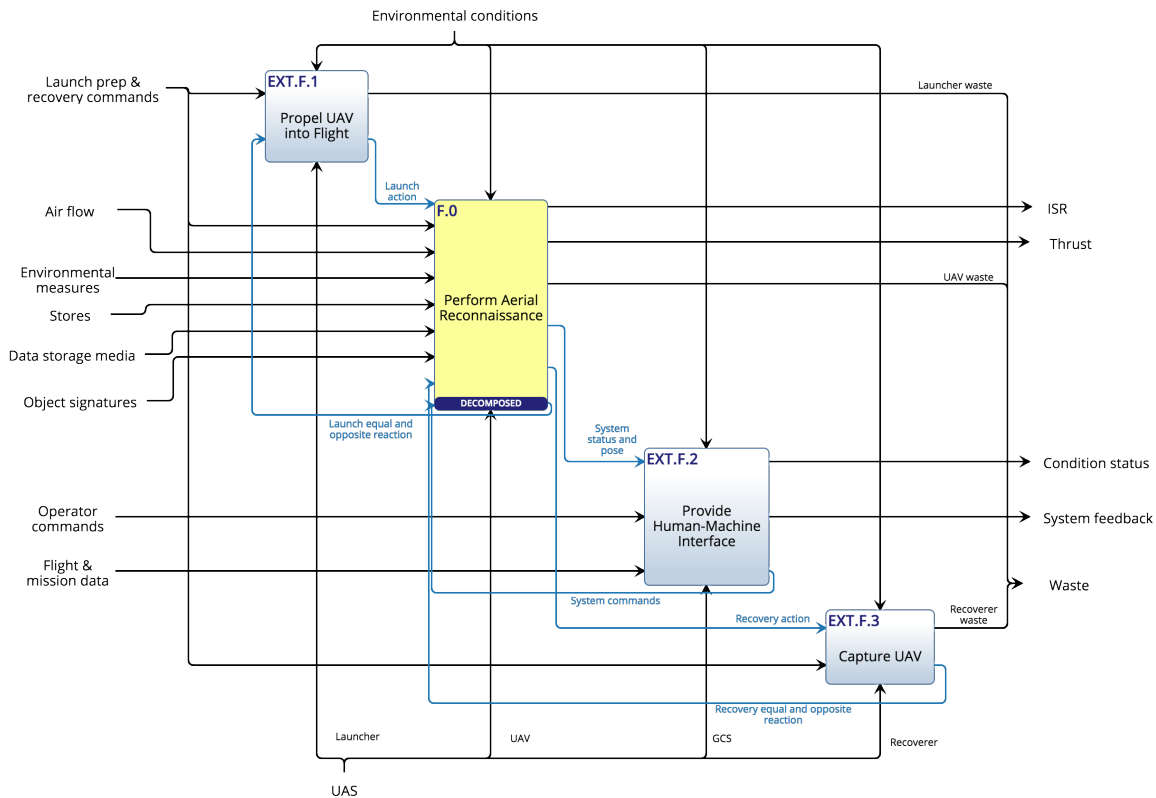


Figure 2.27. Example of an IDEF0 diagram, first level. The input/outputs from the top level (black arrows) are carried down and allocated to first level functions of the UAS, and new internal input/outputs (blue arrows) are created to provide interaction among the first level functions that provide continuity for the top level input/outputs.

First-level diagrams can be decomposed into second-level diagrams, and second-level diagrams into third-level diagrams, and so on. A common question asked is, "When do I stop decomposing?" A general rule of thumb is to stop when your detailed designers have enough to go on and are clear about what it is the system or subsystem is to do in the bigger picture (and not do, but that also comes later through addition of constraints). If one encroaches too far into the detailed specification, the design space can get inadvertently over-constrained. There is solution-neutral or operational architecture, and there is solution-oriented architecture, but systems architecting is *high-level* design work.

The second-level diagram example in Figure 2.28 shows the decomposition of the UAV, which has the first assets named as "subsystems" in this model of an SoS. Again, if the model were simpler, subsystems like these may have appeared at the first level - it is only due to decisions about how to group and nest systems and subsystems in this model that they appear at the second level. This particular model is an example, but it also contains a pattern in

that the subsystems shown are typically found in many systems (power, propulsion, control, structure, communication, and sensing). Even a human being has functions for these, albeit the biological nomenclature for human subsystems is a little different.

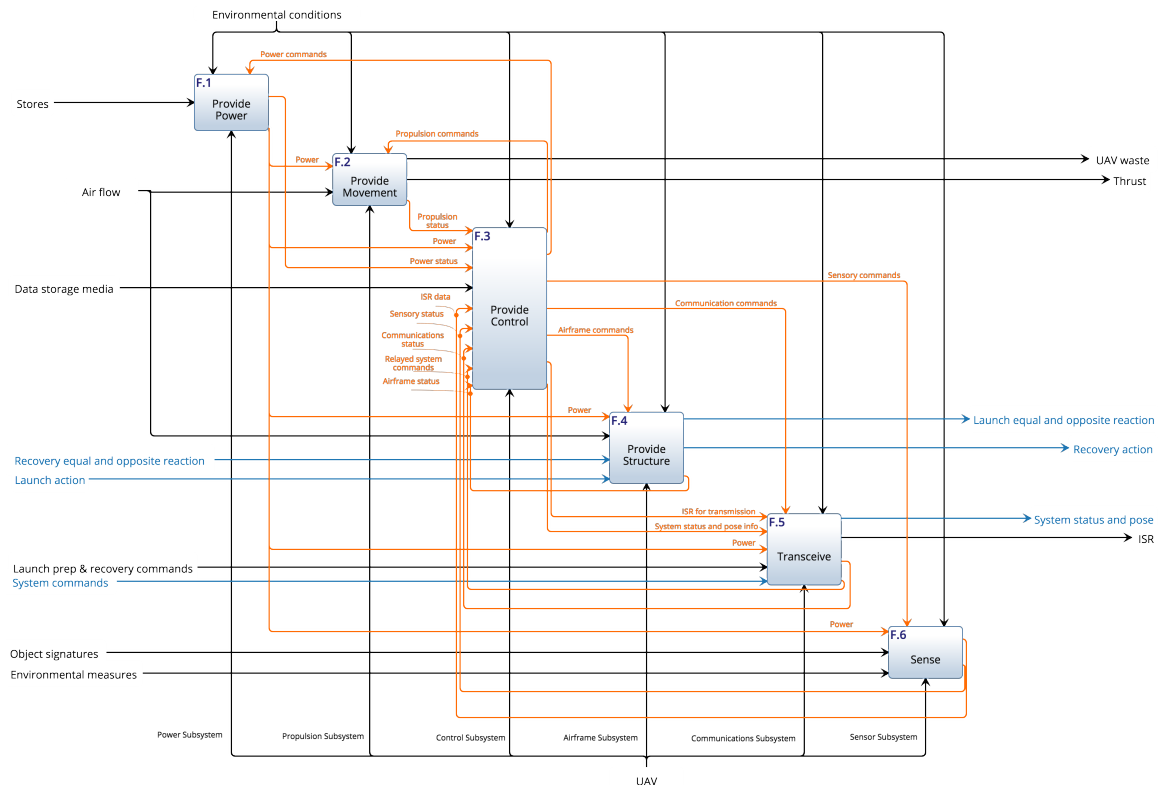


Figure 2.28. Example of an IDEF0 diagram, second level. We can distinguish input/outputs originating from the top level (black arrows), the first level (blue arrows), and new internal input/outputs (orange arrows) created to provide interaction among the second level functions that support first level and top level interactions.)

2.3.9 Asset Diagrams

An asset diagram is a type of physical block diagram that shows assets and their connectivity via conduits. Each conduit transfers one or more input/outputs exchanged between assets (consistent with the IDEF0, for example). Like other diagrams, these diagrams can be step-wise refined to show the connectivity of systems, subsystems and lower level components of the architecture. Figure 2.29 shows a pattern for a top level asset diagram that can be configured to resemble a high level operational concept. When other parts of the model are populated, the connector between the assets transfers the same input/outputs as shown on other diagrams such as an action diagram or an IDEF0 diagram.

The environment and the system both have a blue "Decomposed" bar beneath them, indicating there is an elaborating diagram beneath each. Figure 2.30 shows the system's first-level asset diagram, and 2.31 shows the environment's first-level asset diagram. Both diagrams have overlays that call out the input/outputs transferred by the conduit connecting the assets. All information in the asset diagrams is present in IDEF0 diagrams, but asset diagrams do not address functions as does the IDEF0.

Figure 2.32 shows the example high level operational concept for HADR as a context-level asset diagram with a call-out of input/outputs transferred by the conduit between the UAS and the environment.



Figure 2.29. Pattern for a context-level asset diagram, showing a call-out of input/outputs transferred by one of the conduits.

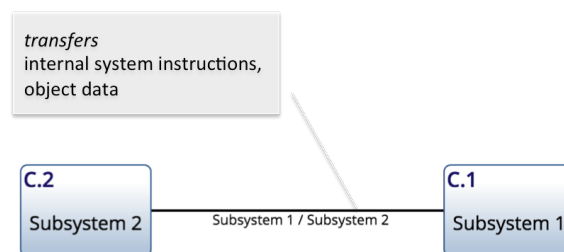


Figure 2.30. Pattern for a first-level asset diagram of the system, showing a call-out of input/outputs transferred by one of the conduits.

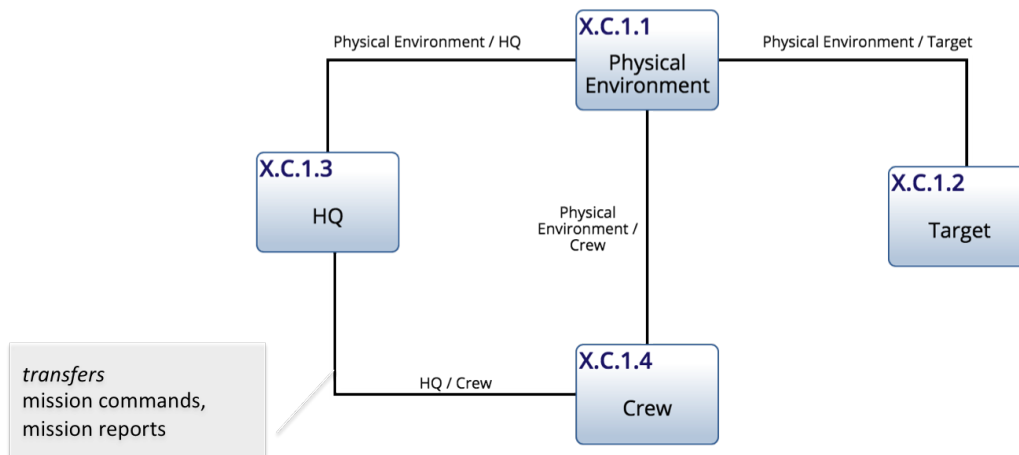


Figure 2.31. Pattern for an asset diagram of assets making up the environment, showing a call-out of input/outputs transferred by one of the conduits.

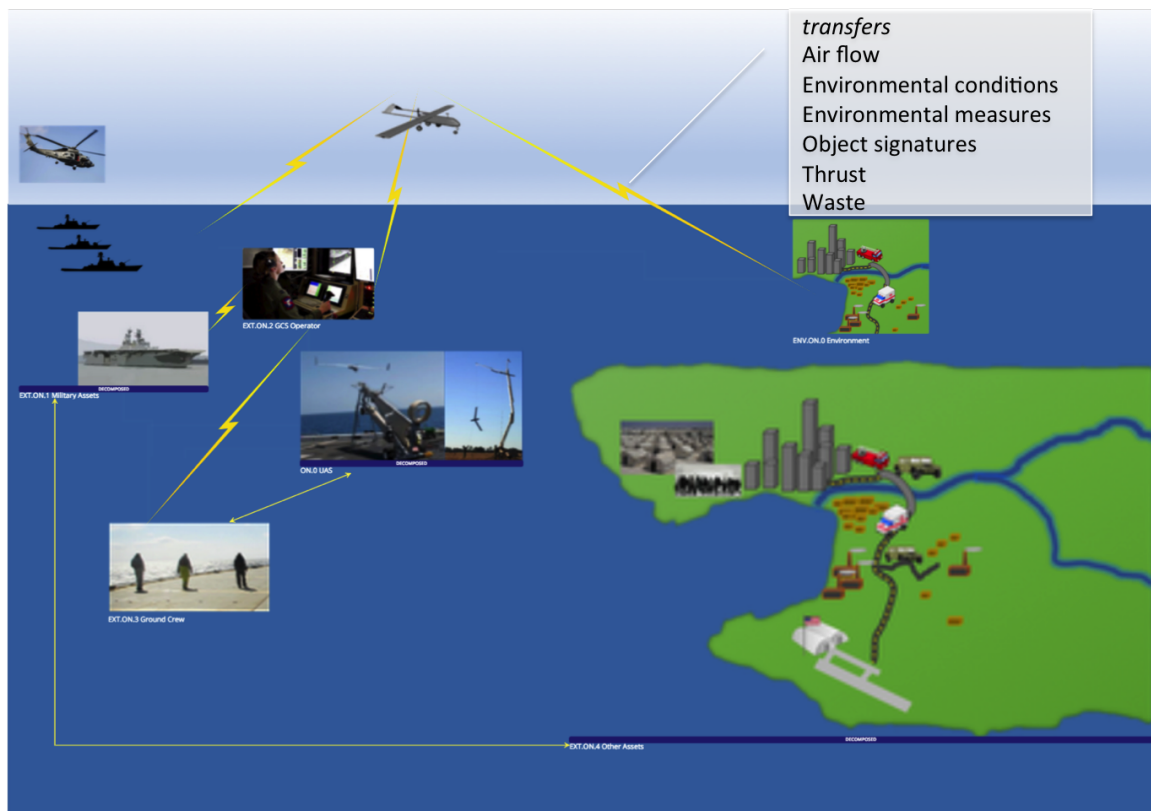


Figure 2.32. Example of a context-level asset diagram, showing a call-out of input/outputs transferred by one of the conduits.

An example first-level asset diagram is shown in Figure 2.33. This asset diagram depicts the connections between the elements of the UAS (UAV, Launcher, Recoverer, and GCS) as well as elements external to the UAS (GCS Operator, Environment, and Ground Crew). Different views can be projected to include other assets at different levels, in effect "flattening" the hierarchy to make more connections visible (in this case, for the communications and airframe subsystems). As assets are tangible things, pictures may be used to make the assets more memorable and custom to the project. The input/outputs transferred by the conduit between the GCS and UAV are shown in a call-out; all conduits transfer input/outputs consistent with the IDEF0 diagram.

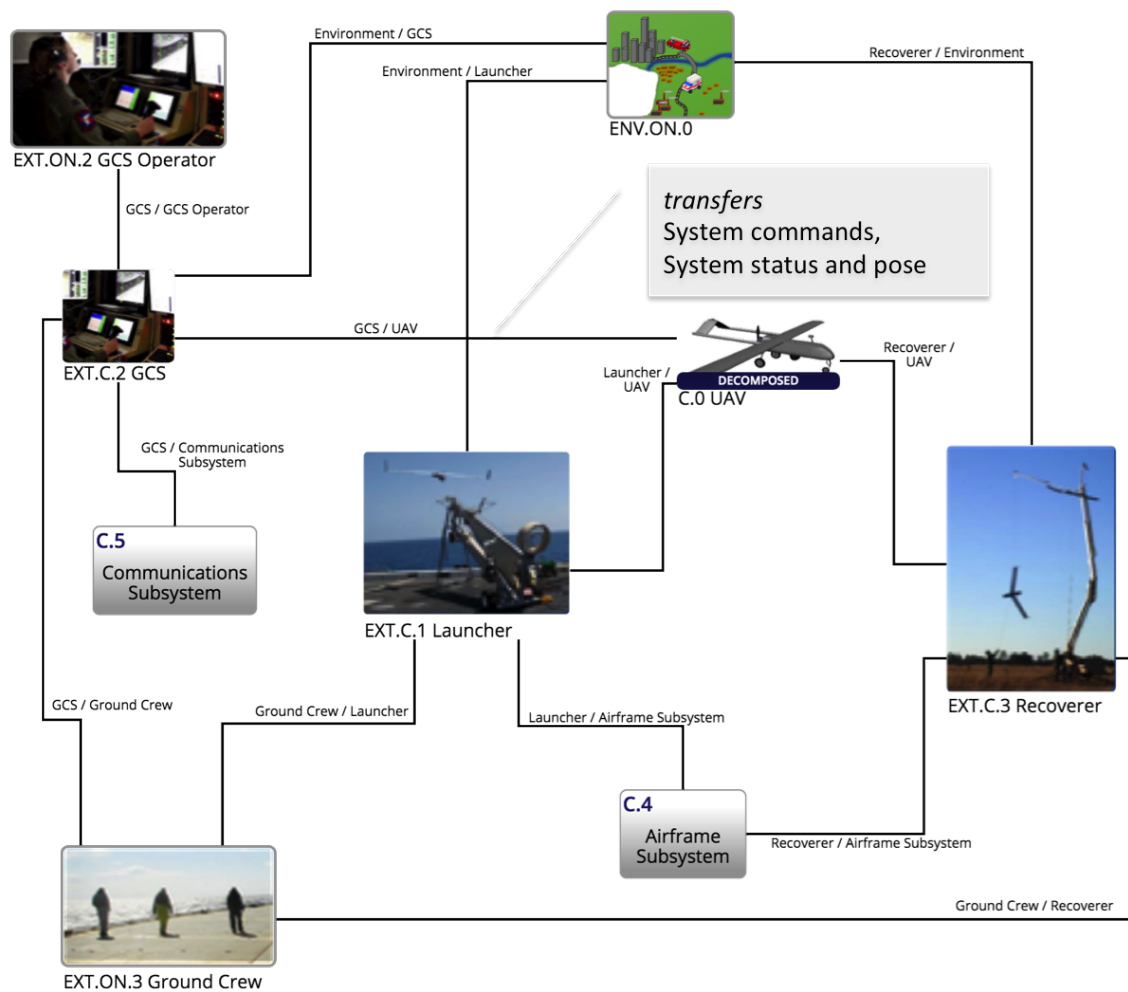


Figure 2.33. Example of a first-level asset diagram of the UAS system, showing a call-out of input/outputs transferred by one of the conduits.

An example second-level asset diagram is shown in Figure 2.34. This asset diagram

depicts the connections between the elements of the UAV (Power Subsystem, Propulsion Subsystem, Control Subsystem, Airframe Subsystem, Communications Subsystem, and Sensor Subsystem) as well as elements external to the UAV (Environment, GCS Operator, Ground Crew, GCS, Launcher and Recoverer). The input/outputs transferred by the conduit between the Ground Crew and Power Subsystem are shown in a call-out; all conduits transfer input/outputs consistent with the IDEF0 diagram.

Asset diagrams are mainly for showing different views of the physical architecture. The diagram expands an asset into its constituent assets, and arranges other connected assets around them.

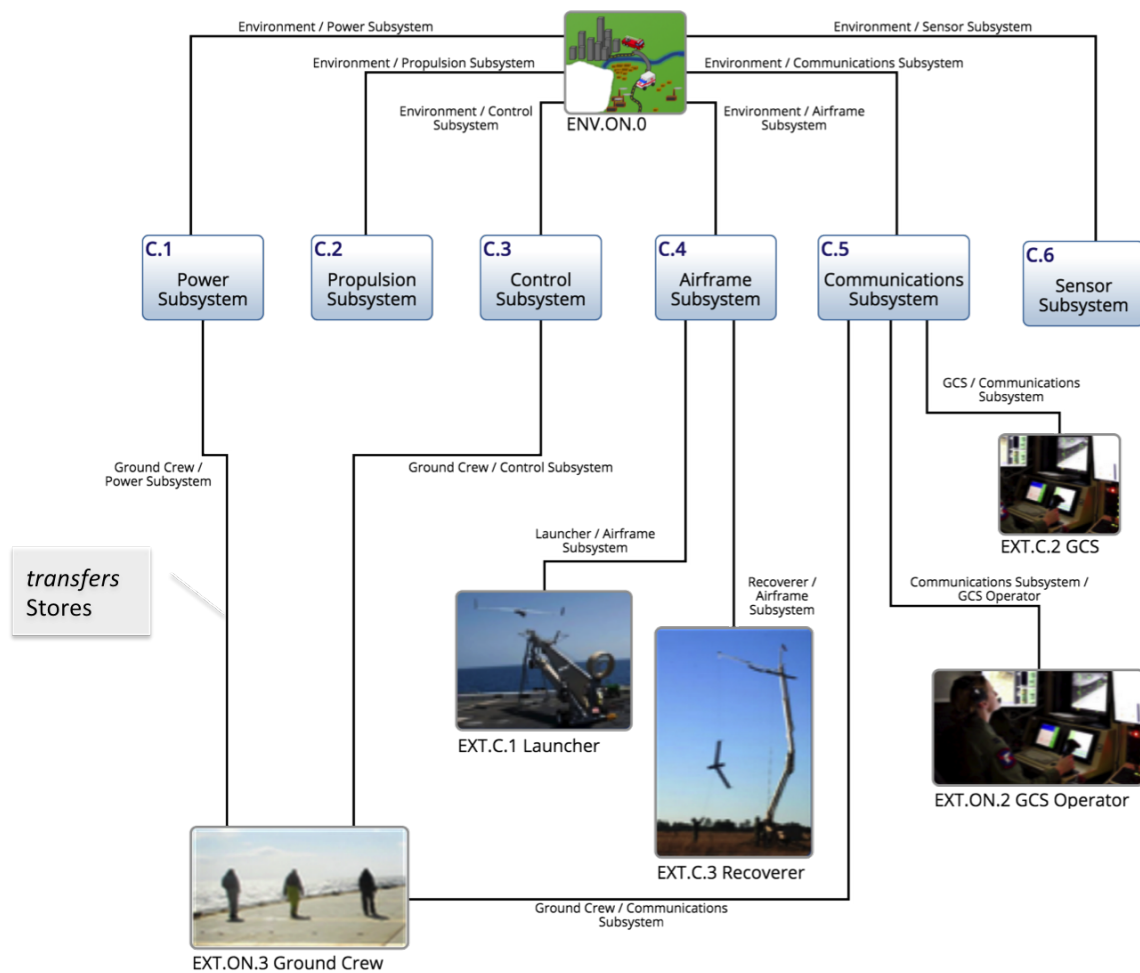


Figure 2.34. Example of a second-level asset diagram of the UAV system, showing a call-out of input/outputs transferred by one of the conduits.

2.4 Chapter Summary

A conceptual data model underlies every system model, and the basic concepts in many CDMs are compatible. A CDM provides a powerful logical language for expressing typical architecture modeling heuristics and analytics in a tool-agnostic manner. The analytics presented herein provide decision makers with the means to express expectations for model quality and maturity, and enable new and experienced architects to purge many known poor modeling practices from their models. Translations of the analytics into SDL, DM2, and UPIA were provided, as were examples of well-formed architecture models that follow the provided heuristics.

CHAPTER 3:

A Tool-Agnostic Methodology for Deriving and Documenting Architecture-Level Patterns

Kristin Giammarco

This chapter begins with a description of the motivation and objectives for deriving architecture model patterns from Conceptual Data Models (CDMs) like those presented in Chapter 2. Next, we present an Alloy model of UPIA classes and relationships that were contained in NAVAIR-provided Generic Model examples, along with a visual of the metamodel. We then describe a methodology based on [3] and [4] for identifying architecture-level patterns by inspecting simulated instances of the Alloy model. This chapter closes with a general description of the methodology for use on any tool's CDM.

This chapter contains a version of work authored by Kristin Giammarco appearing as the paper entitled “A formal method for assessing architecture model and design maturity using domain-independent patterns” in proceedings of the 12th Annual Conference on Systems Engineering Research, Redondo Beach, CA, March 21-22, 2014.

3.1 Introduction

Design patterns have been used as a formal or systematic means for extracting and patterning knowledge about good design choices, as well as capturing lessons learned associated with poor design choices (or so-called anti-patterns). Yet little attention has been devoted specifically to pattern languages that are based on the fabric of architecture models – the conceptual data model – to capture reusable design knowledge as well as good and poor architecting practices that can be applied in more than one domain at a high level of abstraction. We demonstrate a simple model-based method for identifying and patterning architecture design aspects that are domain-independent, and thus transferable and reusable in any system design or tool with a comparable data model. Discovered patterns can be shared in a community, codified in a tool of choice, and sought out in actual architecture models of systems using automation. Since there may not be universal agreement on a common set of “good” and “poor” patterns, individual architects or organizations can use this method to state their particular practices as formalized heuristics, and structure them to assist in the assessment of model and design maturity against their own specific standards.

3.1.1 Motivation

Researchers and practitioners continue to face a longstanding challenge throughout government and industry: to understand and predict architecture-level design flaws earlier, and thus relatively less expensively, compared with fixing them later in the lifecycle [22]. The ability to document and reuse architectural patterns that are desired or constructive (“good”), as well as architectural patterns that are undesired, disruptive or destructive (“poor”), would assist the systems engineering community with the exposure and transfer of valuable information. At the intersection of the software and systems engineering fields, formal methods have been used to begin to codify rules that apply in any system architecture model with a comparable data model.

3.1.2 Objective

The objective of this part of the research is to test the domain- and tool- neutrality of a simple model-based method for identifying and patterning architecture design aspects, by implementing it using a language to which the method has not yet been applied. The language to be tested is the UPIA [23]. Chapter 2 has already demonstrated the neutrality of known architecture heuristics, and the analytics tables show the degree to which these heuristics are transferable and reusable. The method underpinning the derivation of these analytics is the subject of this chapter.

3.2 Related Work

The use of design patterns as a formal or systematic means to extract and manage knowledge about good and poor designs can be found throughout the literature. The languages of these patterns run the gamut from formal to informal/natural. Gamma et. al [24] introduce design patterns, or “reusable micro-architectures that contribute to an overall system architecture,” as a means for abstracting and describing recurring themes that appear in many object oriented designs. Bosch [25] describes various kinds of structural and behavioral design patterns (e.g., Adapter, Bridge, Observer, Mediator) as object-oriented compositions that are free of implementation details and thus reusable in many designs calling for the functions or behaviors described in those patterns. The concept of design patterns has been applied in the systems engineering domain, for example Haskins’ [26] pattern for “Multidisciplinary Teams,” and Cloutier’s [27] pattern for “Command and Control.” Rebovich and DeRosa [28] capture two success patterns in Information Technology (IT)-intensive systems in a government acquisition environment, namely “Balancing the Supply Web” and “Harnessing Technical Complexity.” There has also been work in capturing lessons learned associated with poor design choices, or anti-patterns [29], and patterns for system misuse [30]. Weilkiens [31] points out that graphical layouts for MBSE diagrams can be patterned, such as a tree diagram being presented in a top-down, bottom-up, or non-tree layout. In the system sciences community, 55 candidate systems processes have been identified for

patterning at multiple levels of abstraction [32]. Giammarco [3] [7] [8] and Rodano and Giammarco [6] utilize an architecture data model similar to that presented in the nomenclature as a pattern language, and develop domain-independent patterns from that language. This work in pattern finding pertains to formally specifying and transferring knowledge about design practices and configurations that permeate architecture models of any system described using the UPIA or a compatible data model such as the DoDAF Meta Model (DM2), Lifecycle Modeling Language (LML), or System Description Language (SDL).

3.3 Method

The method for formally documenting good and poor architecture-level design patterns is described and demonstrated, along with some example patterns that were identified while using the method. The research methodology that ultimately resulted in this formal method is documented in more detail in the author's doctoral research [3]. This method is a generalization and abbreviation of a formal method tailored specifically for interoperability assessment of a modeled architecture, also created as part of the author's doctoral research.

3.3.1 Identify the problem

The analysis question of interest must first be identified to bound the scope of the effort. That is, what is it that the architect wants to know about the model? For the purposes of this demonstration, assume that the architect asks the following questions:

1. Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?
2. Are there any patterns present that may create issues for the interoperability of the design described by the model?

Note the generality of both questions in the sense that these questions can be asked of any system under design, i.e., they are domain-independent questions. Also note that the questions are framed to expose problems or deficiencies in the model for correction. This is consistent with the architect's intent to correct known deficiencies early in design. The first question focuses on determining how well-formed [6] [3] the model is, while the latter identifies a specific area of concern for the design described by the model (i.e., interoperability [3]). To answer these questions, the architect must define specifically what conditions would cause a deficiency in the well-formedness of the model, and what conditions would cause a deficiency in the interoperability of the design, respectively.

3.3.2 Model the CDM

An Alloy model of the Generic Model is used to enable the CDM of the Generic Model to be tested and simulated, without needing any particular architecture model example application. The Alloy model enables many desired architecture model constraints to be specified at a high level of abstraction, for application in any architecture model that adheres to the Generic Model. The idea to constrain a conceptual data model with certain model implementation expectations is described in [3], in which conditions (or constraints) for interoperability of a design that could be expressed in the language of any CDM were sought and found. Translation of the Generic Model into Alloy enabled a mapping of these interoperability conditions as well as other modeling best practices onto the Generic Model.

To keep the patterns domain-independent, the heuristics identified by the architect are expressed in terms of architecture model classes, such as those found in the UPIA, and relationships among those classes (i.e., how elements of the design are related). Figure 3.1(a) depicts a directed graph of classes and relationships from the Generic Model specification using UPIA, and Figure 3.1(b) shows the corresponding Alloy [33] code used to generate this metamodel. Only three classes are included in the first iteration of this demonstration: Capability Role, Operational Task, and Information Element. The specific relationships chosen for inclusion on the first iteration are highly relevant to determining interoperability of the design, which is a goal of the second question above.

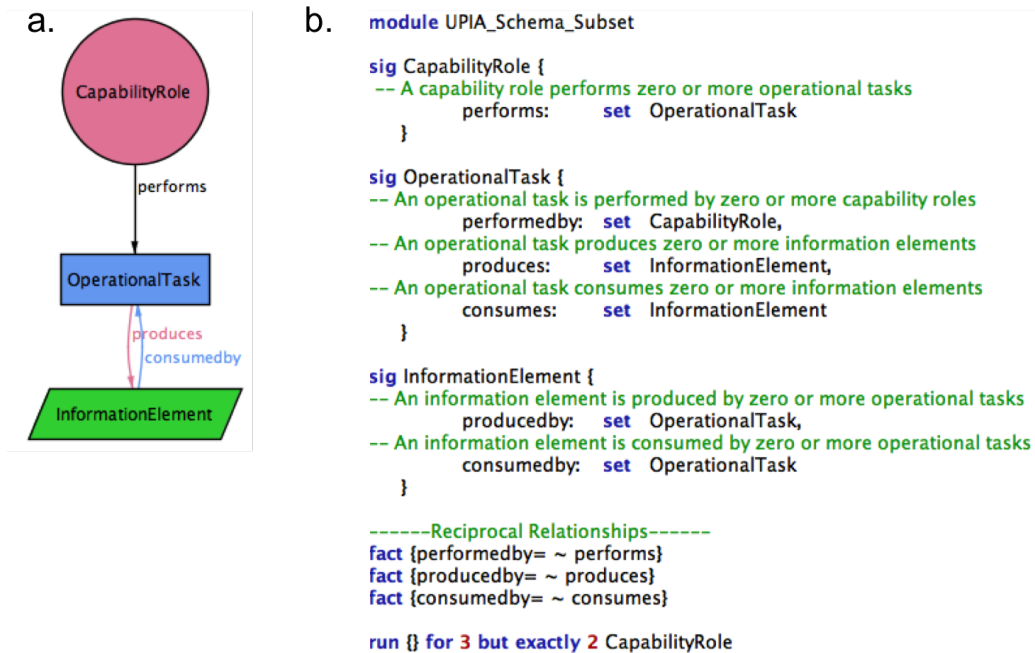


Figure 3.1. (a) Architecture metamodel graph; (b) Architecture metamodel Alloy code.

The Alloy language is a high level language that enables abstract modeling of software and system architecture structures [33]. As can be seen from the annotations, the code straightforwardly formalizes the data model subset to be examined for patterns. In this initial model, no constraints are present to restrict multiplicity (how many relationships are allowed between a given pair of entities). One runs the model, then inspects example instances of implementation of the Generic Model. The Satisfiability (SAT) solver takes every possible combination of class and relationship as specified, and generates example instances up to a specified scope (number of classes, in this use case of Alloy). The results are inspected for applications of the Generic Model that "just don't make sense," which are then formally specified as local or global constraints. The "run" keyword at the bottom of the model sets the scope to 3 but constrains the Vision class to only one instance (which may be an unnecessary constraint in the end, but using a local constraint on the run command helps to reduce the number of possible instances to make inspections a little easier).

3.3.3 Add constraints that capture best modeling practices

The Alloy code in Figure 3.1(b) is executable in the Alloy Analyzer tool [34]. The metamodel in Figure 3.1(a) was generated automatically from the code in Figure 3.1(b). The ability to execute this code is the basis of this formal method. Upon execution, the Alloy code generates many possible instances, showing various implementations of the entities and relationships as they might appear in architecture models using this schema. Thus, this approach provides the architect with an abstract way to reason about the schema of any architecture model without referring to specific architecture models.

The code in Figure 3.1(b) permits the instance of data model relationships shown in Figure 3.2. The most prominent model form issue with this example instance is that Operational-Task0 appears to be looping the same three information elements. While it is conceivable that an operational task may produce and/or consume three information elements, the architect responsible for this design employs a style guide that discourages instances of "looping" information elements, assuming that operational tasks transform inputs into different outputs. The rationale: if an information element crosses a functional boundary, as is the case when an information element leaves the operational task, it does not make sense to wrap it back around into precisely the same operational task that produced it. It is a better practice to show such loops at a lower level of decomposition between child operational tasks for a stepwise description of the feedback transformation and use.

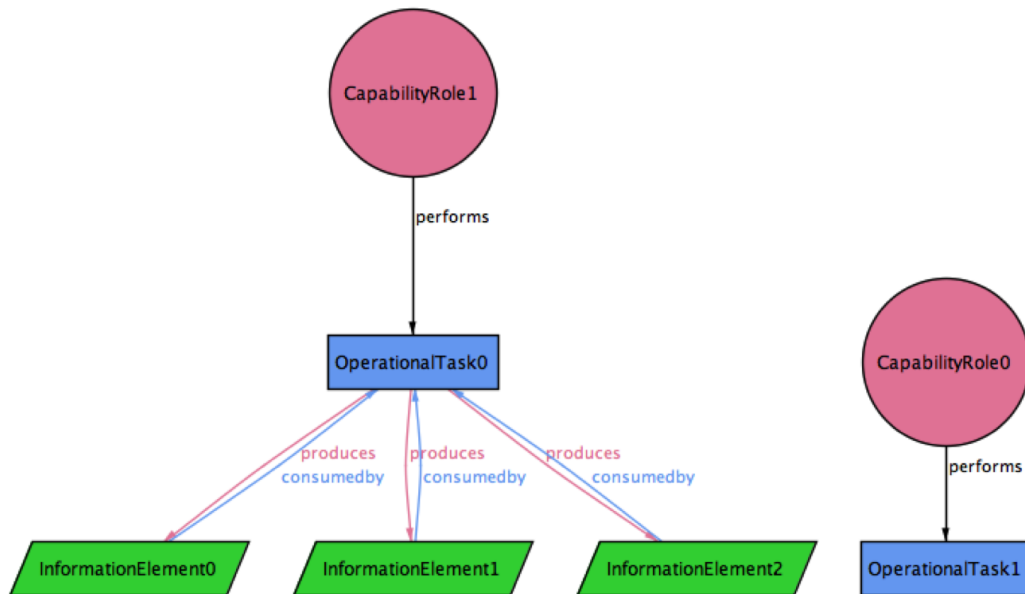


Figure 3.2. OperationalTask0 has three looping information elements.

To forbid these looping information elements, the architect inserts the following constraint into the Alloy model beneath the OperationalTask signature:

```
{ no (produces & consumes) }
```

When the model is executed again with this new constraint, the instance in Figure 3.3 presents. The model form issue spotted in this scenario is that InformationElement2 is consumed, but not produced, by an operational task. This could mean that specification of the source for this information element was overlooked, or assumed. Either way, it is a good practice to specify the source of an input.

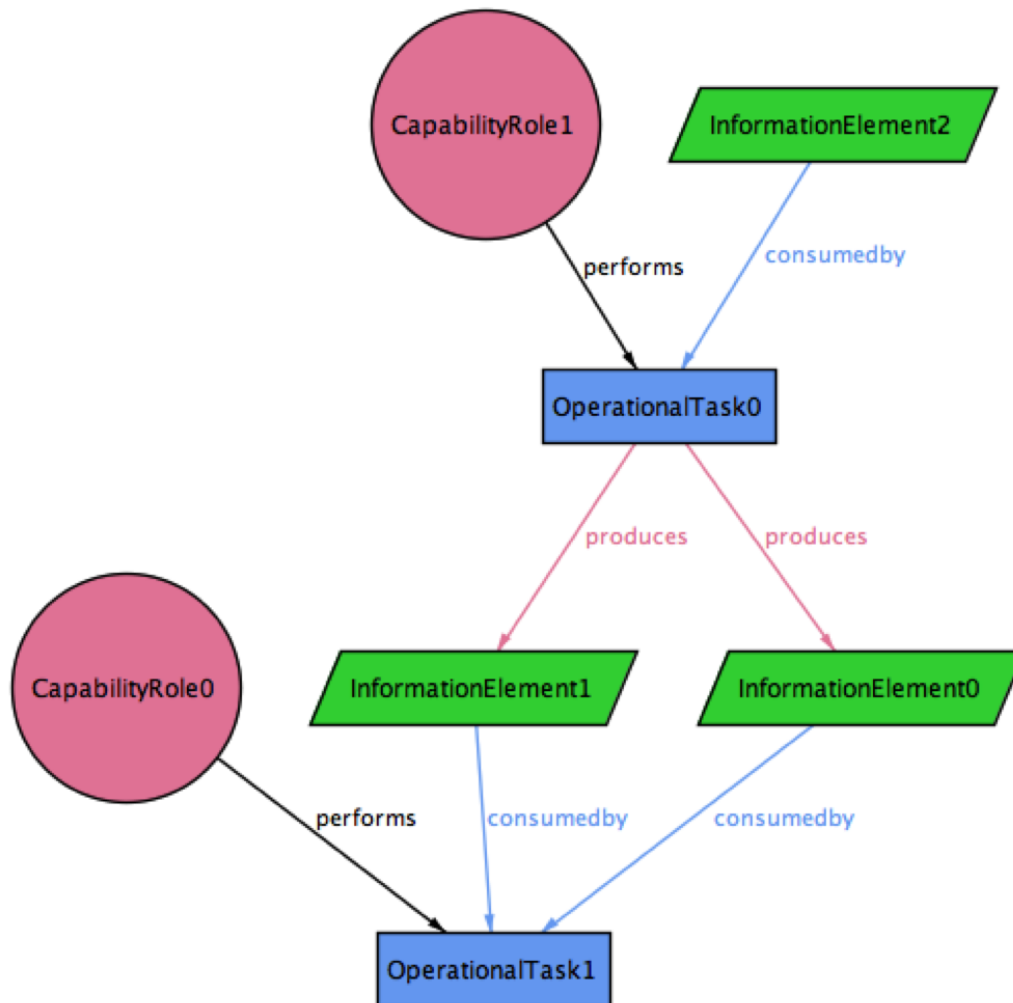


Figure 3.3. InformationElement2 does not have a source operational task specified.

This CDM scenario is remedied by changing `set` to `some` in the `producedby` relation of the `InformationElement` signature as follows:

```

producedby: set    OperationalTask,
to
producedby: some   OperationalTask,

```

This means that an information element is always produced by some operational task.

Running the SAT solver once more, we find the next undesired CDM scenario. In this instance, depicted in Figure 3.4, we have an information element being produced, but not

consumed by any operational task.

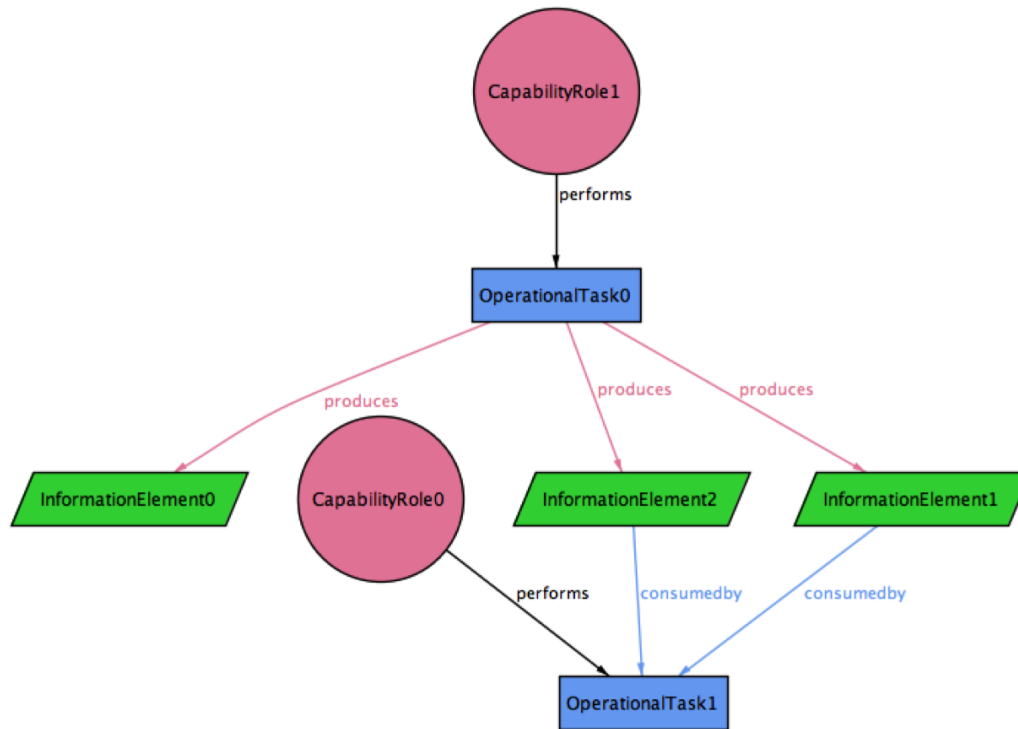


Figure 3.4. InformationElement0 does not have a destination operational task specified.

We change `set` to `some` in the `consumedby` relation of the `InformationElement` signature as follows:

```
consumedby: set    OperationalTask,
```

to

```
consumedby: some    OperationalTask,
```

This means that an information element is always consumed by some operational task.

Running the model again, we encounter a scenario where one capability role is performing two interacting tasks, and a second capability role does not perform any operational tasks.

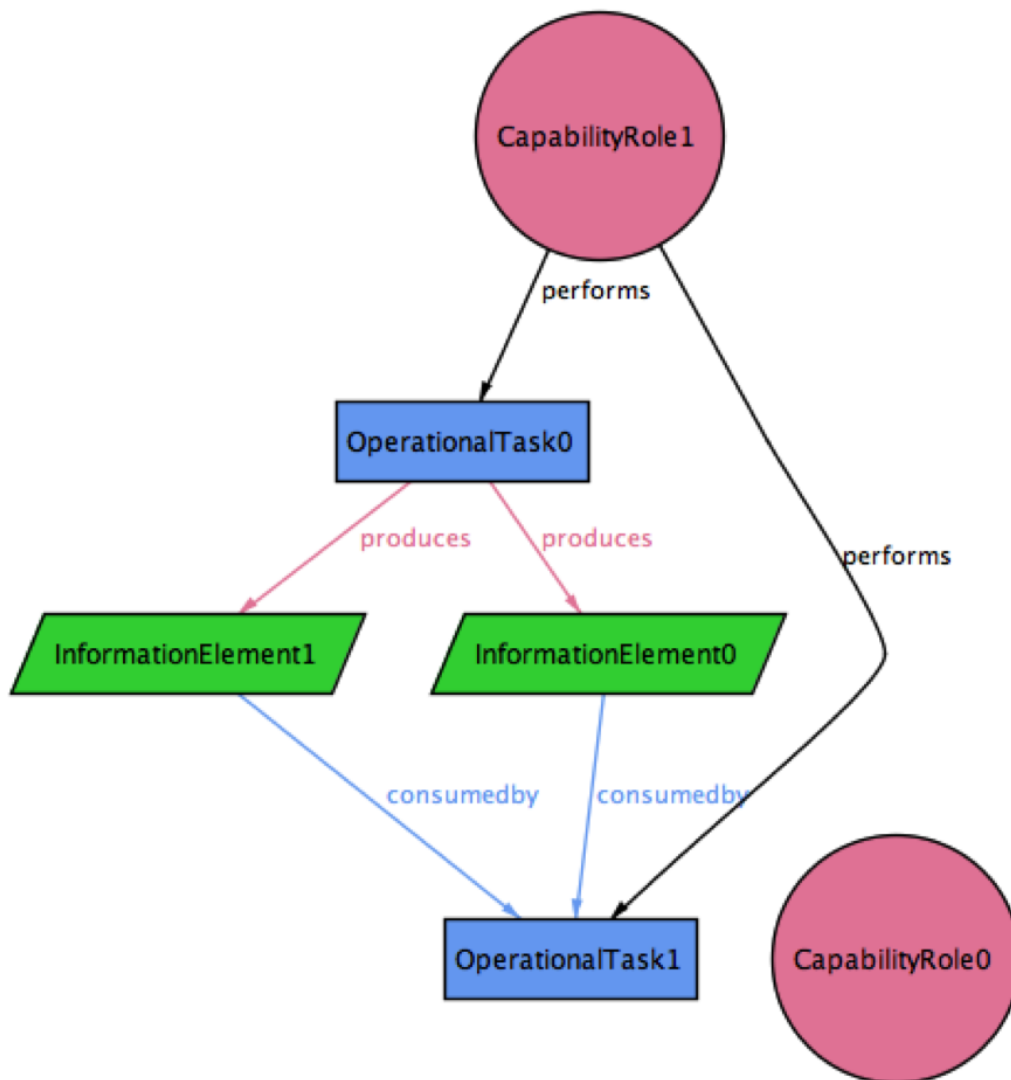


Figure 3.5. CapabilityRole0 does not have an operational task specified.

The issue of the idle capability role is addressed by changing the `set` to `some` in the `performs` relation of the `CapabilityRole` signature as follows:

`performs: set OperationalTask`

to

`performs: some OperationalTask`

This requires a capability role to always perform some operational task.

Having incorporated this constraint, the next interesting scenario is depicted in Figure 3.6. In this case, neither operational tasks are producing or consuming any information elements. Such a case may be valid for a snapshot in time, but for a design specification intended to capture interoperability requirements, such a case raises the question of whether these operational tasks are ever intended to interact with other operational tasks.

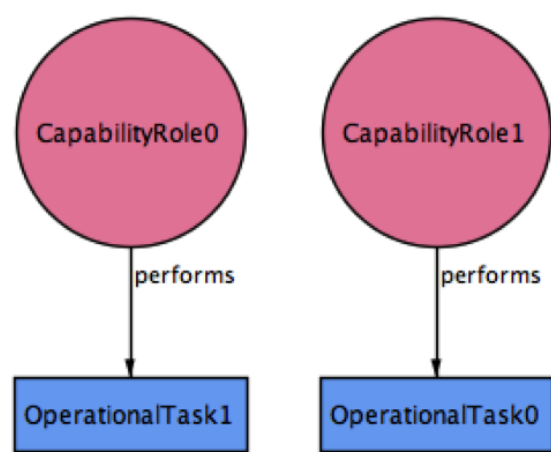


Figure 3.6. Operational tasks that do not interact through exchange of information elements.

The following constraint is then added inside the braces above the first constraint:

`some (produces + consumes)`

which requires each operational task to produce or consume at least one information element.

Continuing the search for anti-patterns, the simulations are run again and the scenario in Figure 3.7 presents. It reveals another undesired pattern, despite adhering to all constraints identified thus far. An operational task is not performed by any capability role.

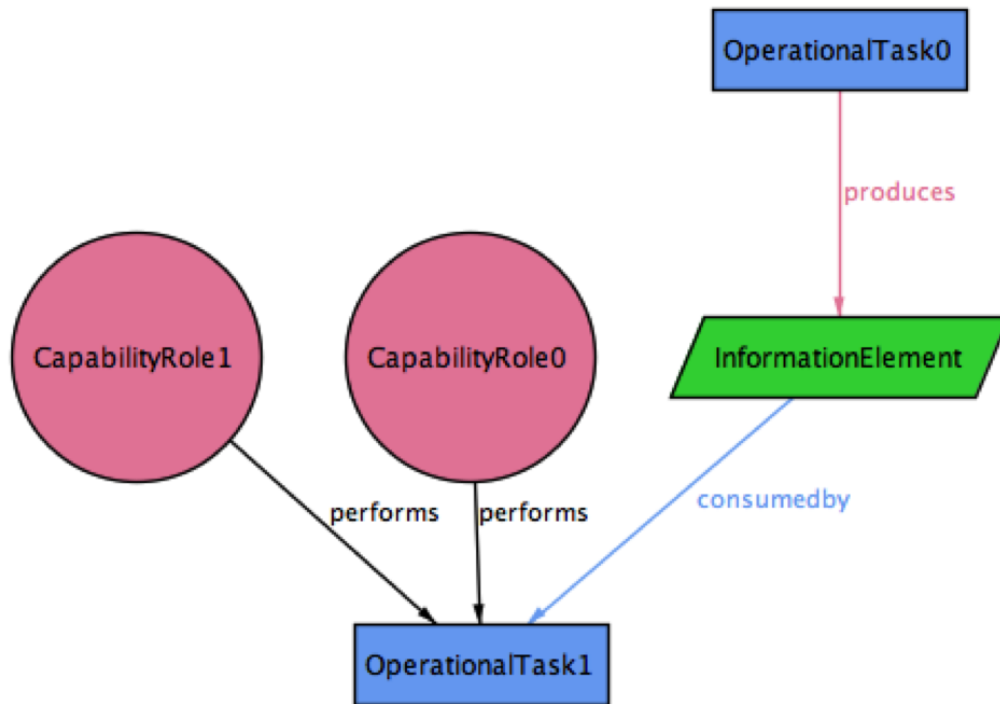


Figure 3.7. An operational task that is not performed by any capability role.

All operational tasks must be allocated to a performing entity, so the following line in the `OperationalTask` signature is changed in the model:

```

performedby: set    CapabilityRole,
to
performedby: some   CapabilityRole,

```

This requires an operational task to always be performed by some capability role.

Once this constraint is applied, inspection of the remaining instances containing pairs of capability roles at a scope of 3 revealed no more critical issues in CDM pattern application (the first question under Identify the Problem, which was "Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?"). Some minor patterns are still identifiable, such as information elements being produced from operational tasks that did not consume anything, and capability roles performing the same operational task, which may be questionable in some cases. For the purposes of this description we will stop with the constraints identified, since they are sufficient in number to demonstrate the method. The Alloy code containing the modifications described above is shown in Figure 3.8.

```

module UPIA_Schema_Subset

sig CapabilityRole {
  --C4: Each capability role performs some operational task
    performs: some OperationalTask
}

sig OperationalTask {
  -- C6: Each operational task is performed by some capability role
    performedby: some CapabilityRole,
  -- An operational task produces zero or more information elements
    produces: set InformationElement,
  -- An operational task consumes zero or more information elements
    consumes: set InformationElement
}

[ -- C5: Each operational task produces or consumes at least one information element
  some (produces + consumes)
  -- C1: No operational task both produces and consumes the same information element
  no (produces & consumes)
]

sig InformationElement {
  --C2: Each information element is produced by one or more operational tasks
    producedby: some OperationalTask,

  --C3: Each information element is consumed by one or more operational tasks
    consumedby: some OperationalTask
}

--run {} for 3 but exactly 2 CapabilityRole

-----Reciprocal Relationships-----
fact {performedby= ~ performs}
fact {producedby= ~ produces}
fact {consumedby= ~ consumes}

assert Interoperability
-- Each capability role produces an information element to or consumes an information element
-- from at least one other disjoint capability role
[ all pA: CapabilityRole | some pB: CapabilityRole - pA |
  some (pA.performs.consumes & pB.performs.produces +
        pB.performs.consumes & pA.performs.produces) ]

check Interoperability for 4 but exactly 2 CapabilityRole

```

Figure 3.8. Modified Alloy code containing constraints.

Since there were many scenarios generated, exhaustive manual inspection was not practical to answer the second question under Identify the Problem, which was "Are there any patterns present that may create issues for the interoperability of the design described by the model?" All scenarios displayed contained capability roles performing operational tasks exchanging information elements, the basis for the definition of interoperability. However, we were curious about whether any scenarios existed that showed conclusively *non-interoperable* behavior between two different capability roles. Since manual inspection

would be prohibitively time-consuming, we used Alloy’s built-in an assertion checking tool to test the model for the presence of the particular scenarios of interest. In our case, we wrote the assertion shown at the bottom of Figure 3.8 to test for the presence of scenarios containing capability roles that do not exchange information elements with at least one other capability role. Running the assertion for pairs of capability roles at scope 4, we found the instance in Figure 3.9, which clearly shows compliance with all of our identified model well-formedness constraints, but the two capability roles are not interacting with one another - which is an interoperability issue if the two capability roles are expected to interoperate.

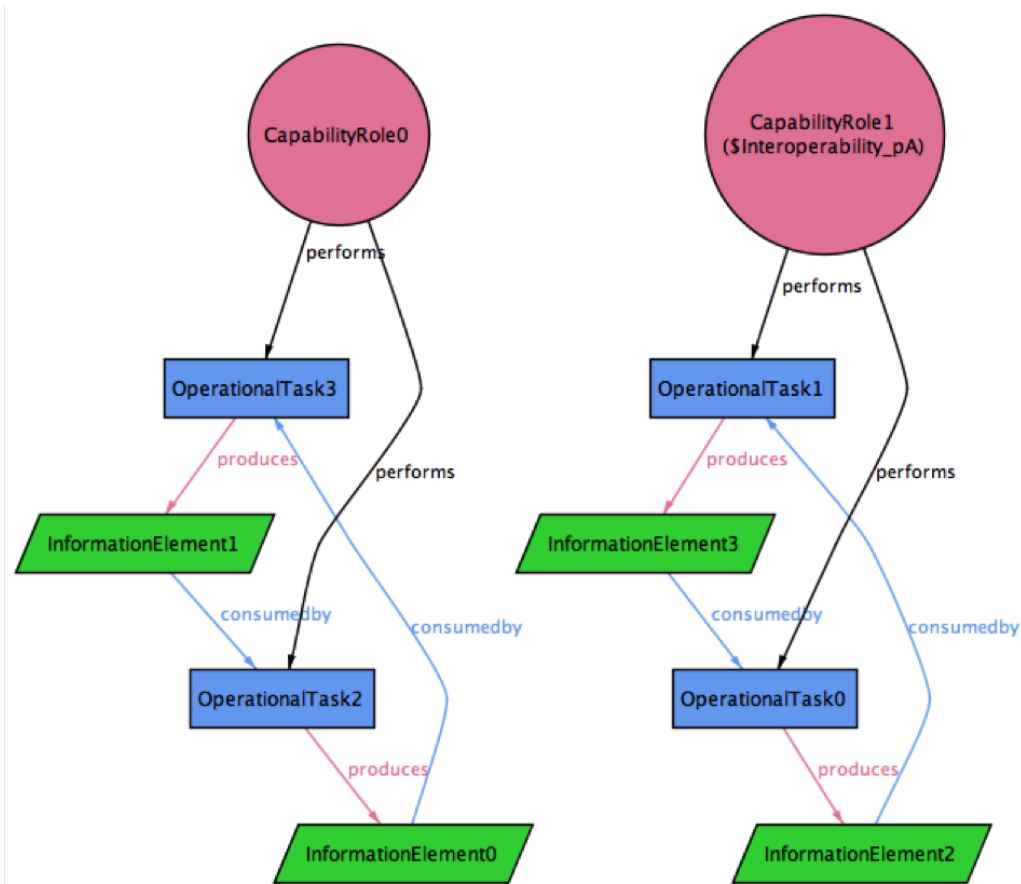


Figure 3.9. Two capability roles are not interoperable with one another.

Reference [3] formally defines *Interoperability* as follows: “the ability of a performer to exchange resources with one or more other performers and to use those resources to accomplish its performed activities according to expected criteria.” This definition of interoperability synthesizes key concepts embodied by most existing definitions [35] [36] [37] [38] [39] [40] [41], with one major difference: it shifts the scope from describing the ability of performers

in general to specifying the ability of a singular performer in the context of other performers with which it may be expected to interoperate. This critical difference treats each performer in the architecture model as a discrete entity, so that interoperability of each performer can be separately and deterministically evaluated based on entities, relationships, and even attributes (for timing and performance factors) in the model.

This definition of interoperability can be used to evaluate the design in Figure 3.9, which shows two performers (called capability roles in UPIA) that do not exchange resources (called information elements in UPIA) with any other performer in the scope of the diagram. To address this concern, a constraint is needed to exclude capability roles that do not exchange information elements with any other capability role from the scope of what is considered interoperable according to the definition above. Note that certain capability roles may be deliberately designed to be standalone performers that do not interact with other capability roles, and these may be systems composed of constituent subsystems that are internally interoperable with each other. The constituent capability roles may still be considered interoperable, even if the top level capability role is not interoperable with external capability roles. The content of the assertion is added as a global constraint to the model to require every capability role in the scope of the model to interoperate with at least one other (different) capability role.

At this model scope, the following analytics can be used to answer the first question: "Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?"

Table 3.1. First iteration heuristics and corresponding analytics supporting Question 1.

No.	Heuristic	Corresponding Analytic
C1	No operational task both produces and consumes the same information element	FI.4.1 Operational tasks that produce and consume the same information element
C2	Each information element is produced by one or more operational tasks	FI.5.1 Information elements not produced by any operational task
C3	Each information element is consumed by one or more operational tasks	FI.6.1 Information elements not consumed by any operational task
C4	Each capability role performs some operational task	FPA.2.1 Capability roles that do not perform any operational task
C5	Each operational task produces or consumes at least one information element	FI.3.1 Operational tasks that do not produce or consume any information elements

Continuation of Table 3.1		
C6	Each operational task is performed by some capability role	FPA.1.1 Operational tasks that are not performed by any capability role
C7	All disjoint capability roles perform disjoint operational tasks (simplifying constraint)	-
End of Table		

At this model scope, the following additional analytic can be used along with the above constraints to answer the second question: "Are there any patterns present that may create issues for the interoperability of the design described by the model?"

Table 3.2. First iteration heuristics and corresponding analytics supporting Question 2.

No.	Heuristic	Corresponding Analytic
	Each capability role produces an information element to or consumes an information element from at least one other disjoint capability role (codified as "Interoperability" fact)	PI.8.1 Capability Roles that do not interact with any other Capability Roles through exchange of Information Elements
End of Table		

3.3.4 Increase the scope of the CDM and repeat

The number of classes and relationships worked with in the previous section are too small in number to be considered sufficient for capturing all interoperability heuristics. Even if we are satisfied with the interoperability heuristics that have been identified, we cannot know for sure that there are not more heuristics that have yet to be discovered. This method supports analysis on incrementally larger data models so that interoperability heuristics can be cataloged in a systematic way.

To demonstrate the extensibility of this method, this subsection demonstrates how it can be iterated on incrementally larger schemas. Figure 3.10 shows a slightly extended version of the metamodel depicted in Figure 3.1(a).

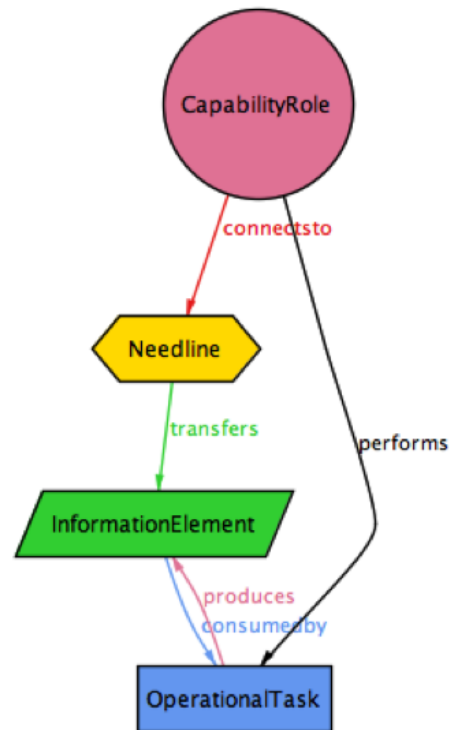


Figure 3.10. Architecture metamodel graph, extended to include needlines.

Figure 3.11 shows the corresponding Alloy code, the extensions for the needline class being incorporated into the constrained model from the previous iteration.

```

module UPIA_Schema_Subset_Extension1

sig CapabilityRole {
  --C4: Each capability role performs some operational task
    performs:    some    OperationalTask,
  --A capability role connects to zero or more needlines
    connectsto:  set    Needline
}

sig OperationalTask {
  -- C6: Each operational task is performed by some capability role
    performedby: some    CapabilityRole,
  -- An operational task produces zero or more information elements
    produces:    set    InformationElement,
  -- An operational task consumes zero or more information elements
    consumes:    set    InformationElement
}

{ -- C5: Each operational task produces or consumes at least one information element
  some (produces + consumes)
  -- C1: No operational task both produces and consumes the same information element
  no (produces & consumes)
}

sig InformationElement {
  --C2: Each information element is produced by one or more operational tasks
    producedby: some    OperationalTask,
  --C3: Each information element is consumed by one or more operational tasks
    consumedby: some    OperationalTask,
  -- An information element is transferred by zero or more needlines
    transferredby: set    Needline
}

sig Needline {
  --A needline connects zero or more capability roles
    connects:    set    CapabilityRole,
  --A needline transfers zero or more information elements
    transfers:    set    InformationElement
}

-----Global Constraints-----
fact Interoperability
  -- Each capability role produces an information element to or consumes an information element
  -- from at least one other disjoint capability role
  { all pA: CapabilityRole | some pB: CapabilityRole - pA |
    some (pA.performs.consumes & pB.performs.produces +
          pB.performs.consumes & pA.performs.produces) }

-----Reciprocal Relationships-----
fact {performedby= ~ performs}
fact {producedby= ~ produces}
fact {consumedby= ~ consumes}
fact {connects= ~ connectsto}
fact {transferredby= ~ transfers}

run {} for 3 but exactly 2 CapabilityRole

```

Figure 3.11. Extensions for Needline class in architecture metamodel Alloy code.

The same constraint discovery process is then used, this time focusing on patterns that involve the new needline class and its relationships. At this model scope, the following analytics were derived and can be used to answer the first question: "Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?"

Table 3.3. Second iteration heuristics and corresponding analytics supporting Question 1.

No.	Heuristic	Corresponding Analytic
C8	Any needline connects at least two capability roles	PI.2.1 Needlines that connect fewer than two different capability roles
C9	Any needline connects no more than two capability roles	PI.3.1 Needlines that connect more than two different capability roles
End of Table		

At this model scope, the following analytics can be used to answer the second question: "Are there any patterns present that may create issues for the interoperability of the design described by the model?"

Table 3.4. Second iteration heuristics and corresponding analytics supporting Question 2.

No.	Heuristic	Corresponding Analytic
C10	If any two capability roles exchange some information element, those capability roles connect to at least one common needline	PI.6.1 Capability roles that exchange some information element, but are not connected to any common needlines
C11	Every exchanged information element between any two capability roles is transferred by some needline	(incorporated in PI.7.1)
C12	Every exchanged information element between any two capability roles is transferred by some needline that connects those capability roles	PI.7.1 Information elements exchanged between capability roles that are not transferred by a needline that connects the two capability roles
End of Table		

As a final demonstrator, Figure 3.12 shows an extended version of the metamodel depicted in Figure 3.10. Standards may constrain any of the other identified classes.

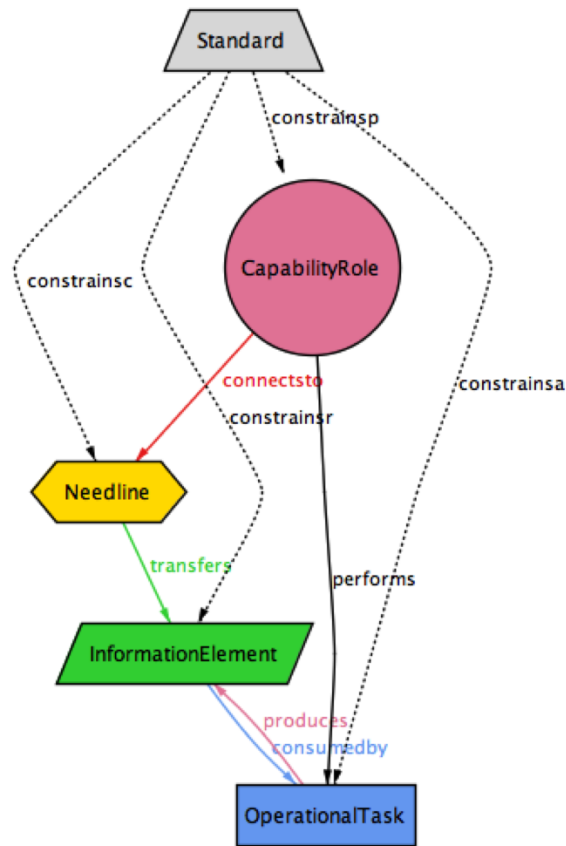


Figure 3.12. Architecture metamodel graph, extended to include standards.

The same constraint discovery process is repeated at the new model scope, focusing on patterns that involve the new standard class and its relationships. At this model scope, the following analytics can be used to answer the first question: "Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?"

Table 3.5. Third iteration heuristics and corresponding analytics supporting Question 1.

No.	Heuristic	Corresponding Analytic
C13	Each capability role is subject to some standard	S.1.1 Capability roles that are not subject to any standard
C14	Each needline is subject to some standard	S.4.1. Needlines that are not subject to any standard
C15	Each operational task is subject to some standard	S.2.1 Operational tasks that are not subject to any standard

Continuation of Table 3.5		
C16	Each information element is subject to some standard	S.3.1 Information elements that are not subject to any standard
End of Table		

At this model scope, the following analytics can be used to answer the second question: "Are there any patterns present that may create issues for the interoperability of the design described by the model?"

Table 3.6. Third iteration heuristics and corresponding analytics supporting Question 2.

No.	Heuristic	Corresponding Analytic
C17	If any two capability roles exchange some information element, those capability roles are subject to some common standard	S.5.1 Capability roles that interact with each other through exchange of information elements, but share no common standard
C18	Every exchanged information element between any two capability roles is transferred by some needline that both connects those capability roles and is subject to some standard that constrains those capability roles	(combination of C12 and C21)
C19	Every needline that connects any two capability roles and transfers an exchanged information element between those capability roles is subject to some standard that constrains those capability roles	S.8.1 Needlines that connect capability roles transferring some information element that share no common standard with those capability roles
C20	If any information element produced by any operational task a_A of any capability role p_A is consumed by any operational task a_B of another capability role p_B , then a_A and a_B are subject to at least one common standard that constrains both p_A and p_B	S.6.1 Operational tasks that exchange some information element that share no common standard with the capability roles performing those operational tasks

Continuation of Table 3.6		
C21	Every exchanged information element between any two capability roles is subject to some standard that constrains those capability roles	S.7.1 Information elements exchanged between capability roles that share no common standard with those capability roles
End of Table		

The standards constraints are implemented in Alloy across three figures: Figure 3.13, which shows the UPIA signatures, Figure 3.14, which shows the global constraints on those signatures, and Figure 3.15, which shows the reciprocal relations and run statement.

```

module UPIA_Schema_Subset_Extension2

sig CapabilityRole {
  --C4: Each capability role performs some operational task
    performs:    some    OperationalTask,
  --Each capability role connects to at least one needline
    connectsto:  set    Needline,
  -- C13: Each capability role is subject to some standard
    psubjectto: some    Standard
}

sig OperationalTask {
  -- C6: Each operational task is performed by some capability role
    performedby: some    CapabilityRole,
  -- An operational task produces zero or more information elements
    produces:    set    InformationElement,
  -- An operational task consumes zero or more information elements
    consumes:    set    InformationElement,
  -- C15: Each operational task is subject to some standard
    asubjectto:  some    Standard
}

[ -- C5: Each operational task produces or consumes at least one information element
  some (produces + consumes)
  -- C1: No operational task both produces and consumes the same information element
  no (produces & consumes)
]

sig InformationElement {
  --C2: Each information element is produced by one or more operational tasks
    producedby:  some    OperationalTask,
  --C3: Each information element is consumed by one or more operational tasks
    consumedby:  some    OperationalTask,
  -- An information element is transferred by zero or more needlines
    transferredby: set    Needline,
  --C16: Each information element is subject to some standard
    rsubjectto:  some    Standard
}

sig Needline {
  --A needline connects zero or more capability roles
    connects:    set    CapabilityRole,
  --A needline transfers zero or more information elements
    transfers:    set    InformationElement,
  --C14: Each needline is subject to some standard
    csubjectto:  some    Standard
}

[ -- C9: Any needline connects no more than two capability roles
  #connects <= 2
  -- C8: Any needline connects at least two capability roles
  #connects > 1
]

sig Standard {
  -- A standard constrains zero or more capability roles
    constrainsp: set    CapabilityRole,
  -- A standard constrains zero or more operational tasks
    constrainsa: set    OperationalTask,
  -- A standard constrains zero or more information elements
    constrainsr: set    InformationElement,
  -- A standard constrains zero or more needlines
    constrainsc: set    Needline
}

```

Figure 3.13. Implementation of standards constraints in Alloy code, part 1 (signatures).

```

-----Global Constraints-----
fact Interoperability
-- Each capability role produces an information element to or consumes an information element
-- from at least one other disjoint capability role
[ all pA: CapabilityRole | some pB: CapabilityRole - pA |
    some (pA.performs.consumes & pB.performs.produces +
        pB.performs.consumes & pA.performs.produces) ]

-- C7: All disjoint capability roles perform disjoint operational tasks (simplifying constraint)
fact { all disj pA, pB: CapabilityRole |
    no (pA.performs & pB.performs) }

-- C10: If any two capability roles exchange some information element, those capability roles connect
-- to at least one common needline
fact { all disj pA, pB: CapabilityRole |
    some (pA.performs.produces & pB.performs.consumes) implies
        some (pA.connectsto & pB.connectsto) }

-- C11: Every exchanged information element between any two capability roles is transferred by
-- some needline
fact { all disj pA, pB: CapabilityRole,
    r: (pA.performs.produces & pB.performs.consumes) |
    (some c: Needline | c in r.transferredby) }

-- C12: Every exchanged information element between any two capability roles is transferred by
-- some needline that connects those capability roles
fact { all disj pA, pB: CapabilityRole,
    r: (pA.performs.produces & pB.performs.consumes) |
    ( some c: (pA.connectsto & pB.connectsto) | c in r.transferredby ) }

-- C17: If any two capability roles exchange some information element, those capability roles are
-- subject to some common standard
fact { all disj pA, pB: CapabilityRole |
    some (pA.performs.produces & pB.performs.consumes) implies
        some (pA.psubjectto & pB.psubjectto) }

-- C18: Every exchanged information element between any two capability roles is transferred by some
-- needline that both connects those capability roles and is subject to some standard that constrains
-- those capability roles
fact { all disj pA, pB: CapabilityRole,
    r: (pA.performs.produces & pB.performs.consumes) |
    ( some c: (r.transferredby & pA.connectsto & pB.connectsto) |
        some (c.csubjectto & pA.psubjectto & pB.psubjectto) ) }

-- C19: Every needline that connects any two capability roles and transfers an exchanged
-- information element between those capability roles is subject to some standard that constrains
-- those capability roles
fact { all disj pA, pB: CapabilityRole,
    c: (pA.connectsto & pB.connectsto) |
    some (pA.performs.produces & pB.performs.consumes & c.transfers) implies
        some (c.csubjectto & pA.psubjectto & pB.psubjectto) }

-- C20: If any information element produced by any operational task aA of any capability role pA is
-- consumed by any operational task aB of another capability role pB, then aA and aB are subject
-- to at least one common standard that constrains both pA and pB:
fact { all disj pA, pB: CapabilityRole,
    aA: pA.performs, aB: pB.performs |
    some (aA.produces & aB.consumes ) implies
        some (aA.asubjectto & aB.asubjectto & pA.psubjectto & pB.psubjectto) }

-- C21: Every exchanged information element between any two capability roles is subject to some standard
-- that constrains those capability roles
fact { all disj pA, pB: CapabilityRole,
    r: (pA.performs.produces & pB.performs.consumes) |
    some (r.rsubjectto & pA.psubjectto & pB.psubjectto) }

```

Figure 3.14. Implementation of standards constraints in Alloy code, part 2 (global constraints).

```

-----Reciprocal Relationships-----
fact {performedby= ~ performs}
fact {producedby= ~ produces}
fact {consumedby= ~ consumes}
fact {connects= ~ connectsto}
fact {transferredby= ~ transfers}
fact {psubjectto = ~ constrainsp}
fact {asubjectto = ~ constrainsa}
fact {rsubjectto = ~ constrainsr}
fact {csubjectto = ~ constrainsc}

run {} for 2 but exactly 2 CapabilityRole

```

Figure 3.15. Implementation of standards constraints in Alloy code, part 3 (reciprocal relations and run statement).

3.3.5 Document heuristics and corresponding analytics

In addition to the mapping tables in the above sections, the heuristics and corresponding analytics are documented in Chapter 2. For additional utility to potential audiences, translations of the analytics into multiple CDMs have also been offered in Chapter 2. The translation tables therein should be a useful starting point for further translations and implementation.

One may wish to further formalize the constraints identified during the modeling process to make them independent of a specific modeling language like Alloy. First order predicate calculus is universal in its basis in mathematics, and thus an appropriate notation for expressing such heuristics or analytics for the purpose of serving as the basis of implementation in any capable language or tool of choice. Reference [4] contains an example list of heuristics (there referred to as axioms) formalized in predicate calculus, which an architect may be interested in using for ideas for general filters that could be used during the model verification process as the model matures.

3.3.6 Extensions for Additional NAVAIR Generic Model Classes and Relationships

Some additional classes and relationships were explored as part of NAVAIR's Generic Model. The classes modeled in this subsection include Vision, Strategic Mission, Capability, Capability Requirement, Organizational Resource, and Capability Usage. Some heuristics and corresponding analytics were generated from these model extensions. Figure 3.16 shows a graph of Generic Model classes and relationships that were studied as part of this research.

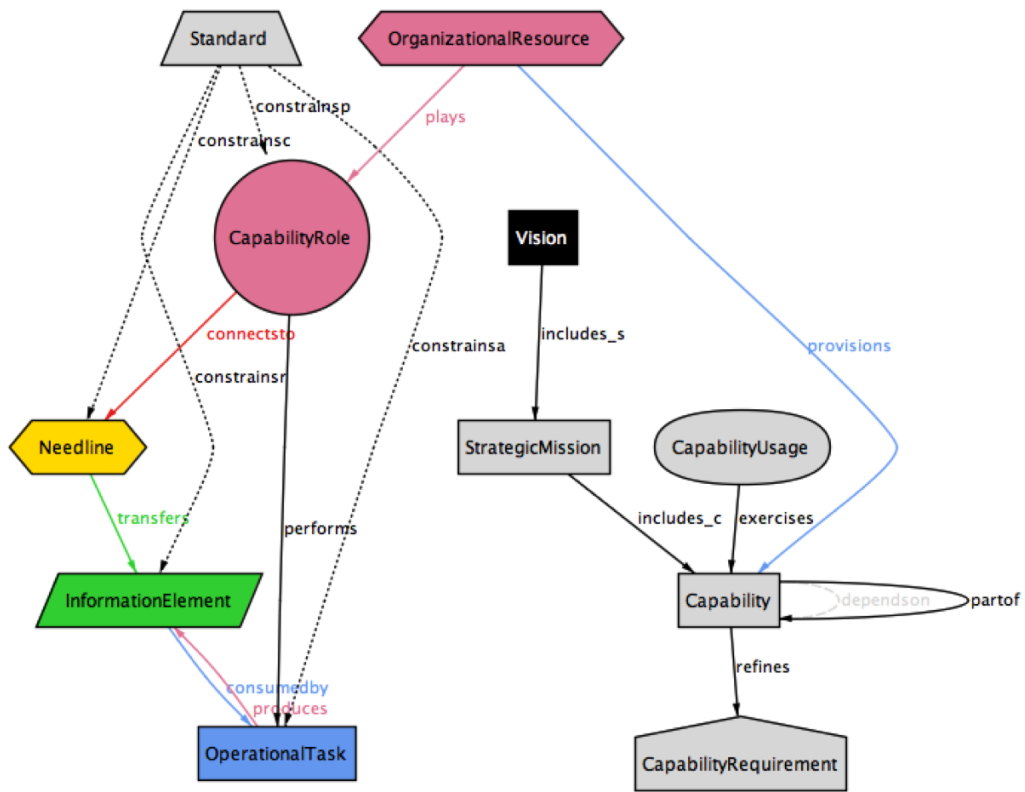


Figure 3.16. Architecture metamodel graph, extended to include more Generic Model classes and relationships.

The extended Alloy model containing all classes in the scope of this research follows in Figures 3.17, 3.18, 3.19, and 3.20.

```

module NAVAIR_Generic_Model

/*****SIGNATURES AND LOCAL CONSTRAINTS*****/
sig Vision {
-- LML mapping: this concept corresponds to a Statement with a Vision label
  type:          Statement_labeled_Vision,
-- C22: Each vision includes at least one strategic mission
  includes_s:    some StrategicMission
}

sig StrategicMission {
-- LML mapping: this concept corresponds to an Action with a StrategicMission label
  type:          Action_labeled_StrategicMission,
-- C23: Each strategic mission includes at least one capability
  includes_c:    some Capability,
-- C24: Each strategic mission is included in at least one vision
  includedin_v:  some Vision
}

sig Capability {
-- LML mapping: this concept corresponds to an Action with a Capability label
  type:          Action_labeled_Capability,
  includedin_s:  set StrategicMission,
  partof:        set Capability,
  composedof:   set Capability,
  refines:      set CapabilityRequirement,
  dependson:    set Capability,
-- C25: Each capability is provisioned by exactly one organizational resource
  provisionedby: one OrganizationalResource,
-- C26: Each capability is exercised by at least one capability usage
  exercisedby:  some CapabilityUsage
}

{ -- C31: Capabilities without a partof relation are included in at least one strategic mission
  (no partof implies some includedin_s)

  -- C32: Each capability with no composedof relation refines one capability requirement
  (no composedof implies one refines)

  -- C33: Capabilities with a composedof relation do not refine a capability requirement
  (some composedof implies no refines)

  -- C34: Each capability is partof no more than one capability
  lone (partof)

  -- C35: No capability is composed of only one capability
  #composedof != 1
}

sig CapabilityRequirement {
-- LML mapping: this concept corresponds to a Requirement with a CapabilityRequirement label
  type:          Requirement_labeled_CapabilityRequirement,
-- C27: Each capability requirement is refinedby at least one capability
  refinedby:     some Capability
}

sig OrganizationalResource {
-- LML mapping: this concept roughly corresponds to an Asset with an OrganizationalResource label
  type:          Asset_labeled_OrganizationalResource,
-- C28: Each organizational resource provisions at least one capability
  provisions:    some Capability,
-- C29: Each organizational resource plays at least one capability role
  plays:        some CapabilityRole
}

sig CapabilityUsage {
-- LML mapping: this concept corresponds to an Action with a CapabilityUsage label
  type:          Action_labeled_CapabilityUsage,
-- C30: Each capability usage exercises at least one capability
  exercises:    some Capability
}

```

Figure 3.17. Alloy model for the Generic Model, part 1 (signatures and local constraints for Vision, StrategicMission, Capability, CapabilityRequirement, OrganizationalResource, and CapabilityUsage).

```

sig CapabilityRole {
-- LML mapping: this concept corresponds to an Asset with a CapabilityRole label
  type:      Asset_labeled_CapabilityRole,
--C4: Each capability role performs some operational task
  performs:  some  OperationalTask,
--Each capability role connects to at least one needline
  connectsto: set  Needline,
-- C13: Each capability role is subject to some standard
  psubjectto: some  Standard,
-- Each capability role is played by at least one organizational resource
  playedby:  some  OrganizationalResource
}

sig OperationalTask {
-- LML mapping: this concept corresponds to an Action with an OperationalTask label
  type:      Action_labeled_OperationalTask,
-- C6: Each operational task is performed by some capability role
  performedby: some  CapabilityRole,
-- An operational task produces zero or more information elements
  produces:   set  InformationElement,
-- An operational task consumes zero or more information elements
  consumes:   set  InformationElement,
-- C15: Each operational task is subject to some standard
  asubjectto: some  Standard
}

{ -- C5: Each operational task produces or consumes at least one information element
  some (produces + consumes)
  -- C1: No operational task both produces and consumes the same information element
  no (produces & consumes)
}

sig InformationElement {
-- LML mapping: this concept corresponds to an Input/Output with an InformationElement label
  type:      InputOutput_labeled_InformationElement,
--C2: Each information element is produced by one or more operational tasks
  producedby: some  OperationalTask,
--C3: Each information element is consumed by one or more operational tasks
  consumedby:  some  OperationalTask,
-- An information element is transferred by zero or more needlines
  transferredby: set  Needline,
--C16: Each information element is subject to some standard
  rsubjectto:  some  Standard
}

sig Needline {
-- LML mapping: this concept corresponds to a Conduit with a Needline label
  type:      Conduit_labeled_Needline,
--A needline connects zero or more capability roles
  connects:  set  CapabilityRole,
--A needline transfers zero or more information elements
  transfers: set  InformationElement,
--C14: Each needline is subject to some standard
  csubjectto: some  Standard
}

{ -- C9: Any needline connects no more than two capability roles
  #connects <= 2
  -- C8: Any needline connects at least two capability roles
  #connects > 1
}

sig Standard {
-- LML mapping: This concept corresponds to an Artifact with a Standard label
  type:      Artifact_labeled_Standard,
-- A standard constrains zero or more capability roles
  constrainsp: set  CapabilityRole,
-- A standard constrains zero or more operational tasks
  constrainsa: set  OperationalTask,
-- A standard constrains zero or more information elements
  constrainsr: set  InformationElement,
-- A standard constrains zero or more needlines
  constrainsc: set  Needline
}

```

Figure 3.18. Alloy model for the Generic Model, part 2 (signatures and local constraints for CapabilityRole, OperationalTask, InformationElement, Needline, and Standard).

```

/*****GLOBAL CONSTRAINTS*****/
-- C36: No entities are part of themselves
fact{ no (iden & ^partof) }

-- C37: No entities depend on themselves
fact{ no (iden & ^dependson) }

-- C38: No entities are part of each other
fact{ no (^partof & ^~partof) }

-- C39: No capabilities that have partof are included in a strategic mission
fact{ no (partof.Capability & includedin_s.StrategicMission) }

-- C40: No capability depends on a capability it is part of
fact{ no (partof.Capability & dependson.Capability) }
-- C41: No capability depends on a capability it is composed of
fact{ no (composedof.Capability & dependson.Capability) }

fact Interoperability
-- Each capability role produces an information element to or consumes an information element
-- from at least one other disjoint capability role
[ all pA: CapabilityRole | some pB: CapabilityRole - pA |
  some (pA.performs.consumes & pB.performs.produces +
    pB.performs.consumes & pA.performs.produces) ]

-- C7: All disjoint capability roles perform disjoint operational tasks (simplifying constraint)
fact { all disj pA, pB: CapabilityRole |
  no (pA.performs & pB.performs) }

-- C10: If any two capability roles exchange some information element, those capability roles connect
-- to at least one common needline
fact { all disj pA, pB: CapabilityRole |
  some (pA.performs.produces & pB.performs.consumes) implies
    some (pA.connectsto & pB.connectsto) }

-- C11: Every exchanged information element between any two capability roles is transferred by
-- some needline
fact { all disj pA, pB: CapabilityRole,
  r: (pA.performs.produces & pB.performs.consumes) |
  (some c: Needline | c in r.transferredby) }

-- C12: Every exchanged information element between any two capability roles is transferred by
-- some needline that connects those capability roles
fact { all disj pA, pB: CapabilityRole,
  r: (pA.performs.produces & pB.performs.consumes) |
  ( some c: (pA.connectsto & pB.connectsto) | c in r.transferredby ) }

-- C17: If any two capability roles exchange some information element, those capability roles are
-- subject to some common standard
fact { all disj pA, pB: CapabilityRole |
  some (pA.performs.produces & pB.performs.consumes) implies
    some (pA.psubjectto & pB.psubjectto) }

-- C18: Every exchanged information element between any two capability roles is transferred by some
-- needline that both connects those capability roles and is subject to some standard that constrains
-- those capability roles
fact { all disj pA, pB: CapabilityRole,
  r: (pA.performs.produces & pB.performs.consumes) |
  ( some c: (r.transferredby & pA.connectsto & pB.connectsto) |
    some (c.csubjectto & pA.psubjectto & pB.psubjectto) ) }

-- C19: Every needline that connects any two capability roles and transfers an exchanged
-- information element between those capability roles is subject to some standard that constrains
-- those capability roles
fact { all disj pA, pB: CapabilityRole,
  c: (pA.connectsto & pB.connectsto) |
  some (pA.performs.produces & pB.performs.consumes & c.transfers) implies
    some (c.csubjectto & pA.psubjectto & pB.psubjectto) }

-- C20: If any information element produced by any operational task aA of any capability role pA is
-- consumed by any operational task aB of another capability role pB, then aA and aB are subject
-- to at least one common standard that constrains both pA and pB
fact { all disj pA, pB: CapabilityRole,
  aA: pA.performs, aB: pB.performs |
  some (aA.produces & aB.consumes ) implies
    some (aA.asubjectto & aB.asubjectto & pA.psubjectto & pB.psubjectto) }

-- C21: Every exchanged information element between any two capability roles is subject to some standard
-- that constrains those capability roles
fact { all disj pA, pB: CapabilityRole,
  r: (pA.performs.produces & pB.performs.consumes) |
  some (r.rsubjectto & pA.psubjectto & pB.psubjectto) }

```

Figure 3.19. Alloy model for the Generic Model, part 3 (global constraints).

```

/*****RECIPROCAL RELATIONS*****/
fact {includedin_v= ~ includes_s}
fact {includedin_s= ~ includes_c}
fact {composedof= ~ partof}
fact {refinedby= ~ refines}
fact {provisionedby= ~ provisions}
fact {exercisedby= ~ exercises}
fact {connects= ~ connectsto}
fact {playedby= ~ plays}
fact {performedby= ~ performs}
fact {producedby= ~ produces}
fact {consumedby= ~ consumes}
fact {connects= ~ connectsto}
fact {transferredby= ~ transfers}
fact {psubjectto = ~ constrainsp}
fact {asubjectto = ~ constrainsa}
fact {rsubjectto = ~ constrainsr}
fact {csubjectto = ~ constrainsc}

/*****MAPPING OF LML TO GENERIC MODEL*****/

lone sig Action_labeled_StrategicMission { }
lone sig Action_labeled_Capability { }
lone sig Statement_labeled_Vision { }
lone sig Requirement_labeled_CapabilityRequirement { }
lone sig Asset_labeled_OrganizationalResource { }
lone sig Action_labeled_CapabilityUsage { }
lone sig Asset_labeled_CapabilityRole { }
lone sig Conduit_labeled_Needline { }
lone sig Action_labeled_OperationalTask { }
lone sig InputOutput_labeled_InformationElement { }
lone sig Artifact_labeled_Standard { }

/*****LML MAPPING CONSTRAINTS*****/
-- Each LML object appears with its type relationship to its corresponding Generic Model object.
fact{ some Action_labeled_StrategicMission implies some type.Action_labeled_StrategicMission }
fact{ some Action_labeled_Capability implies some type.Action_labeled_Capability }
fact{ some Statement_labeled_Vision implies some type.Statement_labeled_Vision }
fact{ some Requirement_labeled_CapabilityRequirement
      implies some type.Requirement_labeled_CapabilityRequirement }
fact{ some Asset_labeled_OrganizationalResource
      implies some type.Asset_labeled_OrganizationalResource }
fact{ some Action_labeled_CapabilityUsage implies some type.Action_labeled_CapabilityUsage }
fact{ some Asset_labeled_CapabilityRole implies some type.Asset_labeled_CapabilityRole }
fact{ some Conduit_labeled_Needline implies some type.Conduit_labeled_Needline }
fact{ some Action_labeled_OperationalTask implies some type.Action_labeled_OperationalTask }
fact{ some InputOutput_labeled_InformationElement
      implies some type.InputOutput_labeled_InformationElement }
fact{ some Artifact_labeled_Standard implies some type.Artifact_labeled_Standard }

/*****RUN COMMAND WITH SCOPE*****/
run {} for 2 but exactly 2 CapabilityRole, 1 Vision, 1 StrategicMission

```

Figure 3.20. Alloy model for the Generic Model, part 4 (reciprocal relations and LML mappings).

The extended Generic Model architecture metamodel using UPIA is mapped to compatible LML classes in Figure 3.21.

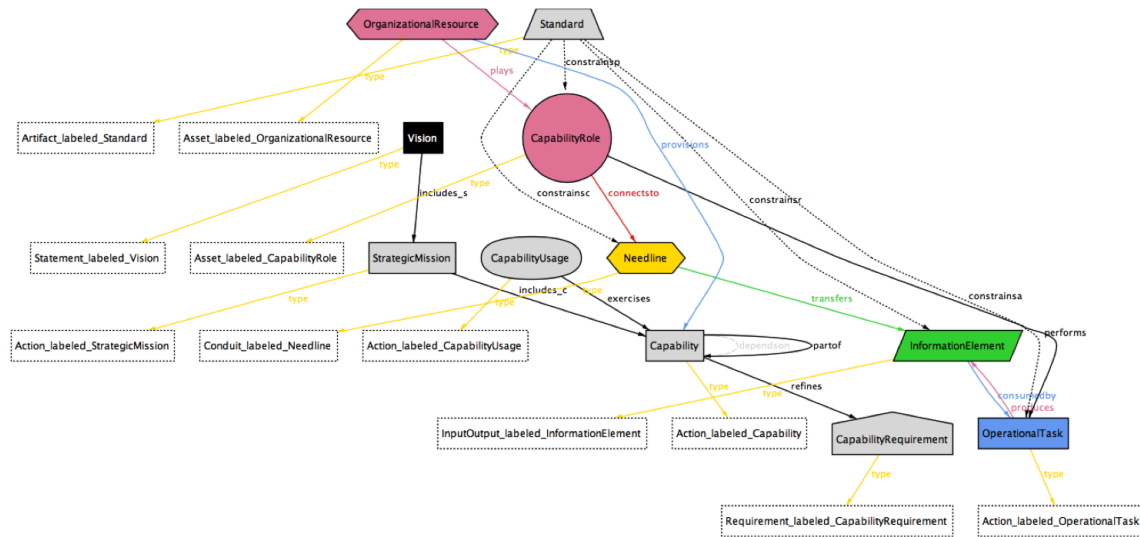


Figure 3.21. Generic model UPIA metamodel graph, with mappings to LML classes and relationships.

Finally, suggestions for additional heuristics resulting from these model extensions are provided in Table 3.7. These new heuristics have not been fully studied for which analysis questions they support, but serve to show that a larger data model results in more opportunities to identify potentially useful architecture model analytics.

Table 3.7. Additional heuristics and corresponding analytics derived from the extended UPIA-based Generic Model.

No.	Heuristic	Corresponding Analytic
C22	Each vision includes at least one strategic mission	Visions that include no strategic missions
C23	Each strategic mission includes at least one capability	Strategic missions that include no capabilities
C24	Each strategic mission is included in at least one vision	Strategic missions that are not included in any vision
C25	Each capability is provisioned by exactly one organizational resource	Capabilities that are not provisioned by exactly one organizational resource
C26	Each capability is exercised by at least one capability usage	Capabilities that are not exercised by any capability usage
C27	Each capability requirement is refined by at least one capability	Capability requirements that are not refined by any capability

Continuation of Table 3.7		
C28	Each organizational resource provisions at least one capability	Organizational resources that do not provision any capability
C29	Each organizational resource plays at least one capability role	Organizational resources that do not play any capability role
C30	Each capability usage exercises at least one capability	Capability usages that do not exercise any capability
C31	Capabilities without a <i>partof</i> relation are included in at least one strategic mission	Capabilities that do not have a <i>partof</i> relation, and are not included in any strategic mission
C32	Each capability with no <i>composedof</i> relation refines one capability requirement	Capabilities without a <i>composedof</i> relation, and do not refine one capability requirement
C33	Capabilities with a <i>composedof</i> relation do not refine a capability requirement	Capabilities with a <i>composedof</i> relation, and refine some capability requirement
C34	Each capability is part of no more than one capability	Capabilities that are part of more than one capability
C35	No capability is composed of only one capability	Capabilities that are composed of only one capability
C36	No entities are part of themselves	Entities that are part of themselves
C37	No entities depend on themselves	Entities that depend on themselves
C38	No entities are part of each other	Entities that are part of each other
C39	No capabilities that have <i>partof</i> are included in a strategic mission	Capabilities that have <i>partof</i> , and are included in a strategic mission
C40	No capability depends on a capability it is part of	Capabilities that depend on a capability it is also part of
C41	No capability depends on a capability it is composed of	Capabilities that depend on a capability it is also composed of
End of Table		

3.4 Implementation

Once such heuristics are codified in a capable tool compatible with the data model used for their expression, specific instances of noncompliance with the heuristic in an actual architecture model can be flagged using analytics. Such tools implement these rules as filters or warnings that may be toggled on or off at each architect's or organization's discretion for automated assistance in finding instances in the model that fit a certain pattern. Implementation for several example analytics has been accomplished in two different systems

engineering tools to date: CORE using SDL [6], and Innoslate using LML [42]. Full implementation was beyond the scope of this research, but the small implementation experiments that were conducted demonstrate that automation of the analytics is possible in different tools and languages.

3.5 Discussion and Summary

The content of this chapter has demonstrated the application of the method in [4] to the UPIA, illustrating compatibility between the classes and relationships shared by LML and UPIA. Giammarco [3] further illustrates compatibility with DM2.

Model maturity is subjective, and appropriate heuristics can be applied by individual architects or organization at different phases of a system's design. It is worth emphasizing that not all heuristics identified need be enforced in all models, since not all may conform to the preferences and practices of every architect. Nor could one expect all heuristics to hold true early in a model's development, as a model takes time to be populated with the correct entities and relationships. The heuristics may be used to set organizational standards for design maturity, by establishing formal definitions for concepts currently lacking a formal means for assessment in an architecture model. In [4], the example heuristics pertain to assessing for non-interoperability of a design. The design is considered non-interoperable if, for example, an asset generates an input/output that is not received by any other asset, if an asset receives an input/output that is not generated by any other asset, or if an input/output that is exchanged between any two assets is not transferred by some conduit that connects to those two assets. Additional heuristics for non-interoperability may be developed by extending the scope of the data model considered [3]. Thus, this method offers a formal platform for experienced architects and engineers to debate the meanings of hard to quantify terms, and create a knowledge base of their collective experiences in the process.

The content of this chapter provides NAVAIR with the ability to specify not only the Generic Model, but the acceptable use cases for the Generic Model. The constraints are used to purge known poor modeling practices, admitting only good modeling practices. These specifications can then be provided to solution developers as self-assessment criteria before delivering their solution-specific models. With these criteria, NAVAIR will have a tool-agnostic way to do some quality control on all architecture models developed for them.

The expression of these modeling heuristics in the Generic Model language now enables anyone to query their Generic Model-compliant architecture model for the presence or absence of known interoperability deficiencies and other typical problems.

Monterey Phoenix (MP) uses the same concept of generating examples using a small scope to find errors, except MP is focused on behavior. Scope in MP is used to limit iteration (the number of times an activity occurs, where as scope in this use of Alloy limits the number of

classes. In reality, many more relationships / iterations may occur, but the idea is that most errors can be exposed on small examples, and this is called the Small Scope Hypothesis (the same principle used by Alloy's SAT solver) [33].

Since the model maturity and model design heuristics presented herein are expressed at such a high level of abstraction, they have applications to modeled architectures in many domains. Discovered patterns formalized as heuristics can be distributed, codified in a tool of choice, and sought out in actual architecture models of systems using automation. The formalized heuristics provide a basis for assessing development progress of modeled architectures, as exemplified in Tables 1 and 2 in [4]. Each heuristic is written to be minimally constraining, so that any combination of heuristics may be used to restrict the design at given level of architecture maturity according to an architect's preference. The collection of heuristics is intended to grow as the user community identifies more model patterns for abstract capture.

As an outcome of formalization of such expectations, it becomes possible to establish measurable analytics for both model maturity and design maturity that can be tailored for different events and decision points. Metrics tied to the results of queries based on these analytics can be used to provide a more quantitative and precise basis for assessing model and design characteristics, such as interoperability of a design, and to remove subjectivity from the assignment of descriptors, levels, and other qualitative instruments.

Although the UPIA specification served as the basis for the examples in 3, other data models may be substituted for use with the method presented. Likewise, interoperability served as one example characteristic of concern; the same method could potentially be used for the assessment of other nonfunctional system-wide design characteristics such as security, safety, survivability, and other -ilities.

Formally derived heuristics and analytics can provide systems engineers and integrators of complex systems with the ability to create clear, unambiguous, and tool-independent expectations for models and designs. They can be used to establish requirements and instructions that can be tailored for any system under design and its corresponding decision points. Formal heuristics can be used to define what, precisely, a government, corporation, or other customer/stakeholder means by terms like interoperable, integrated, complete, mature, sustainable, and other quality attributes for which many principles, guidelines and policy statements exist in natural language [16] [19] [43], but for which formal specifications are largely still absent. Formal analytics with target values for architecture model qualities such as interoperability of a design provide clear, tailorable, testable requirements that can be used to assess and measure the degree to which architecture models comply with expectations for these quality attributes over time as the model matures. Architects may then use the analytics to conduct pre-assessments of their architecture models and ensure they match the formally defined expectations before delivery to and assessment by the customer. The formal method presented can be used to set and validate architecture model

and design criteria using automation, giving the humans working on a particular system a collective body of knowledge and experience to draw upon and contribute back to, building an architecture model patterns repository in the process.

In formalizing our human experience with desired and undesired characteristics of systems into patterns, we externalize that experience so that others without direct involvement in the resulting lessons learned can benefit. In his International Test and Evaluation Association (ITEA) journal article [44], RADM Dunaway states that “The bottom line is that when we have the discipline to execute our known best practices we minimize the cost to the taxpayer and the adverse impact to warfighting capability.” Formal methods provide an elegant way to exercise that discipline, providing a vehicle to address our collective memory loss of best practices. Applied to architecture models, they provide a means to share patterns for architecture best practices and lessons learned with minimal error and high reusability. The heuristics and analytics developed in this research are simultaneously abstract and precise: abstract enough to apply in multiple domains, and precise enough to provide deterministic assessment results on a given set of data when implemented in a tool. This formal method provides an analytic underpinning for assessment of models and designs based on architecture entities and relationships, and lays the groundwork for further formalization in the pursuit of precise and measurable definitions for system-wide characteristics that continue to be difficult to assess.

CHAPTER 4:

Graphical Templates for Executable Behavior Models

Kristin Giammarco

This chapter presents a number of reusable templates for behavior models. The graphical Lifecycle Modeling Language (LML) action diagram notation is used for expressing the templates given its simplicity and level of abstraction for translating into other languages. The Innoslate modeling tool makes the LML available, accessible, and executable using a small set of model constructs. Readers familiar with Systems Modeling Language (SysML) should consider these templates since analogous representations are likely available with a little translation work.

4.1 Actions

The LML employs different types of actions arranged in a sequence on one or more concurrent branches to denote logical behavior. **And** (concurrency), **or** (alternate), **loop** (iteration), and **sync** (synchronization) action logic are supported. The different types of actions are described in the following subsections.

4.1.1 Action

An action may be used to model an activity, process, phase, or other event with a time duration of zero or more. A plain action is independent of logic construct.

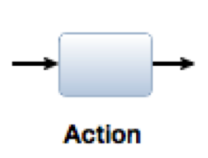


Figure 4.1. An action is denoted by a rectangular box. Actions can follow and precede other actions. Icon source: Innoslate.

4.1.2 Or Action

An "or" action has at least two possible alternative outcomes, and so graphically includes two or more trailing branches upon which following actions can be arranged.

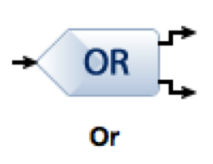


Figure 4.2. An "or" action is denoted by a box leading with a diamond shape on the left, and a trailing branch for each of its possible outcomes. Icon source: Innoslate.

There are at least two possible styles that can be employed in naming "or" actions. The first style names the "or" action after the decision in question, as in Figure 4.3. The trailing branches in this style are typically labeled to the effect of "Yes" and "No," or "Affirmative" and "Negative." The second style assigns a definitive action-oriented name to the "or" action rather than a question, as in Figure 4.4. The trailing branches are then directly labeled with descriptions of the outcomes they represent. "Or" actions may have more than two branches if there are more than two possible outcomes.

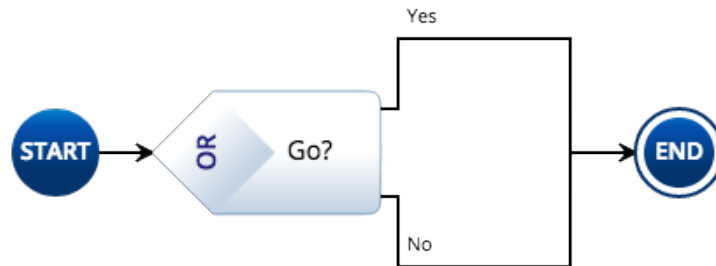


Figure 4.3. An "or" action, named in question format. Outcomes are of a yes or no nature.

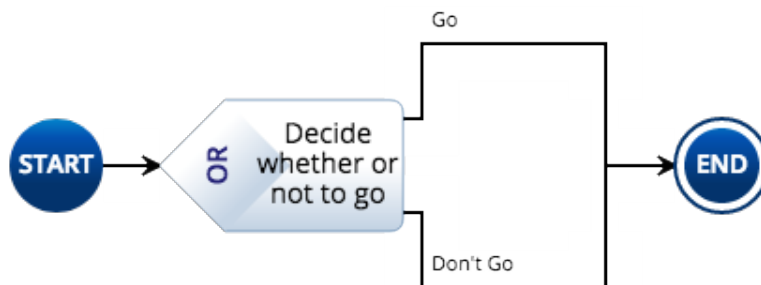


Figure 4.4. An "or" action, named in activity format. Outcomes are the possible decisions.

LML action diagrams provide the modeler with the means to use one of the following approaches to determine which trailing branch to select during model execution:

- *Probability*. This approach assigns a probability to each outcome (branch).
- *Resource availability*. This approach makes a selection automatically based on the quantity of a specified resource.
- *Custom script*. This approach allows the modeler to specify custom conditions for selecting each outcome (branch).

4.1.3 Loop Action

A "loop" action has exactly two possible alternative outcomes: to continue the loop, or to exit the loop. If the Continue branch is selected, the "loop" action repeats. If the Exit branch is selected, the logic proceeds to the next action.

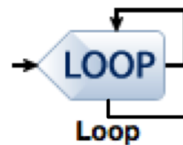


Figure 4.5. A "loop" action is denoted by a box leading with a diamond shape and a trailing branch that loops back into the top of the box. A second branch exits the loop from the bottom of the box. Icon source: Innoslate.

LML action diagrams provide the modeler with the means to use one of the following approaches to determine when to end each loop:

- *Loop iterations*. This approach assigns a specific number of times the loop is to iterate before exiting.
- *Probability*. This approach assigns a probability of continuing the loop or a probability of exiting the loop.
- *Resource availability*. This approach selects to continue or exit the loop automatically based on the quantity of a specified resource.
- *Custom script*. This approach allows the modeler to specify custom conditions for loop continuation or exit. Examples of this approach are included in some of the templates that follow.

At this time, the LML does not employ the concept of scope used by MP to limit iteration of looping events. As a consequence, LML and other languages do not exhaustively produce all scenario variants up to a predefined global scope limit as does MP. MP is fundamentally different from other languages in how it uses small scope to govern loop iteration, rather

than having a modeler specify case-by-case logical prescriptions to govern each loop in a model. In this way, MP permits more possible scenario variants to emerge.

4.1.4 Sync Action

Unlike other types of actions, a "sync" action has two or more leading branches representing alternate paths to the "sync" action. A "sync" action accepts the input of the *first* of its preceding branches to finish, and cancels the branches that did not finish. It can be thought of as a finish line in race between competing alternatives. The first "sync" branch(es) to complete are accepted as the precedent, and the remaining incomplete branches are ignored. Examples are shown in some of the templates that follow.

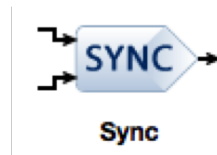


Figure 4.6. A "sync" action is denoted by a box trailing with a diamond shape, and a leading branch for each of its possible predecessors. Icon source: Innoslate.

4.2 Branches

A action diagram branch is analogous to a lifeline on a sequence diagram in that time progresses through actions on the branch from the beginning to end of the branch.

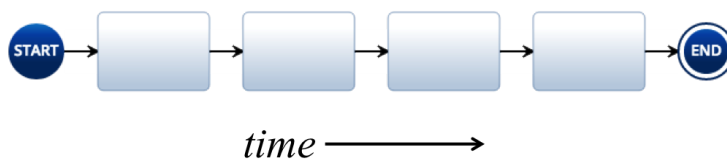


Figure 4.7. Time proceeds from the beginning to the end of the branch (here, left to right).

4.2.1 Single Branch

All of the actions on a single branch happen sequentially, in a definite order. The timing of an action then depends on the timing of preceding actions. For example, in Figure 4.7, the third action cannot begin until the second action has completed, and second action cannot begin until the first action has completed. A single branch is analogous to a series circuit.

4.2.2 Multiple Branches

When two or more branches are put in parallel, actions on each branch can be made concurrent, or independent of the timing of actions on other branches. In this arrangement, the first action on each branch can begin as soon as its prerequisites are met (if none, they begin at the same time). Parallel branches are analogous to a parallel circuit.

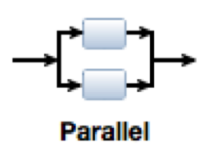


Figure 4.8. Parallel branches denote logical concurrency. Icon source: Innoslate.

4.2.3 Branch Assets

If all the actions on a branch are performed by the same asset, then the branch may be formally associated with that asset. This approach of grouping actions performed by the same asset onto the same branch is a practice we recommend to avoid ambiguity regarding which part of a system, or which system, organization or person is responsible for each action.

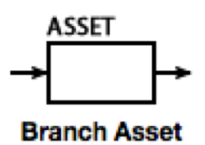


Figure 4.9. If all the actions on a given branch are performed by the same asset, then the branch can be labeled with the performing asset. Icon source: Innoslate.

4.3 Interaction Types

Interactions among actions take place in the form of resources or input/outputs carrying Energy, Matter, Material wealth, or Information (EMMI) [45]. Resources can be used for a period of time then returned (e.g., bandwidth, personnel), or used up / transformed (e.g., fuel, food). Actions that seize or consume resources depend on those resources to complete. Actions may also be modeled to produce resources.



Figure 4.10. Actions may depend on the presence of sufficient resources. Icon source: Innoslate.

Two types of input/outputs are triggers and optional input/outputs. Optional input/outputs are effectively inert in the behavior logic specification, in that the receiving action does not depend on them to execute. Triggering input/outputs, on the other hand, are regarded in the behavior logic since they introduce a dependency between a sending and a receiving action. The receiving action cannot complete until the sending action completes, similar to two actions in series. What the triggering input/output brings to the logical specification is the means to create dependencies between actions on different branches, or even entirely different diagrams or models. Coordinate statements in MP are used to create such dependencies among events in different roots. MP coordinate statements are higher level (more abstract) specifications, and have more expressions to specify more complex coordination. The templates that follow demonstrate how triggering input/outputs are used to control the logical sequence and timing of events in LML action diagrams. Development of SysML analog templates are a topic for future work.



Figure 4.11. Actions may depend on input/outputs such as information or instructions. Triggering input/outputs (required dependencies) are green; optional input/outputs are grey. Icon source: Innoslate.

4.4 Templates

The first five templates mainly illustrate different styles for organizing actions onto branches. The remaining templates incorporate some examples that use script logic to direct the selection of decisions as the model executes, permitting different variants of the model in simulation. These represent a small set of common templates, which is not meant to be exhaustive. More complex behaviors can be achieved from nesting templates within other templates.

4.4.1 One-branch Sequence

In the **One-branch Sequence** template, actions are arranged on one branch in a simple sequence. This straightforward template was introduced during the discussion of branches in the prior section. In Figure 4.12, "Perform A," "Perform B," and "Perform C" represent three actions performed in sequence (A, then B, then C). "Perform B" does not begin until "Perform A" completes, and "Perform C" does not begin until "Perform B" completes. This is considered an atomic template since it is not composed from other templates.

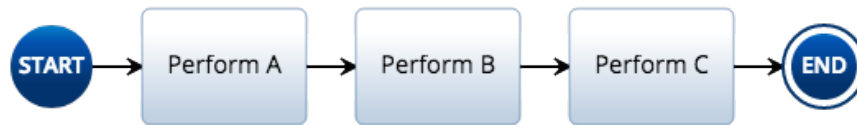


Figure 4.12. A one-branch sequence.

4.4.2 Multi-branch Sequence

The **Multi-branch Sequence** template is made up of at least two branches. Each branch is allocated to a branch asset, and composed of some instance of the **One-branch Sequence**. Since this template is composed of other templates, it is considered a composite template. In Figure 4.13, The Environment performs action A, which generates an output that triggers the system to perform action B. When Perform B finishes, the system performs C, then D. Perform D generates an output that triggers some action in the environment (Perform E). The environment's and the system's functions are on parallel branches, so triggering is used to capture the timing dependencies.

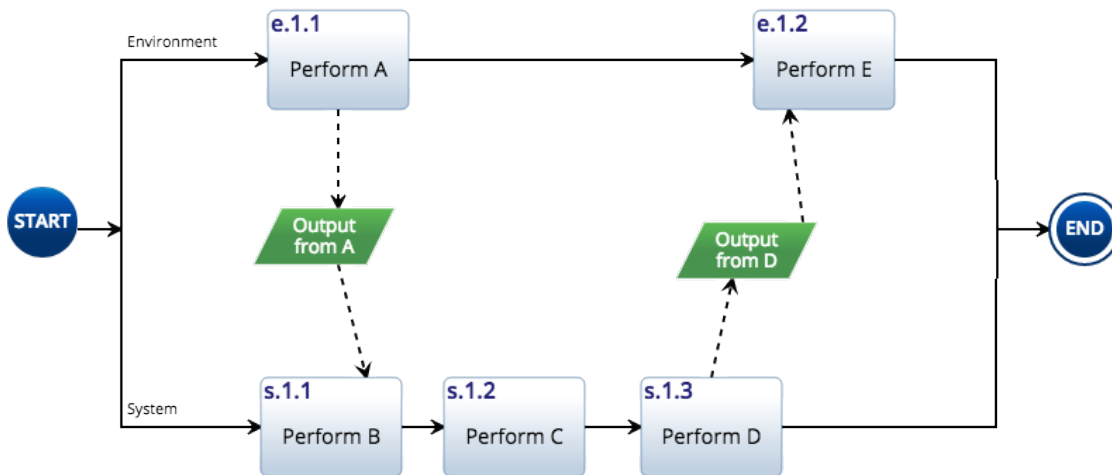


Figure 4.13. A multi-branch sequence.

4.4.3 Single Decision Point

In the Single Decision Point template, we recognize the One-branch Sequence nested in the Multi-branch Sequence. In addition to these templates, we have another element of interest - that is, an "or" action that makes a decision resulting in two possible outcomes. In one outcome the system performs action D, and in the other outcome the system performs action F.

Now, it happens in this template, as it often does in actual models, that a decision on one branch (here, the system) impacts activities on another branch (here, the environment). If the system performs D, then the environment reacts by performing E. On the other hand, if the system performs F, then the environment reacts by performing G.

In simulation of the "or" action, only one branch (outcome) will be chosen (via one of the means earlier discussed), and the other branch (outcome) will be ignored. The environment will likewise not execute all of its possible responses to the decision, and so we need a mechanism to cancel / disregard the responses not driven by a selected decision. The "sync" action is used to accept the input of its triggered leading branch, and then prune its other branches from the simulation.

This pairing of an "or" action on one branch with a "sync" action on another branch enables the representation of decisions and their various outcomes in different parts of the system or the system's environment. As a convention, the duration of the "sync" action can be set to zero if it is used as a pruning device and not a substantial and time-consuming activity of the performing asset.

When using this template, be aware of the following limitation. The "sync" action logic depends on the timing of its preceding activities, and branch pruning is done in a memoryless manner. What is typically no issue for a non-repeating sequence can become uncoordinated once the decision is set to repeat on a loop. This is because actions may have mixed durations (times to complete), and action durations may be set such that the "sync" action is still processing its reaction to the last decision when a new decision is already firing a new trigger at it. In this situation it is possible to generate an uncoordinated response. The MP language provides abstract coordination that prevents such uncoordinated interactions explicitly in the formal logic. The symptoms of the issue can be avoided in discrete event simulation by paying careful attention to duration settings. Doing so, however, only alleviates the symptoms of the underlying lack of formal coordination logic in this template. Therefore, use this template with careful attention on the action duration settings.

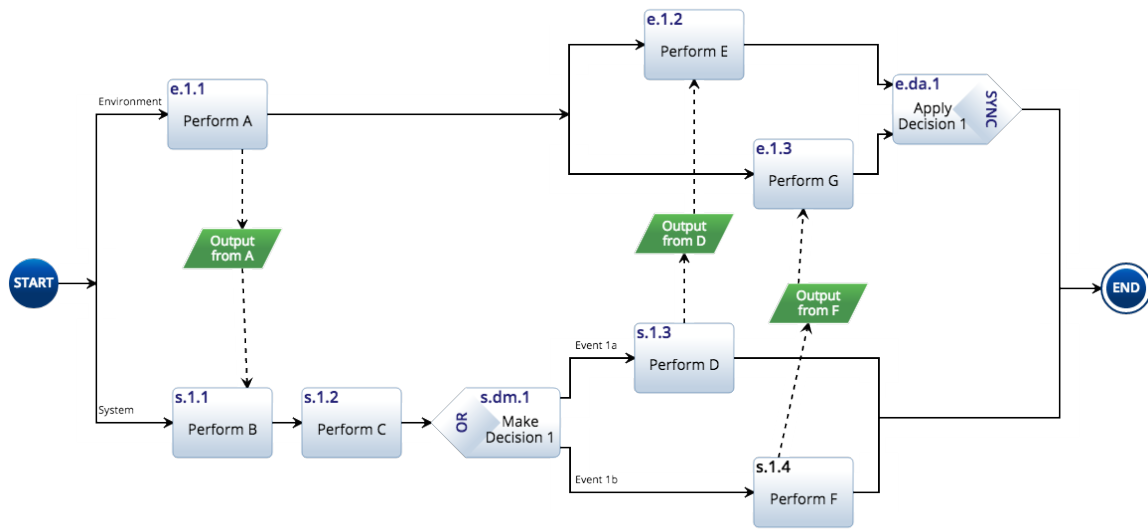


Figure 4.14. A single decision point.

4.4.4 Separation of Actors

This is a model partitioning template that can be employed to separate actions performed by different components or systems onto separate diagrams. This template becomes essential as the model size grows. As more behaviors for each system or component are included, it becomes harder and harder to fit them all on one diagram without making the diagram illegible. Architects should work to make the models as simple and easy to understand as possible to facilitate error discovery.

There are three related models in Figure 4.15. The model shown in the center uses

Multi-branch Sequence to group all environment activities and all system activities into separate, concurrent actions that each decompose into their respective composite actions. The top model shows the decomposed environment actions, and the bottom model shows the decomposed system actions. Critical to this approach, the triggering input/outputs in the environment model on the top of the figure are the same as those shown on the system model on the bottom of the figure. This is, in fact, the same behavior logic as shown in Figure 4.14. The environment performs A and generates output from A. That same output from A triggers the system to perform B. The system then performs C, then makes a decision to perform either D or F. If "Perform D" is chosen, then output from D triggers the environment to perform E and disregard "Perform G." If "Perform F" is chosen, then output from F triggers the environment to perform G and disregard "Perform E." So, this template is simply a regrouping of the same actions and input/outputs shown in Single Decision Point.

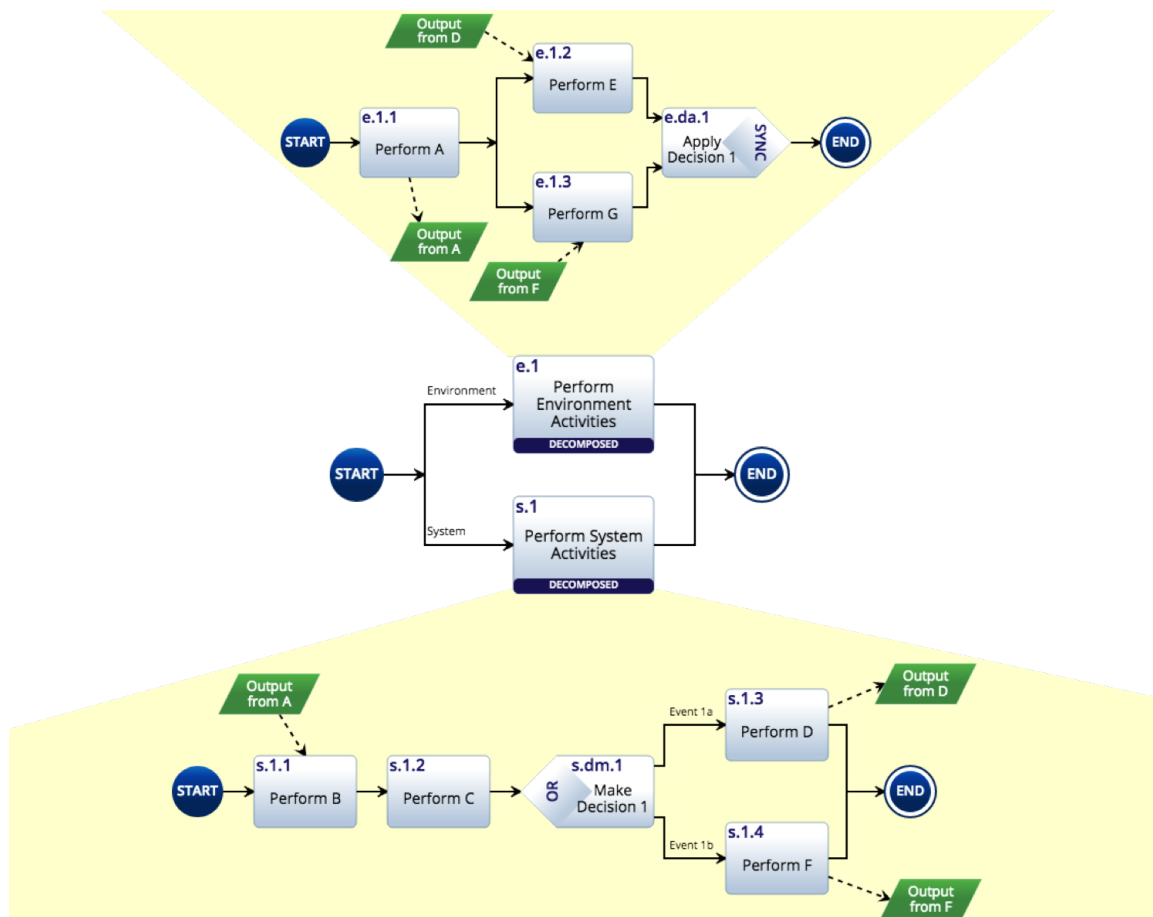


Figure 4.15. A separation of actors, where the behaviors of each are partitioned onto different diagrams.

4.4.5 Separation of Phases

We encourage breaking lengthy behavior models into phases, so that chunks of related actions can be partitioned into time-relevant segments. Phases are time segments of a process model, mission, use case, or other sequence of related events. For example, the HADR model used as an example in Chapter 2 has a mission that can be broken up into phases. Figure 4.16 shows the Ingress, OnStation, Egress and Postflight phases of the mission using One-branch Sequence, with each action decomposing into its constituent behaviors. The Phase.1 Ingress action, for example, decomposes into the actions shown in Figure 2.15, which uses Multi-branch Sequence.

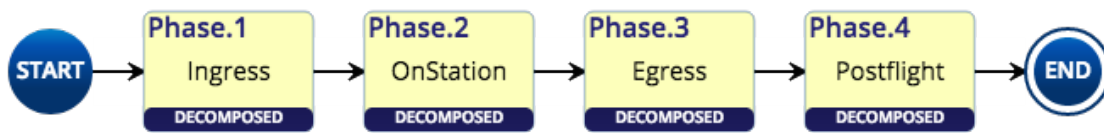


Figure 4.16. Separation of phases applied to the HADR mission example.

It is possible to compose a sequence of separate models using One-branch Sequence for Separation of Phases. The parent actions used to develop the models in Figures 4.13, 4.14, and 4.15 can be strung together as in 4.17. Actions and input/outputs can even be reused across phases, as Giles shows in her doctoral research on Mission-Based Architecture for Swarm Composability (MASC) [46], to make the executable models modular and composable.

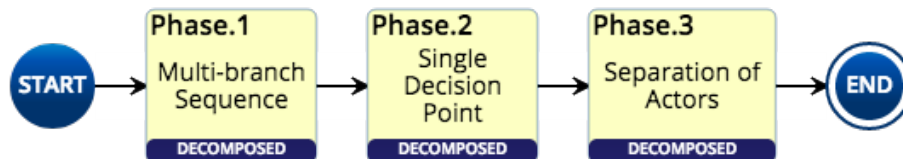


Figure 4.17. Separation of phases applied to the template examples.

As a final general example (not pictured), an Information Technology (IT) operation may be partitioned into phases for authentication, system usage, and session termination - each phase composed of multiple possible alternative behaviors that can be modeled, verified and validated separately before testing the entire sequence of all phases.

4.4.6 Loop until Failure Mode

The modeler may desire to continue a loop until a certain condition, such as a failure mode, is reached. This template shows how to do this in a simple model of a system and its environment. In the notional scenario depicted in Figure 4.18, the environment emits an output (Perform I) detected by the system (Perform J). The emission/detection cycle continues until the system detects a failure mode, at which point the scenario ends.

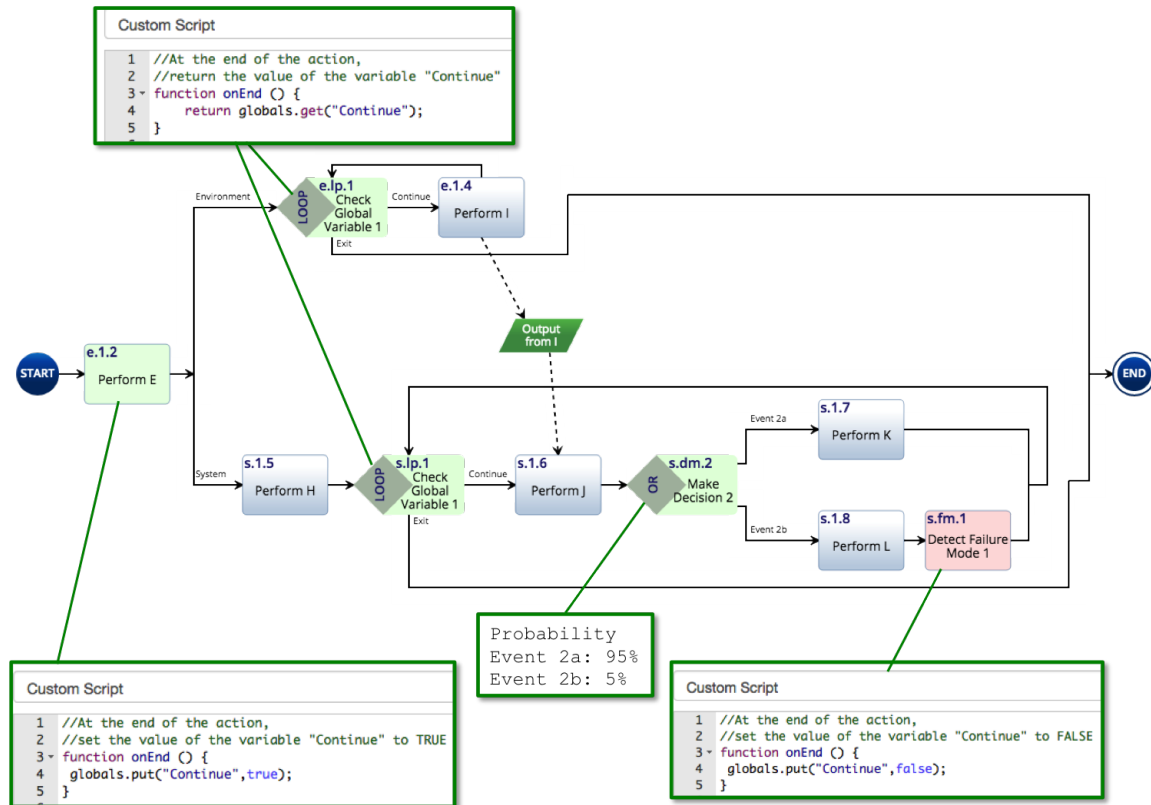


Figure 4.18. Continuation of loops until a failure mode occurs.

A modeler can set up this type of scenario using custom scripts. Each action using script logic is color-coded light green (an optional convention, color-coding action boxes makes it easier to remember which actions contain scripts). First, a global variable "Continue" is set to "true" inside the preceding action "Perform E." Note that "Perform E" may be present inside an action that precedes this model, via Separation of Phases. For completeness of the illustration, we show "Perform E" as the first action in this model.

Having been initialized, the value of the global variable "Continue" can now be retrieved by other actions. The first actions to make use of it are the loops in the environment and the system, both of which have the same custom script to return the value of "Continue." Since

we initialized it to "true," the "loop" action will initially select the Continue branch in each case and proceed with performing I and performing J, respectively.

Next, the system makes a decision - perhaps concerning an action using the detected input, or perhaps involuntarily concerning a failure of some part of the system. Notional probability settings of the "or" action are called out. In this example, there is a 95% chance that there will not be failure. The loops continue their iteration until the 5% probability branch is selected, and the value of Continue changes to "false."

The lower branch of the "or" action shows how to set up a failure mode detection (and change Continue to "false"). If the lower branch is selected (a 5% chance), the system will perform L and then detect a failure mode (light red action). Upon completion of "Detect Failure Mode 1," a custom script sets the value of the Continue variable to "false." The next iteration of both loops will result in the respective Exit branches to be selected, and the simulation ends.

Be aware that "loop" actions are subject to the duration settings. For example, if the environment's loop duration is one minute and the system's loop duration is one hour, then the system will not be keeping up with the triggering input/outputs being sent from the environment (which may or may not have consequences, depending on the intended use of the model). On the other hand, if the environment's loop is one hour and the system's loop is one minute, then "Perform J" will be waiting an hour to receive the trigger from the environment before proceeding through the rest of the model. Be aware of the consequences of duration settings when using this template. Use concurrency (parallel branches) to separate functionally different actions (such as move and sense). Using MP to model, verify, and validate behaviors leverages rigorous formal specification to automatically coordinate and synchronize repeating activities independent of duration.

4.4.7 Loop until Low Resource Supply

The modeler may wish to continue iterating actions in a scenario until a certain resource is depleted below a certain threshold. This template shows how to accomplish this, also using a simple model of a system and its environment. In the case of Figure 4.19, the environment presents different kinds of objects for potential detection by the system. Each iteration consumes 1 unit of fuel. The environment keeps presenting objects, and the system keeps detecting objects, until the system's fuel falls below 5 units.

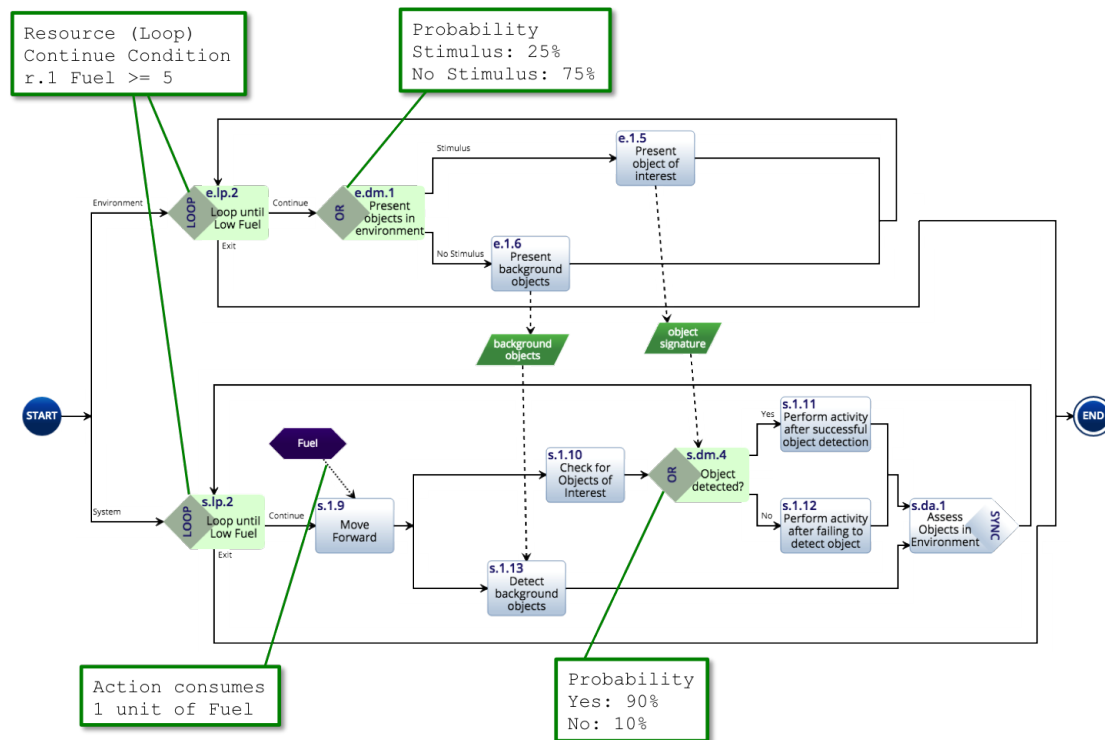


Figure 4.19. Continuation of loops until a resource is in low supply.

There are several practical issues with this template worth discussing. First, there is a lack of concurrency of movement (propulsion function) and object detection (sensing function). These two functions are forced by this logic into lock-step, which is at best not ideal and at worst unrealistic. A more accurate model may have the movement function (Move Forward) on its own concurrent branch, in parallel with sensing, so that the movement and sensing functions can operate independent of one another. Another practical issue is the way in which the environment presents pre-classified objects (objects of interest and background objects). In reality, the environment does not classify objects, but the system (or users of the system) do. A more accurate model of the environment might only present one type of object, which would then undergo classification by the system. The third practical issue may be the omission of a consequence for failing to detect an object of interest. The primary purpose of these templates is to illustrate how to control the flow of logic, but we are leveraging the opportunity to also describe typical anti-patterns that are commonly encountered in student models.

First, both loops have a custom script that defines the condition to select the Continue branch. The continue branch will be selected in this case if the fuel resource has a value equal to or greater than 5. The fuel resource itself is separately initialized with a value of 10, so the first time the "loop" actions are encountered, they both select their respective

"Continue" branches.

The system moves forward, and in so doing, consumes 1 unit of fuel (defined as an attribute of the "Move Forward" action's relationship with the fuel resource). Meanwhile, the environment presents either background objects (75% probability) or an object of interest (25% probability). The system is primed to detect the background objects, if presented by the environment, or an object of interest, if presented with an object signature from the environment. The system contains a decision point to capture whether the object signature from the environment was in fact detected (90% probability) or missed (10% probability). The system takes appropriate actions based on the outcome of this decision point. The loops repeat until the fuel resource falls below 5 units, at which point both loops exit.

4.4.8 Countdown Loop

The modeler may wish to continue iterating actions in a scenario until a counter reaches zero. This template is useful when loop iteration is conditional based on success or failure to detect an object (in this case, a Tactical Disc (TD), as described in the Tactical Disc Clearance System DRM [47]), and hard-coding the number of loop iterations does not give the desired result. The template in Figure 4.20 contains a system, environment, and an operator, and is on the edge of legibility as a tradeoff for showing the behavior of three different actors on one diagram. The need for the Separation of Actors template becomes obvious in this model.

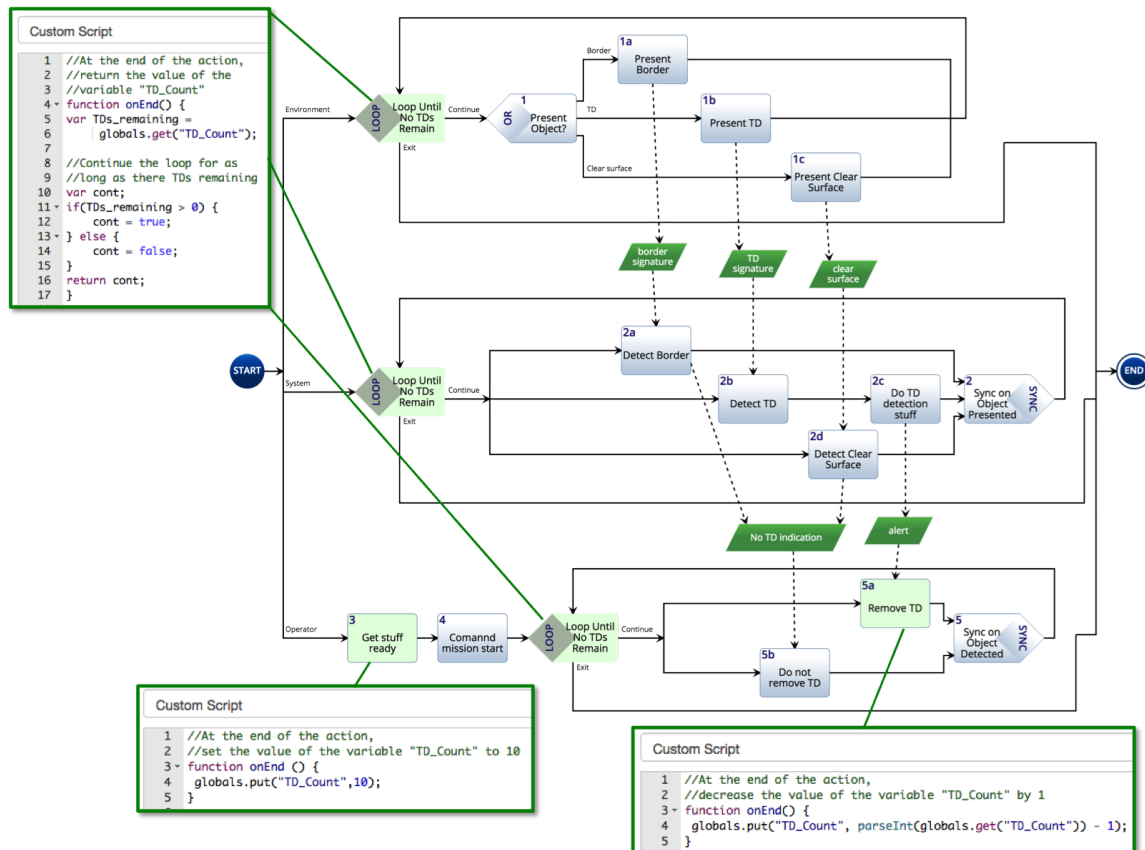


Figure 4.20. Continuation of loops until a counter reaches zero.

Here, a global variable called "TD_Count" is initialized to 10 in a notional action of the operator called "Get stuff ready" (which clearly could use a more descriptive name in an actual model). Each of the three loops select their respective "Continue" branches the first time through, since upon checking the value of "TD_Count", they find the number of TDs remaining to be greater than zero. The environment presents three kinds of objects that the system is primed to detect: borders, TDs, and clear surface. Each loop of the environment will generate one of these types of objects for detection. If a TD is detected, the system will alert the operator, who removes the TD from the search area. The loops continue until all 10 TDs are detected and removed.

CHAPTER 5:

Virtual Testing of System Behavior using Executable Models in Monterey Phoenix

Bruce Allen and Kristin Giammarco

The previous chapters have explored various aspects of architecture modeling, including conceptual data modeling and its use for expressing heuristics and analytics, and patterns for different architecture views. This chapter focuses only on the behavior aspect of a system, and utilizes a formal approach and language for modeling system behaviors and business processes known as Monterey Phoenix (MP).

This chapter first summarizes a method for virtual testing of system behavior using MP, which is fully documented in Appendix D of [21]. Next, we report progress on the development of a locally installable MP tool called MP-Python, and then use it to illustrate examples of UAV behaviors from the model started in earlier chapters. Finally, the MP graphical output is related back to the Generic Model to illustrate the main data model concepts that are needed for MP analysis.

5.1 Method for Using MP

The unique model partitioning strategy behind MP is what enables its thorough scenario generation capability. Component behaviors are modeled separately from one another, and component interactions are modeled separately from component behaviors. An event trace generator computes all possible combinations of events from these models, then prunes away the invalid combinations using the interaction constraints as rules to disallow certain behaviors from a logical, simplification, or design perspective [48]. In this "generate all, then prune" approach, more behaviors appear in the model output for inspection and decision making about whether to allow the behavior to remain, or to reject the behavior by modifying the model or constraints. The auto-generated event trace scenarios are inspected for unwanted behaviors arising as a result of misrepresentation of a system's behavior, or missing logical, simplification, or design constraints.

The basic steps for establishing an MP model are:

1. develop a narrative of the behavior,
2. identify MP events,
3. identify coordination, and
4. define constraints.

For more background about MP as well as a full description of each of these steps, see Appendix D of the interim technical report entitled "Verification and Validation (V&V) of System Behavior Specifications" created for the Systems Engineering Research Center (SERC), dated 30 September, 2017 [21], available from www.sercuarc.org.

5.2 Introduction to MP-Python

MP-Python enables automated trace generation and visualization of MP code models written using the MP event grammar. The MP-Firebird academic implementation provides this functionality, but is hosted on a public server (firebird.nps.edu). MP-Python is under development as an installable distribution for local modeling. MP-Python includes the following components:

- **Python User Interface**

The user runs the Monterey Phoenix Python User Interface application to generate and display MP code traces.

- **Trace Generator**

MP-Python internally uses the trace generator to compile MP code into a data format that the User Interface can display.

5.2.1 Using MP-Python

This section provides an overview of MP-Python and how it was used to generate and test the UAV behaviors.

Main Window

Figure 5.1 shows an example screenshot of the MP-Python GUI window with the UAV Ingress case study loaded and compiled at scope 1. The figure shows the general process for writing and running an MP model: 1), use the text editor for typing or loading an MP code model, 2) use the tool bar to select a scope and run the model, and 3) browse and inspect the generated event traces.

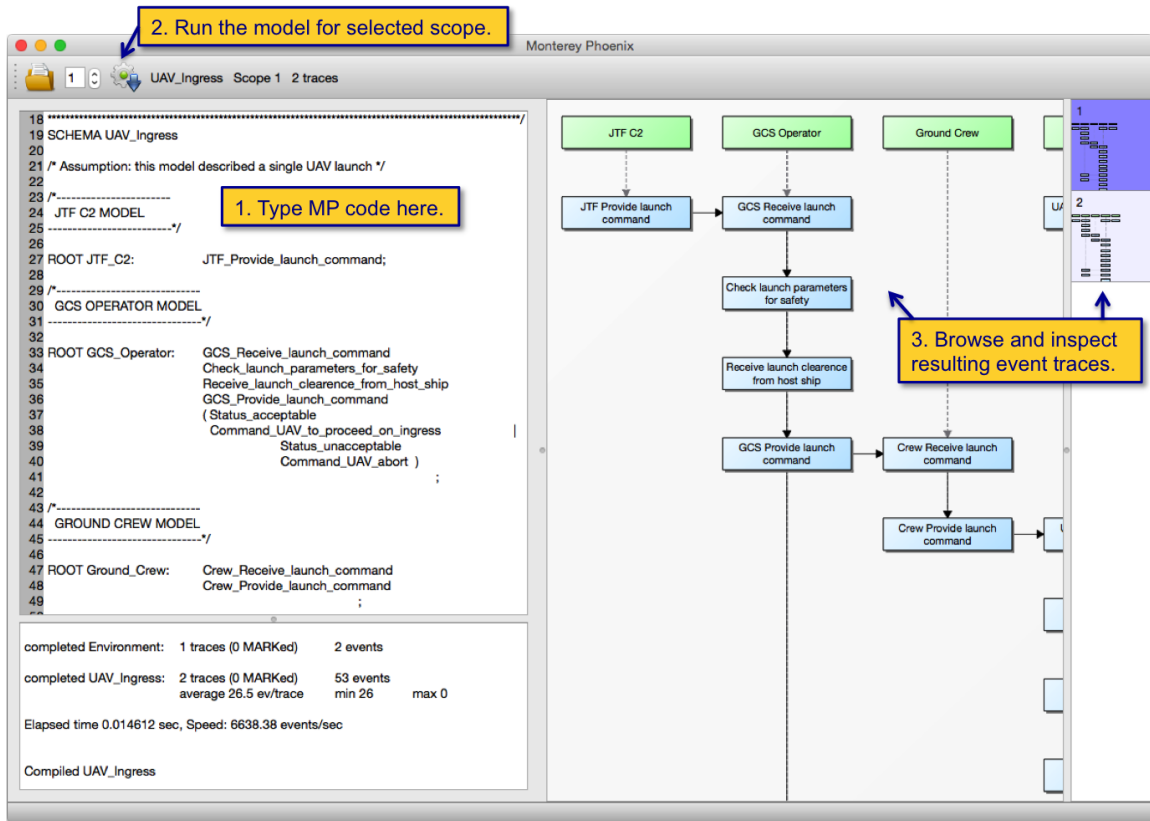


Figure 5.1. MP-Python Main Window.

View areas are similar to those in MP-Firebird:

- Upper left text editor pane contains the MP code.
- Lower left console pane contains trace generator log information.
- Center graph pane contains the currently selected event trace graph.
- Right navigation pane contains a complete thumbnail list of traces as specified for the given schema at the given scope.

Toolbar

The toolbar provides quick controls and an indication of the currently active schema and scope. From left to right in Figure 5.1, these controls are:

- The folder icon, which opens an MP code file from a file directory.
- The scope control, which is used to select the scope at which to run the model.
- The gear icon, which runs the event trace generator at the selected scope level.

- Status text, in this example reading UAV_Ingress Scope 1 2 traces to indicate that the view shows the UAV_Ingress schema built with a scope of one and that two traces were generated.

Menus

Menu controls above the toolbar provide additional capability. Figure 5.2 shows the top menu selections, with the File menu expanded to show the available pre-loaded examples.

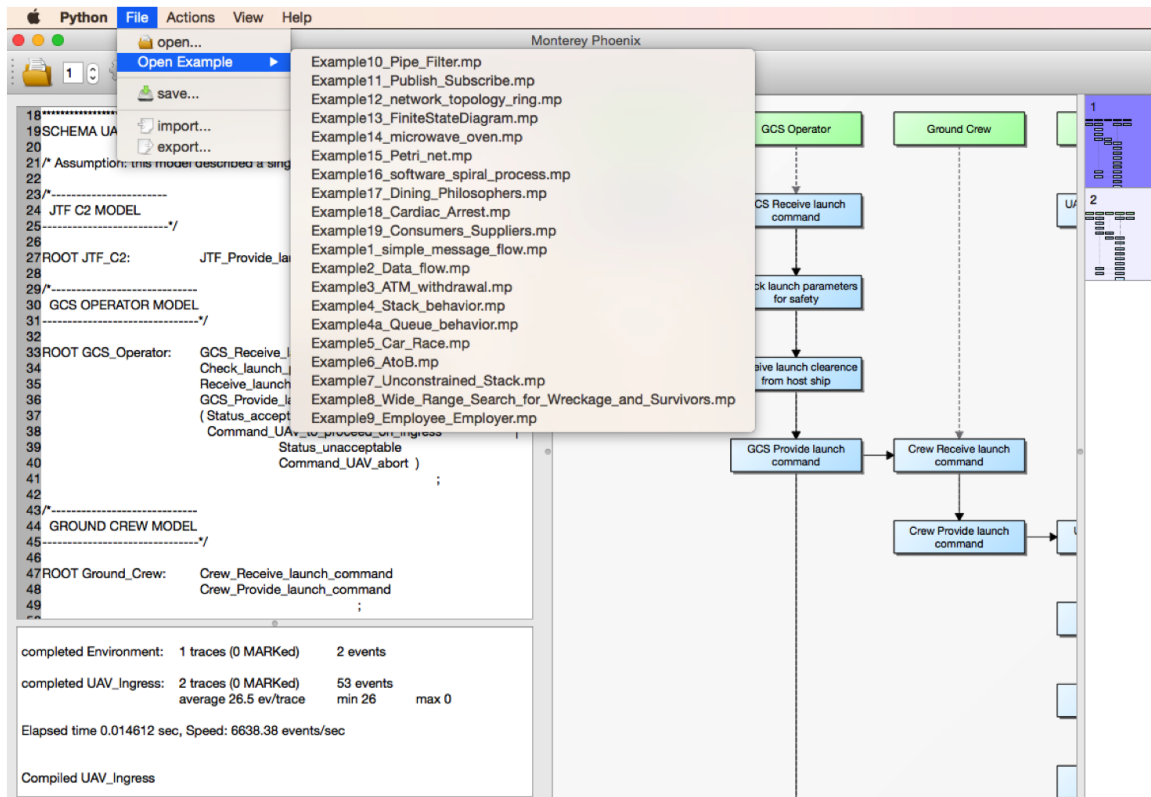


Figure 5.2. MP-Python File Menu, expanded to show pre-loaded examples.

- Under File, a user may open MP code from a folder, open MP code from an example, or import/export MP code and graphs in JSON Graph Format (JGF).
- Under Actions, a user may run the code generator or clear the MP code log.
- Under View, a user may choose a standard or customized theme for boxes, arrows and background colors used in the graphs.
- Under Help, a user may obtain help on using MP-Python and obtain information about MP-Python.

Status Bar

A status bar at the very bottom indicates status from the latest action or information about a control. For example, immediately after running this model, the status bar will indicate that the UAV_Ingress schema has just been generated:

Compiled UAV_Ingress

5.2.2 Differences between MP-Python and MP-Firebird

The main architectural difference between MP-Python and MP-Firebird is that MP-Python runs on a stand-alone system, while MP-Firebird uses a client-server architecture. MP-Python allows users to run Monterey Phoenix without a connection to a server, which may be necessary in cases ranging from model sensitivity to nonavailability of an Internet connection. MP-Firebird, on the other hand, allows users to run Monterey Phoenix without having to install any software, and so is ideal for first time users and others getting initially acquainted with Monterey Phoenix modeling. At present, the MP-Python architecture is developmentally simpler to maintain and evolve with new features.

MP-Python has some functional differences from MP-Firebird. Here are some new capabilities:

- Compiled MP code is laid out using a new MP code graph layout engine
- Annotation is added to indicate the active schema and scope
- Runtime status feedback is provided
- Performance is improved
- Import/export follows the JGF standard for multigraphs, <http://jsongraphformat.info>. By following this standard, MP graphs are compatible with other visualization tools and programs that process multigraph data. This standard defines names for graph, node, and edge objects and their fields, and include content to define user-specific fields such as node coordinates and MP-specific markings.

Here are some MP-Firebird capabilities that were not in MP-Python at the time of this writing, but planned for near term follow on work:

- Support for Windows systems
- Export .png images of traces or cropped areas
- Syntax highlighting in the text editor pane
- Node hiding/collapsing
- Node and node group selection and move
- Help and About documentation

5.3 HADR scenario UAV behaviors modeled using MP-Python

This section examines four HADR case studies:

- UAV_Phase1_MA.mp, UAV_Ingress
- UAV_Phase2_MA.mp, UAV_On_station
- UAV_Phase3_MA.mp, UAV_Egress
- UAV_Phase4_MA.mp, UAV_Postflight_v5_CMR_170611

Note that any timing values reported were performed using MP-Python on a 4-Core Intel i5-2400 CPU running at 3.10GHz. The generated scenarios in MP-Python are logically identical to the those that were produced from the same model on MP-Firebird, however, all MP-Python traces are laid out using a new and improved layout algorithm.

5.3.1 Ingress Phase

The Ingress phase of HADR behavior is shown in Listing 5.1.

Listing 5.1: MP code file UAV_Phase1_MA.mp showing HADR UAV Ingress

```
/* *****
UAV HADR Mission

The following model specifies the ingress phase of the UAS HADR
reference mission. The ingress phase is preceded by the preflight
phase, and followed by the on-station phase.

created by K. Giammarco on 04-26-2017 (Template established)
modified by ... on ... (Brief description of changes)
modified by C. Reese on 170610... added branch in the GCS operator
model - status acceptable/unacceptable
modified by C. Reese on 170611... Added Environment provide / UAV
receive navigation reference

modified by M. Auguston, 9/5/17

***** */

SCHEMA UAV_Ingress

/* Assumption: this model described a single UAV launch */

/*-----
JTF C2 MODEL
```

```

-----*/

ROOT JTF_C2: JTF_Provide_launch_command;

/*-----
   GCS_OPERATOR MODEL
-----*/

ROOT GCS_Operator: GCS_Receive_launch_command
                   Check_launch_parameters_for_safety
                   Receive_launch_clearance_from_host_ship
                   GCS_Provide_launch_command
                   (
                     Status_acceptable
                     Command_UAV_to_proceed_on_ingress |
                     Status_unacceptable
                     Command_UAV_abort
                   ) ;

/*-----
   GROUND CREW MODEL
-----*/

ROOT Ground_Crew: Crew_Receive_launch_command
                  Crew_Provide_launch_command
                  ;

/*-----
   UAV MODEL
-----*/

ROOT UAV: UAV_Receive_navigation_reference
          UAV_Receive_launch_command
          Launch
          Execute_climb
          Maneuver_to_clear_obstacles
          Maneuver_to_ingress_altitude
          Level_off_at_ingress_altitude
          Transmit_status_and_position

          (
            Receive_command_to_proceed
            Follow_flight_path_to_reach_onstation_area_and_altitude
            Reach_onstation_waypoint |
            Receive_command_to_abort
            Abort_mission
          ) ;

/*-----

```

```

NAV REFERENCE MODEL
-----*/

ROOT Environment: Provide_navigation_reference ;

/*-----
COORDINATION
-----*/

COORDINATE $a: JTF_Provide_launch_command FROM JTF_C2 ,
             $b: GCS_Receive_launch_command FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Provide_navigation_reference FROM Environment ,
             $b: UAV_Receive_navigation_reference FROM UAV
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: GCS_Provide_launch_command FROM GCS_Operator ,
             $b: Crew_Receive_launch_command FROM Ground_Crew
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Crew_Provide_launch_command FROM Ground_Crew ,
             $b: UAV_Receive_launch_command FROM UAV
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Transmit_status_and_position FROM UAV,
             $b: ( Status_acceptable |
                   Status_unacceptable ) FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Command_UAV_to_proceed_on_ingress FROM GCS_Operator ,
             $b: Receive_command_to_proceed FROM UAV
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Command_UAV_abort FROM GCS_Operator ,
             $b: Receive_command_to_abort FROM UAV
DO ADD $a PRECEDES $b; OD;

```

This model results in two traces, no matter what the scope setting is because there is no iteration to respond to scope increases. The trace shown in Figure 5.3 resolves in a “Proceed” condition. The trace shown in Figure 5.4 follows the same general flow until near the end, where it resolves in an “Abort” condition after the GCS operator determines status to be unacceptable.

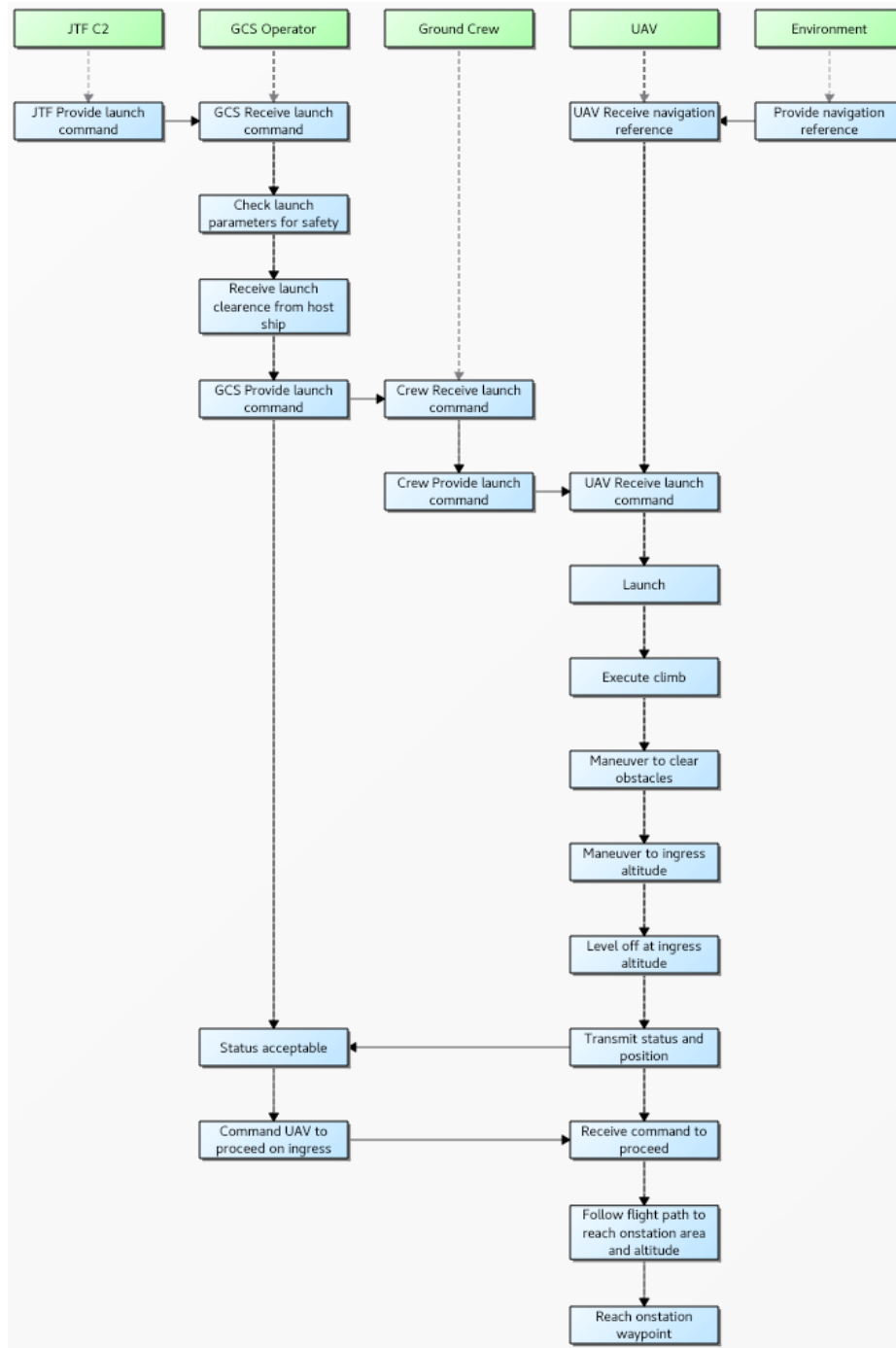


Figure 5.3. UAV Ingress, Scope 1 Trace 1, Proceed.

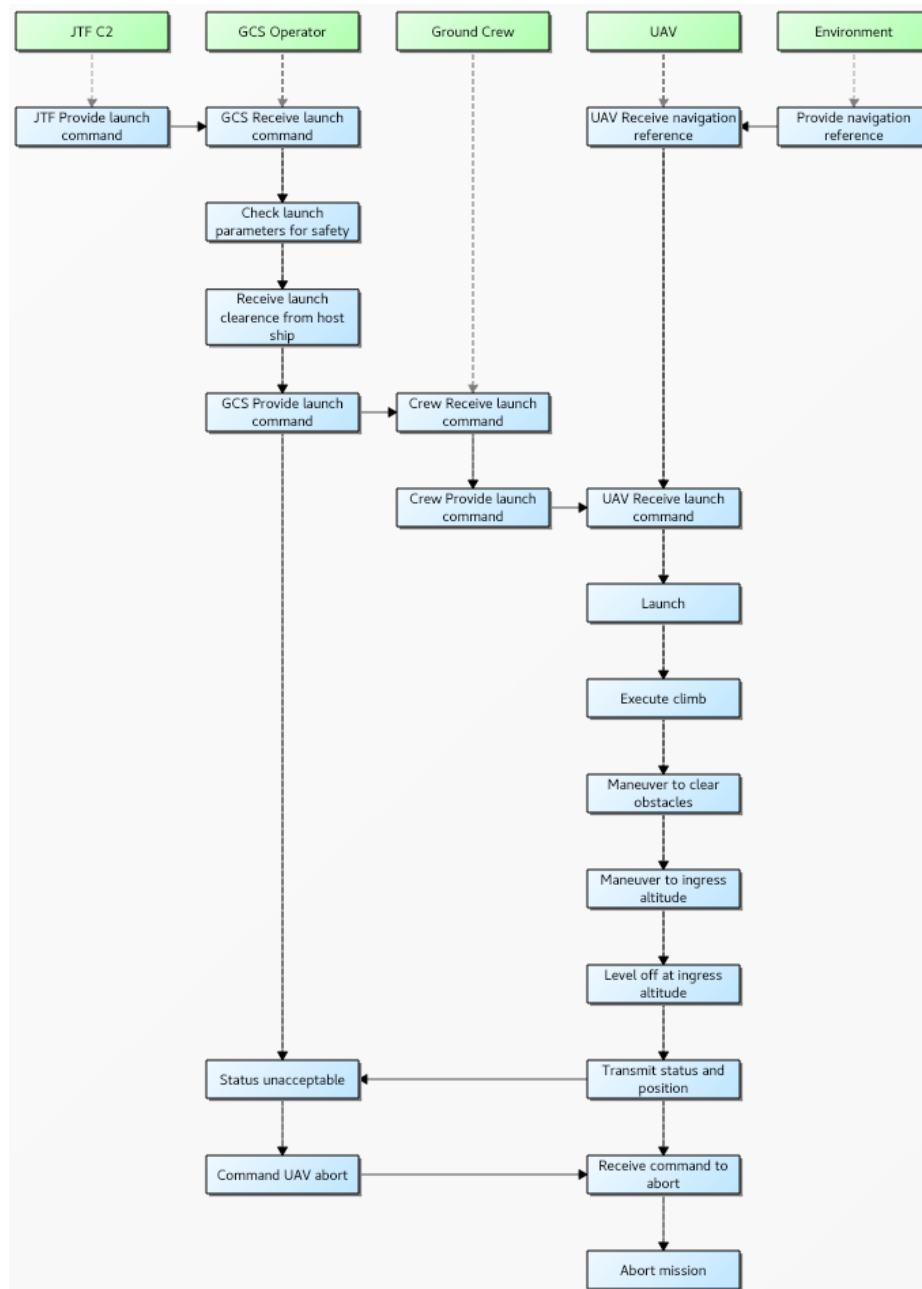


Figure 5.4. UAV Ingress, Scope 1 Trace 2, Abort.

5.3.2 On Station Phase

The On-Station phase of HADR scenario UAV behavior is shown in Listing 5.2.

Listing 5.2: MP code file UAV_Phase2_MA.mp showing HADR UAV On-Station

```

/*****
UAV HADR Mission

The following model specifies the on-station phase of the UAS HADR
reference mission. The on-station phase is preceded by the ingress
phase, and followed by the egress phase.

created by K. Giammarco on 04-26-2017          (Template established)
modified by M. Auguston, 9/13/17

run for scope 2 or up
*****/

SCHEMA UAV_On_station

/*-----
MILITARY ASSETS MODEL
-----*/

ROOT Military_Assets: Receive_report;

/*-----
GCS OPERATOR MODEL
-----*/

ROOT GCS_Operator: Verify_correct_UAV_position
                  Command_on_station_tasking
                  Report_commencement_of_on_station_tasking

                  (*
                    Receive_target_data
                    Analyze_target_data

                    (
                      Command_SAR_mission
                      Confirm_SAR_mission_complete |
                      Command_continue_on_station_tasking
                    )
                  *)

                  Command_egress;

/*-----
UAV MODEL
-----*/

```

```

ROOT UAV: Report_on_station_waypoint_arrival
          Conduct_on_station_tasking

          (*
            ( Identify_target
              Transmit_target_data
              (
                Conduct_SAR_mission
                Report_SAR_mission_complete |
                Continue_on_station_tasking
              ) |
              Fail_to_identify_target
            )
          *)

          (
            Bingo_fuel |
            On_station_time_complete |
            Receive_command_egress
          )
          Proceed_to_egress;

/*-----
  COORDINATION BETWEEN UAV and GCS OPERATOR
-----*/

COORDINATE $a: Report_on_station_waypoint_arrival FROM UAV,
               $b: Verify_correct_UAV_position FROM GCS_Operator
               DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Command_on_station_tasking FROM GCS_Operator ,
               $b: Conduct_on_station_tasking FROM UAV
               DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Report_commencement_of_on_station_tasking FROM
               GCS_Operator ,
               $b: Receive_report FROM Military_Assets
               DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Transmit_target_data FROM UAV,
               $b: Receive_target_data FROM GCS_Operator
               DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Command_SAR_mission FROM GCS_Operator ,
               $b: Conduct_SAR_mission FROM UAV
               DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Report_SAR_mission_complete FROM UAV,
               $b: Confirm_SAR_mission_complete FROM GCS_Operator

```

```

DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Command_continue_on_station_tasking FROM GCS_Operator ,
               $b: Continue_on_station_tasking FROM UAV
DO ADD $a PRECEDES $b; OD;

IF #Receive_command_egress > 0 THEN
    /* this coordination is performed only when Receive_command_egress
       is present */
    COORDINATE $a: Command_egress FROM GCS_Operator ,
               $b: Receive_command_egress FROM UAV
    DO ADD $a PRECEDES $b; OD;
FI;

/*-----
ENVIRONMENT MODEL
-----*/

ROOT Environment: (*
                   Target_appears
                   *) ;

COORDINATE $a: Target_appears ,
               $b: (Identify_target | Fail_to_identify_target)
DO ADD $a PRECEDES $b; OD;

/* mark traces with more than one Target */
IF #Target_appears > 1 THEN MARK; FI;

```

At scope 1, this model results in 12 traces, calculated in 0.0277 seconds. Example trace 7 is shown in Figure 5.5.

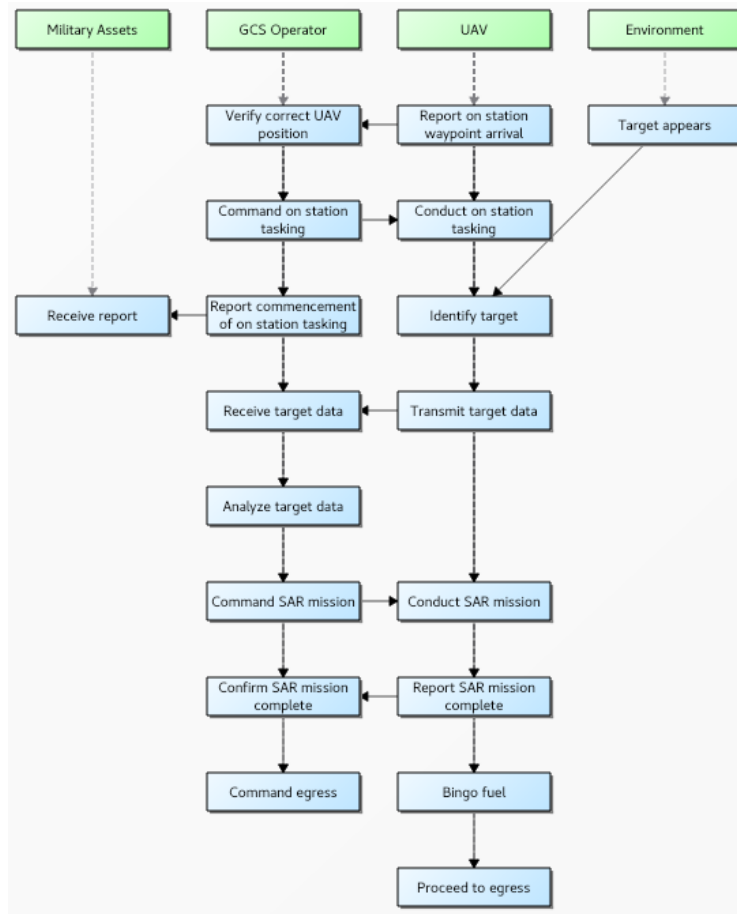


Figure 5.5. UAV On Station, Scope 1 Trace 7.

At scope 2, this model generates 39 traces, calculated in 0.354 seconds. More traces are generated at scope 2 than at scope 1 since there is some coordinated iteration in the GCS Operator and UAV roots on target identification. 26 of the 39 traces are marked as containing more than one target. Example trace 33 is shown in Figure 5.6, in which the identification of the first target provokes a SAR mission and the identification of the second target does not.

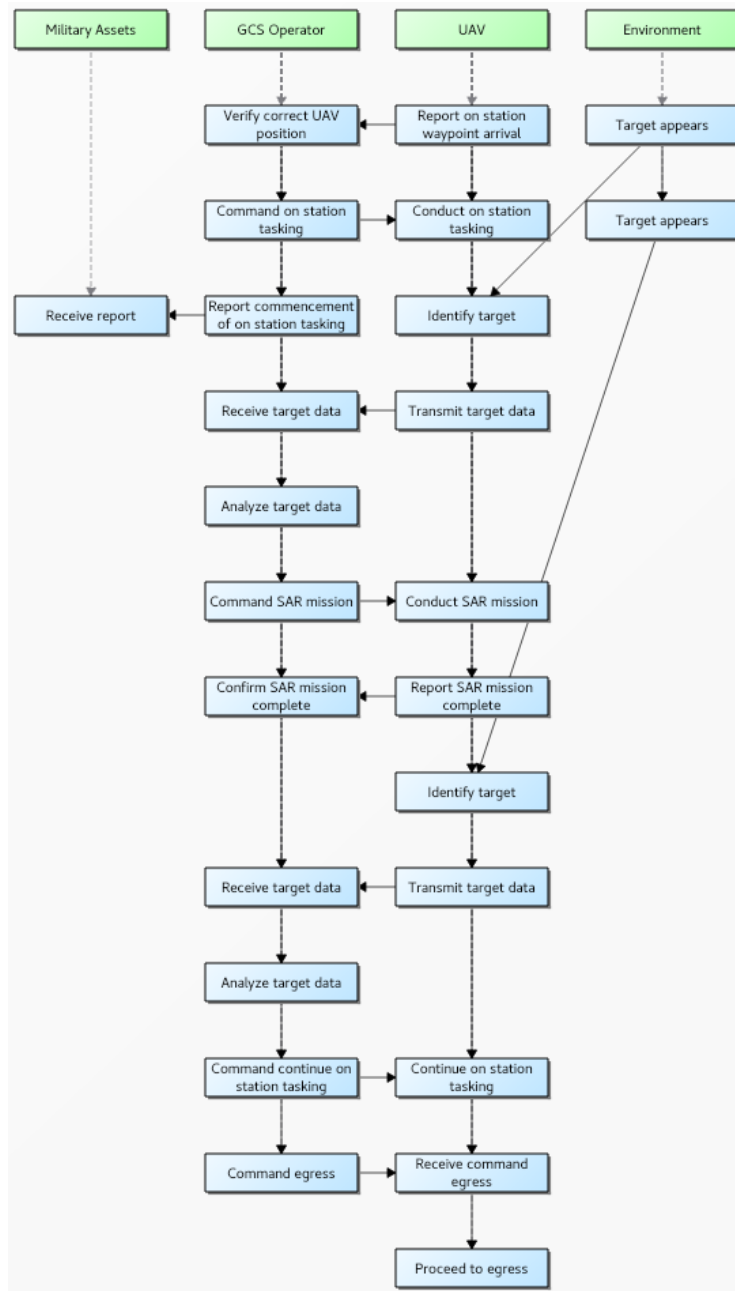


Figure 5.6. UAV On Station, Scope 2 Trace 33.

Running the model at scope 3 generates 120 traces in 3.35 seconds, and 363 traces at scope 4 in 31.0 seconds. Figure 5.7 illustrates a scenario in which four targets appear in the environment, but none result in the initiation of a SAR mission.

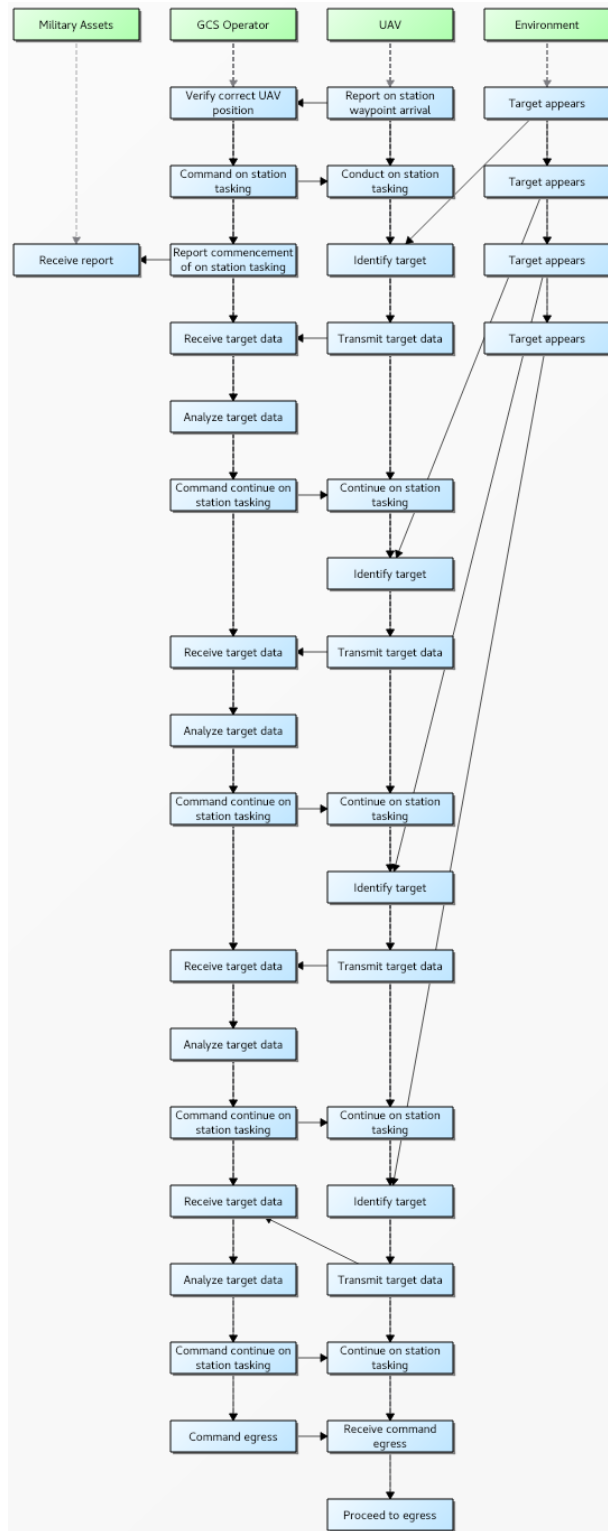


Figure 5.7. UAV On Station, Scope 4 Trace 363, illustrating four targets.

5.3.3 Egress Phase

The Egress phase of HADR UAV behavior is shown in Listing 5.3.

Listing 5.3: MP code file UAV_Phase3_MA.mp showing HADR UAV Egress

```
/******  
  
UAV HADR Mission  
  
The following model specifies the egress phase of the UAS HADR  
reference mission. The egress phase is preceded by the on-station  
phase, and followed by the postflight phase.  
  
created by K. Giammarco on 04-26-2017          (Template established)  
modified by M. Auguston, 9/19/17  
  
run for scope 1 (99 traces in approx. 2 sec.),  
               or 2 (1235 traces in approx. 138 sec.)  
*****  
  
SCHEMA UAV_Egress  
  
/*-----  
GCS OPERATOR MODEL  
-----*/  
  
ROOT GCS_Operator: /* the following events may happen concurrently */  
{  
    Command_UAV_to_commence_Egress,  
    Notify_ground_crew  
}  
  
Monitor_system_health_and_flight_path  
  
Check_recovery_parameters  
    (+  
        Receive_flight_parameters  
        (  
            Execute_Autoland |  
            Execute_manual_landing  
        )  
    )  
  
    Monitor_recovery_profile  
  
    [  
        Safety_Parameters_Exceeded  
        (  
            Receive_confirmation_of_autonomous_waveoff |  
            UAV_Still_Approaching  
            Send_manual_waveoff_command
```

```

        Confirm_waveoff
    )
    Command_UAV_to_return_to_recovery_waypoint
]
+)

[
    Approaching_Bingo_Fuel
    (
        Alternate_Landing_Site_Available
        Command_UAV_to_alternate_landing_site |
        Alternate_Landing_Site_Not_Available
        Ditch_aircraft
    )
]

[
    Recover_aircraft
];

/*-----
   UAV MODEL
-----*/

```

```

ROOT UAV:
    Commence_Egress
    Report_system_health_and_flight_path
    Receive_navigation_reference
    (+
        Report_arrival_at_recovery_waypoint

        (
            Receive_Autoland_command |
            Receive_manual_commands
        )

        Execute_recovery_profile

        [
            Safety_Parameters_Exceeded
            (
                Execute_autonomous_waveoff |

                UAV_Still_Approaching
                Execute_manual_waveoff
            )

            Maneuver_to_intercept_recovery_waypoint ]
    +)

```

```

[
    Approaching_Bingo_Fuel
    [
        Maneuver_to_alternate_landing_site
    ]
]

Execute_recovery
(
    Land_safely |
    Controlled_crash |
    Uncontrolled_crash
) ;

/*-----
COORDINATION BETWEEN UAV and GCS OPERATOR
-----*/

ENSURE #Ditch_aircraft == #Controlled_crash;

GCS_Operator, UAV SHARE ALL Safety_Parameters_Exceeded,
                        UAV_Still_Approaching,
                        Approaching_Bingo_Fuel;

COORDINATE $a: Command_UAV_to_commence_Egress FROM GCS_Operator,
            $b: Commence_Egress FROM UAV
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Report_system_health_and_flight_path FROM UAV,
            $b: Monitor_system_health_and_flight_path FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;

COORDINATE $a: Report_arrival_at_recovery_waypoint FROM UAV,
            $b: Receive_flight_parameters FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;

IF #Execute_Autoland > 0 THEN
    COORDINATE $a: Execute_Autoland FROM GCS_Operator,
                $b: Receive_Autoland_command FROM UAV
    DO ADD $a PRECEDES $b; OD;
FI;

IF #Execute_manual_landing > 0 THEN
    COORDINATE $a: Execute_manual_landing FROM UAV,
                $b: Receive_manual_commands FROM GCS_Operator
    DO ADD $a PRECEDES $b; OD;
FI;

```

```

COORDINATE $a: Execute_recovery_profile FROM UAV,
             $b: Monitor_recovery_profile FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;

IF #Execute_autonomous_waveoff > 0 THEN
    COORDINATE $a: Execute_autonomous_waveoff FROM UAV,
                $b: Receive_confirmation_of_autonomous_waveoff FROM
                    GCS_Operator
    DO ADD $a PRECEDES $b; OD;
FI;

IF #Send_manual_waveoff_command > 0 THEN
COORDINATE $a: Send_manual_waveoff_command FROM GCS_Operator ,
             $b: Execute_manual_waveoff FROM UAV,
             $c: Confirm_waveoff FROM GCS_Operator
DO ADD $a PRECEDES $b;
    ADD $b PRECEDES $c;
OD;
FI;

IF #Command_UAV_to_return_to_recovery_waypoint > 0 THEN
COORDINATE $a: Command_UAV_to_return_to_recovery_waypoint FROM
    GCS_Operator ,
             $b: Maneuver_to_intercept_recovery_waypoint FROM UAV
DO ADD $a PRECEDES $b; OD;
FI;

IF #Command_UAV_to_alternate_landing_site > 0 THEN
COORDINATE $a: Command_UAV_to_alternate_landing_site FROM GCS_Operator ,
             $b: Maneuver_to_alternate_landing_site FROM UAV
DO ADD $a PRECEDES $b; OD;
FI;

IF #Controlled_crash > 0 THEN
COORDINATE $a: Ditch_aircraft FROM GCS_Operator ,
             $b: Controlled_crash FROM UAV
DO ADD $a PRECEDES $b; OD;
FI;

IF #Land_safely > 0 THEN
COORDINATE $a: Land_safely FROM UAV,
             $b: Recover_aircraft FROM GCS_Operator
DO ADD $a PRECEDES $b; OD;
FI;

/*-----
   GROUND CREW MODEL
-----*/

```

```

ROOT Ground_Crew: Prepare_recovery_device ;

COORDINATE $a: Notify_ground_crew FROM GCS_Operator ,
             $b: Prepare_recovery_device FROM Ground_Crew
DO ADD $a PRECEDES $b; OD;

/*-----
   NAV REFERENCE MODEL
-----*/

ROOT Navigation_Reference: Send_navigation_reference ;

COORDINATE $a: Send_navigation_reference FROM Navigation_Reference ,
             $b: Receive_navigation_reference FROM UAV
DO ADD $a PRECEDES $b; OD;

/*=====*/
IF #Land_safely > 0 THEN MARK; SAY("Safe_landing"); FI;

```

Since there is no iteration, this model results in 90 traces at any scope, calculated in 1.53 seconds. Example trace 81, shown in Figure 5.8, shows complications but a safe landing, as indicated by the yellow “Safe landing” label.

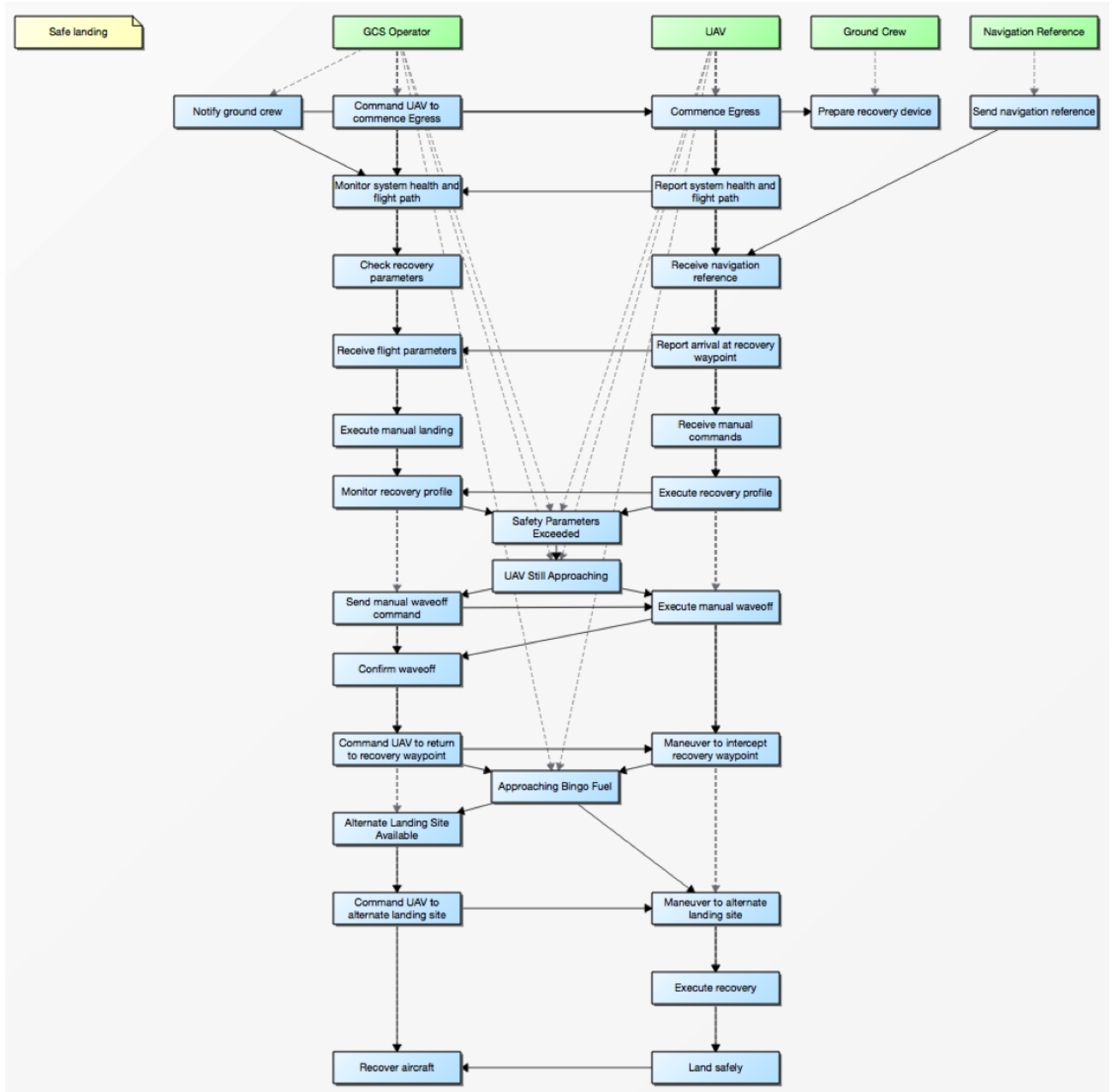


Figure 5.8. UAV Egress, Scope 1 Trace 81.

5.3.4 Postflight Phase

The Postflight phase of HADR UAV behavior is shown in Listing 5.4.

Listing 5.4: MP code file UAV_Phase4_MA.mp showing HADR UAV Post-flight

```

/*****
UAV HADR Mission

The following model specifies the postflight phase of the UAS HADR
reference mission. The postflight phase is preceded by the egress
phase, and is the last phase of the mission.

created by K. Giammarco on 04-26-2017          (Template established)
modified by C. Reese on (Cody's first model update based on latest
mission narrative)
updated by C. Reese on 170610 - re-order assets to change appearance
of output graph
updated by C. Reese on 170611 - Added JTF_C2 as an explicit actor to
receive AAR
updated by M. Auguston, 9/5/17

*****/

SCHEMA UAV_Postflight_v5_CMR_170611
/*-----
OTHER ASSETS MODEL - NOTE: INCLUDES JCF C2, HN, NGO, OTHERS
-----*/

ROOT Other_Assets: Receive_after_action_report;

ROOT JTF_C2: Receive_after_action_report;

ROOT GCS: Release_data_storage_devices
Shutdown;

ROOT GCS_Operator: Remove_data_storage_devices
Hand_off_data_storage_devices
Issue_shutdown_command
Debrief_mission
Generate_after_action_report
Transmit_after_action_report;

ROOT Ground_Crew: Assess_UAV_for_damage
Assess_launcher_for_damage
Assess_recoverer_for_damage
Complete_inspection_report
Remove_data_storage_devices
Hand_off_data_storage_devices
Issue_shutdown_command
Debrief_mission
Generate_after_action_report

```

```

Transmit_after_action_report;

ROOT UAV:  Undergo_postflight_inspection
          Release_data_storage_devices
          Shutdown;

ROOT Launcher:  Undergo_postflight_inspection
                Shutdown;

ROOT Recoverer:  Undergo_postflight_inspection
                 Shutdown;

/*-----
  COORDINATION BETWEEN UAS and GROUND CREW
-----*/

COORDINATE $a: Assess_UAV_for_damage FROM Ground_Crew ,
             $b: Undergo_postflight_inspection FROM UAV,
             $c: Complete_inspection_report FROM Ground_Crew
DO  ADD $a PRECEDES $b;
    ADD $b PRECEDES $c;
OD;

COORDINATE $a: Remove_data_storage_devices FROM Ground_Crew ,
             $b: Release_data_storage_devices FROM UAV,
             $c: Hand_off_data_storage_devices FROM Ground_Crew ,
             $d: Issue_shutdown_command FROM Ground_Crew ,
             $e: Shutdown FROM UAV
DO  ADD $a PRECEDES $b;
    ADD $b PRECEDES $c;
    ADD $c PRECEDES $d;
    ADD $d PRECEDES $e;
OD;

COORDINATE $a: Assess_launcher_for_damage FROM Ground_Crew ,
             $b: Undergo_postflight_inspection FROM Launcher ,
             $c: Complete_inspection_report FROM Ground_Crew ,
             $d: Issue_shutdown_command FROM Ground_Crew ,
             $e: Shutdown FROM Launcher
DO  ADD $a PRECEDES $b;
    ADD $b PRECEDES $c;
    ADD $c PRECEDES $d;
    ADD $d PRECEDES $e;
OD;

COORDINATE $a: Assess_recoverer_for_damage FROM Ground_Crew ,
             $b: Undergo_postflight_inspection FROM Recoverer ,
             $c: Complete_inspection_report FROM Ground_Crew ,
             $d: Issue_shutdown_command FROM Ground_Crew ,

```

```

    $e: Shutdown FROM Recoverer
DO   ADD $a PRECEDES $b;
      ADD $b PRECEDES $c;
      ADD $c PRECEDES $d;
      ADD $d PRECEDES $e;
OD;

/*-----*/
COORDINATION BETWEEN GCS OPERATOR and GCS
/*-----*/
COORDINATE $a: Remove_data_storage_devices FROM GCS_Operator ,
            $b: Release_data_storage_devices FROM GCS,
            $c: Hand_off_data_storage_devices FROM GCS_Operator ,
            $d: Issue_shutdown_command FROM GCS_Operator ,
            $e: Shutdown FROM GCS
DO   ADD $a PRECEDES $b;
      ADD $b PRECEDES $c;
      ADD $c PRECEDES $d;
      ADD $d PRECEDES $e;
OD;

/*-----*/
COORDINATION BETWEEN GROUND CREW and GCS OPERATOR
/*-----*/
GCS_Operator , Ground_Crew SHARE ALL Debrief_mission ,
Generate_after_action_report ,
Transmit_after_action_report;

COORDINATE $a: Transmit_after_action_report FROM GCS_Operator ,
            $b: Receive_after_action_report FROM Other_Assets
DO   ADD $a PRECEDES $b;
OD;

COORDINATE $a: Transmit_after_action_report FROM GCS_Operator ,
            $b: Receive_after_action_report FROM JTF_C2
DO   ADD $a PRECEDES $b;
OD;

/*=====*/
GCS, UAV, Launcher , Recoverer SHARE ALL Shutdown;

COORDINATE $a1: Transmit_after_action_report FROM Ground_Crew ,
            $b1: Receive_after_action_report FROM Other_Assets ,
            $b2: Receive_after_action_report FROM JTF_C2
DO
      ADD $a1 PRECEDES $b1;
      ADD $a1 PRECEDES $b2;
OD;

```

At any scope, this model results in one trace, calculated in 0.0143 seconds, since there are no alternate or optional flows contained in this model. This trace is shown in Figure 5.9.

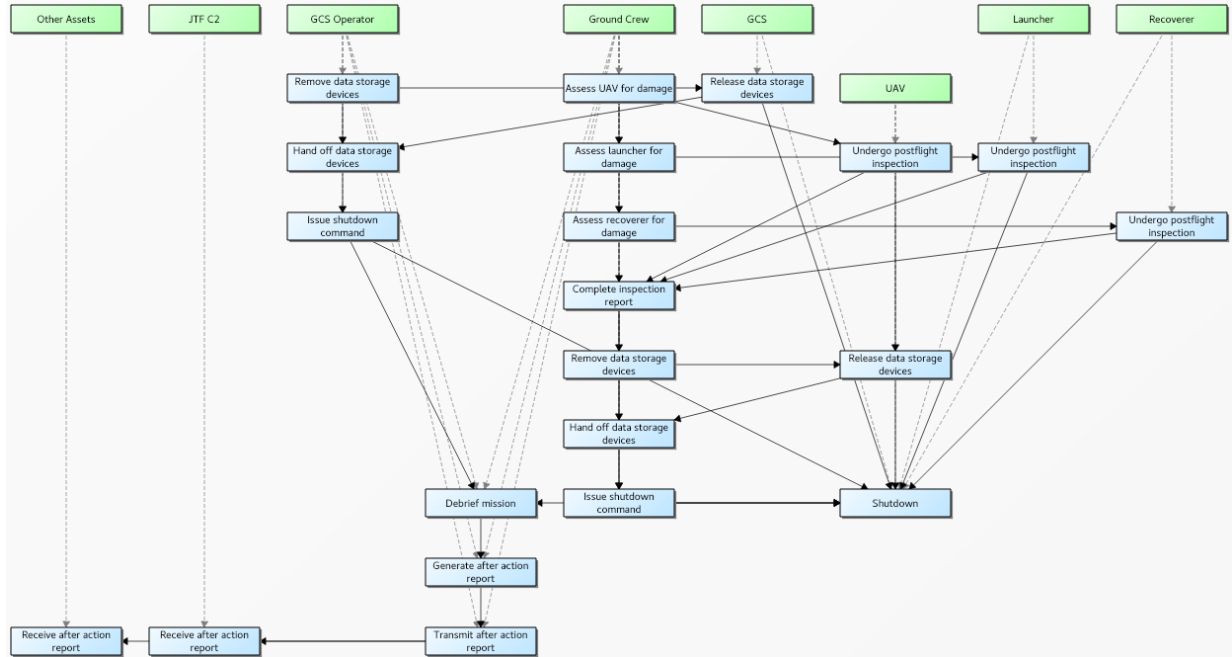


Figure 5.9. UAV Postflight, Scope 1 Trace 1.

As a comparison, Figure 5.10 is included, showing HADR UAV Postflight using the MP-Firebird version 2 tool.

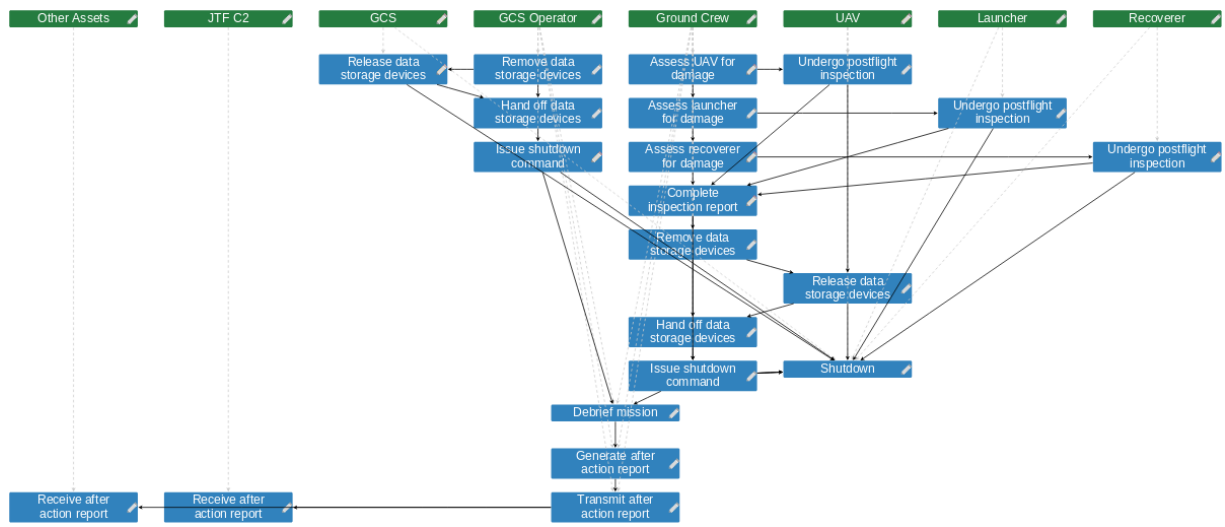


Figure 5.10. UAV Postflight, Scope 1 Trace 1, MP-Firebird version 2 view, for comparison.

5.4 Relationship of MP Events to Generic Model Classes

Classes from the Generic Model UPIA, LML, or other modeling languages related to system behaviors pertain to MP event traces.

For example, the root events in the MP models presented in this chapter can be typed as the UPIA CapabilityRole class, or as the LML Asset class. Composite and atomic events in these MP models can be typed as the UPIA OperationalTask class, or as the LML Action class. Pattern illustrations for these concepts are shown in Figures 5.11 and 5.12, to bridge MP to these conceptual languages.

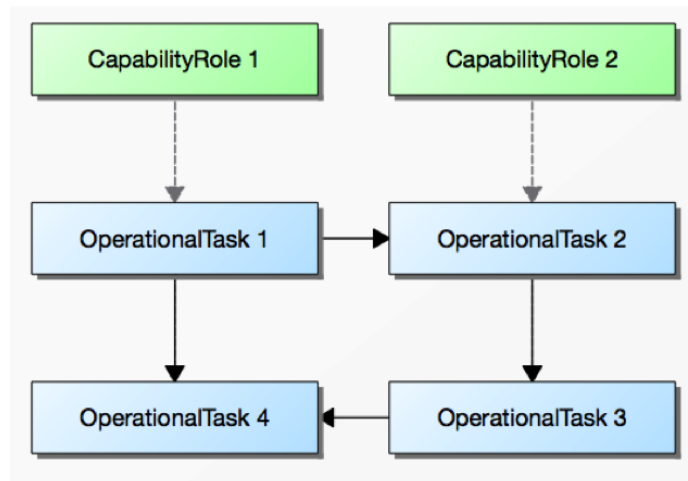


Figure 5.11. Root events in MP typically correspond to components of a system, systems, or the environment, and so can be typed as UPIA capability roles. High level composite and atomic events may be typed as UPIA operational tasks.

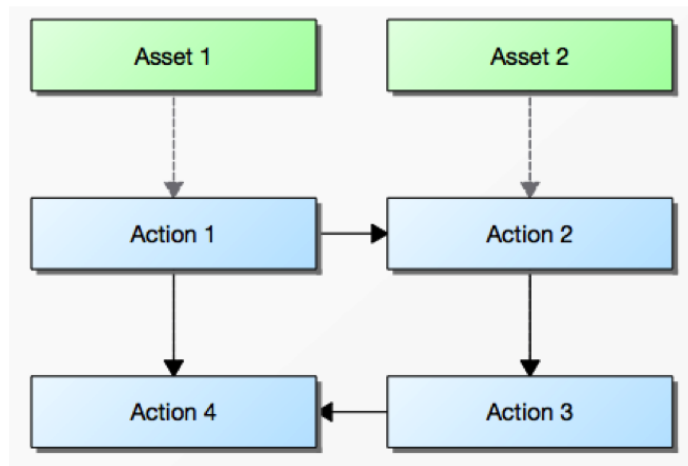


Figure 5.12. Root events can alternatively be typed as LML assets. Composite and atomic events may be typed as LML actions.

CHAPTER 6:

Conclusions and Proposed Way Ahead

Kristin Giammarco

At the beginning of this effort, we set out to accomplish the following research objectives:

1. Assess the reproducibility of the RSA-native Generic Model in other tools, specifically in Innoslate and Monterey Phoenix
2. Produce tool-agnostic patterns and style guidance
3. Test the executability of successfully reproduced Generic Model behavior views in Innoslate and Monterey Phoenix
4. Provide tool-agnostic instructions for creating and analyzing executable activity models

This chapter summarizes the conclusions from this research as findings and recommendations with respect to these objectives. The advancements delivered from this project are intended to support NAVAIR's mission accomplishment, especially in improving and promoting the principles behind the Generic Model effort, to have uniform standards for the quality and presentation of both static and executable architecture models, and to make design errors and issues easier to detect across programs.

6.1 Reproducibility of the Generic Model in other tools

We had the objective to assess the reproducibility of the RSA-native Generic Model in other tools, specifically in Innoslate and Monterey Phoenix. We accomplished this objective as follows. The Generic Model, as a set of example architecture instances using class and relationship names from the UPIA, contained all the elements needed to formally specify and graph a metamodel, or conceptual data model. As shown in Chapter 3, we found that the Generic Model was easily modeled in the lightweight formal methods language Alloy, and that we could graph the UPIA-based conceptual data model automatically from the provided example instances. We further found that we could reproduce the example instances in the source data along with many more examples, automatically, using the Alloy Analyzer tool. Having formalized the Generic Model in Alloy, we found straightforward mappings between the UPIA-based Generic Model language and the LML used by Innoslate. The Alloy Generic Model contains relevant signature mappings to LML classes, and an Innoslate model containing the example patterns in Chapter 2 has relevant mappings of LML classes to their counterparts in UPIA. The Monterey Phoenix models in Chapter 5 are behavior models, and we found that root events typically map to the UPIA CapabilityRole class (LML Asset class), and that composite and atomic events typically map to the UPIA

OperationalTask class (LML Action class). Since MP’s capability focuses on generating behavior scenarios, it follows that the behavior-related classes from UPIA and LML carry over to the MP models.

6.2 Tool-agnostic patterns and style guidance

We had the objective to produce tool-agnostic patterns and style guidance. We accomplished this objective as follows. We created, updated and translated LML architecture heuristics and analytics into three other conceptual data model languages: DM2, UPIA, and SDL. The expressions in Chapter 2 show the conceptual data mappings across these four languages (LML, DM2, UPIA, and SDL), and by extension, tools that implement any of these languages. Furthermore, we present additional examples of well-formed architecture models as patterns, which formally capture best modeling practices of experienced architects. The patterns in Chapter 2 are ready for transition to extend the Generic Model and related successor specifications.

6.3 Executability of behavior views in Innoslate and Monterey Phoenix

We had the objective to test the executability of “successfully reproduced” Generic Model behavior views in Innoslate and Monterey Phoenix. The interpretation of this objective changed as we learned more about the intent of the Generic Model, to provide examples of well-formed views based on a UPIA schema. We pursued this objective by testing executable models mapped to Generic Model concepts in Innoslate and Monterey Phoenix. The Ingress phase of the Innoslate UAV model was discussed in Chapter 2, notional (generic) patterns for executable models in Innoslate were discussed in Chapter 4, and all four MP UAV model phases (Ingress, OnStation, Egress, and Postflight) were discussed in Chapter 5. The models were executable in simulation, each providing a timeline of events that could, in future work, be configured with durations and other attributes.

6.4 Tool-agnostic instructions for creating and analyzing executable activity models

We had the objective to provide tool-agnostic instructions for creating and analyzing executable activity models. We accomplished this objective in Chapter 4, with instructional content about the executable logic constructs available in LML and eight templates for executable action diagram structures. These templates have some noteworthy strengths and limitations. A major strength of the templates is that their diagram structures enable

expression of alternate behaviors in multiple systems or different parts of the same system, and in the environment. The triggering input/outputs are used to create dependencies between actions on different branches. They show how to expand behavior descriptions beyond simple system behavior, to include system of systems behaviors and dependencies. We noted a limitation when using “or” and “sync” action pairs inside loops. The “sync” action depends on the timing of its preceding activities, so if nested inside a loop, caution must be used in applying durations to the sync branch actions such that they keep pace with the triggers being sent to them from the corresponding “or” action. Otherwise, a sync branch from a previous loop iteration can be still running during a current loop iteration. The templates offer practitioners a starting point for modeling scenario variants involving system of systems interactions, concurrent / alternate / iterative behaviors, and dependencies on probabilities and resource availability, making them useful for capturing descriptions of both nominal and off-nominal scenarios variants in the same model.

6.5 Recommendations for Further Research

Since the inception of this project, NAVAIR has made some implementation decisions pertaining to languages and tools. We recommend that follow-on work be conducted to translate the work herein for SysML, and test the translations operationally in other tools that are SysML-focused. Translating the heuristics and analytics into the preferred conceptual data modeling language, as well as generating templates for static and executable architecture models in the tools used by the organization, should be among the tasks of future work. The templates have many common logic constructs found in other languages, making them transferable, reusable, and valuable beyond the languages and tools in which they were developed. MP’s scenario generation and behavior verification and validation capabilities should also be integrated into NAVAIR’s MBSE modeling framework to make these capabilities available to all of its users.

List of References

- [1] Lifecycle Modeling Language (LML) Specification. (2015). LML Steering Committee and others. [Online]. Available: <http://www.lifecyclemodeling.org/specification/>. Accessed Feb. 27, 2017.
- [2] *Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard for Functional Modeling Language - Syntax and Semantics for IDEF0, IEEE Std 1320.1*, New York, NY: Software Engineering Standards Committee of the IEEE Computer Society, 1998.
- [3] K. Giammarco, “Architecture model based interoperability assessment,” Ph.D. dissertation, Naval Postgraduate School, Monterey, CA, 2012.
- [4] 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.
- [5] J. C. Lew and K. M. Giammarco, “Decomposition relationship visualization for executable architecture diagrams,” *Procedia Computer Science*, vol. 28, pp. 626–634, 2014.
- [6] M. Rodano and K. Giammarco, “A formal method for evaluation of a modeled system architecture,” *Procedia Computer Science*, vol. 20, pp. 210–215, 2013.
- [7] C. A. Whitcomb, K. Giammarco, and G. Xie, “A formal method for assessing interoperability using architecture model elements and relationships,” in *Proceedings of the 9th Annual Conference on Systems Engineering Research*, Redondo Beach, CA, 2011.
- [8] K. Giammarco, “Formal methods for architecture model assessment in systems engineering,” in *Proceedings of the 8th Annual Conference on Systems Engineering Research*, Hoboken, NJ, 2010.
- [9] M. Auguston, K. Giammarco, W. C. Baldwin, M. Farah-Stapleton *et al.*, “Modeling and verifying business processes with monterey phoenix,” *Procedia Computer Science*, vol. 44, pp. 345–353, 2015.
- [10] K. Giammarco and M. Auguston, “Well, you didn’t say not to! a formal systems engineering approach to teaching an unruly architecture good behavior,” *Procedia Computer Science*, vol. 20, pp. 277–282, 2013.

- [11] M. Auguston, C. Whitcomb, and K. Giammarco, "New approach to system and software architecture specification based on behavior mode," in *INCOSE-IEEE MBSE Conference*, George Mason University, Fairfax, VA., 2010.
- [12] W. Carlson, "Formal process modeling to improve human-decision-making during test and evaluation range control," Master's thesis, Naval Postgraduate School, Monterey, CA, 2017.
- [13] *Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems*, International Organization for Standardization. ISO/IEC/IEEE 42010, 2011.
- [14] E. Rechtin and M. Maier, *The Art of Systems Architecting*. Boca Raton, USA: CRC Press, 2002.
- [15] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2010.
- [16] DCIO, "Department of defense architecture framework (dodaf) version 2.0," *DoD Deputy Chief Information Officer*. Available via: <http://cionii.defense.gov/sites/-dodaf20/> [accessed Feb. 27, 2017], 2010.
- [17] UPIA DoDAF 2.01 mapping reference. (2010). IBM Corporation. [Online]. Available: <http://www-01.ibm.com/support/docview.wss?uid=swg27019041&aid=1>. Accessed May 12, 2017.
- [18] D. Long and Z. Scott, *A Primer for Model-based Systems Engineering*. Lulu.com, 2011.
- [19] D. M. Buede, *The Engineering Design of Systems: Models and Methods*, 2nd ed. John Wiley & Sons, 2009.
- [20] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information." *Psychological Review*, vol. 63, no. 2, p. 81, 1956.
- [21] K. Giammarco, M. Blackburn, R. Carlson, and et al., "Verification and validation (V&V) of system behavior specifications," Systems Engineering Research Center (SERC), Tech. Rep., Oct. 31, 2017. Available: http://www.sercuarc.org/wp-content/uploads/2014/05/Interim-Technical-Report_RT-176.pdf
- [22] Why HSI? (2010). Georgia Institute of Technology. [Online]. Available: http://hsimed.gtri.gatech.edu/hsi_info/hsi_intro_why.php. Accessed Sept., 2012.

- [23] L. S. Committee *et al.*, “Lifecycle Modeling Language (LML) Specification, version 1,” Available via: <http://www.lifecyclemodeling.org/specification/>. [accessed Aug., 2013], 2013.
- [24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.
- [25] J. Bosch, “Design patterns as language constructs,” *Journal of Object-Oriented Programming*, vol. 11, pp. 18–32, 1998.
- [26] C. Haskins, “11.2.2 application of patterns and pattern languages to systems engineering,” *INCOSE International Symposium*, vol. 15, no. 1, pp. 1619–1627, 2005. Available: <http://dx.doi.org/10.1002/j.2334-5837.2005.tb00776.x>
- [27] R. J. Cloutier and D. Verma, “Applying the concept of patterns to systems architecture,” *Systems Engineering*, vol. 10, no. 2, pp. 138–154, 2007.
- [28] G. Rebovich, J. K. DeRosa *et al.*, “Patterns of success in systems engineering of it-intensive government systems,” *Procedia Computer Science*, vol. 8, pp. 303–308, 2012.
- [29] L. Brownsword, C. Albert, P. Place, and D. Carney, “Software acquisition patterns of failure and how to recognize them,” Carnegie-Mellon Univ., Pittsburgh, PA Software Engineering Inst., Tech. Rep., 2013.
- [30] K. Hashizume, N. Yoshioka, and E. B. Fernandez, “Three misuse patterns for cloud computing,” *Security engineering for Cloud Computing: approaches and Tools*, pp. 36–53, 2012.
- [31] T. Weilkiens, “Two kinds of patterns,” Available via: <http://model-based-systems-engineering.com/2013/02/07/two-kinds-of-patterns/>. accessed Nov., 2013], 2013.
- [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 State Univ, 2012.
- [33] D. Jackson, *Software Abstractions: Logic, language, and analysis*. MIT press, 2012.
- [34] D. Jackson, “Alloy: a language and tool for relational models,” retrieved at <http://alloy.mit.edu/alloy/>, accessed Nov. 2013, vol. 22, p. 1, 2012.
- [35] *Joint communications system*, Joint Publication 6-0, Department of Defense: Joint Chiefs of Staff, Washington, DC, 2010.

- [36] *Department of Defense dictionary of military and associated terms*, Joint Publication 1-02, Joint Chiefs of Staff, Department of Defense: Joint Chiefs of Staff, 2010, Washington, DC, as amended through October 15, 2011.
- [37] *Universal Joint Task List (UJTL)*, Chairman of the Joint Chiefs of Staff Manual 3500.04D, Department of Defense: Joint Staff, Washington, DC, 2005.
- [38] *Procedures for interoperability and supportability of information technology (IT) and national security systems (NSS)*, DoD Instruction 4630.8, Department of Defense: ASD(NII)/DoD CIO, Washington, DC, 2004.
- [39] Standards glossary. (2011). Institute of Electrical and Electronics Engineers (IEEE). [Online]. Available: http://www.ieee.org/education_careers/education/standards/standards_glossary.html#sect9. Accessed Jan. 2012.
- [40] H. Information and M. S. S. (HIMSS), *HIMSS Dictionary of Healthcare Information Technology Terms, Acronyms and Organizations*, 2nd ed. Chicago, IL: HIMSS, 2010.
- [41] DM2 conceptual data model (DIV-1) diagram. (n.d.). DoD Chief Information Officer (DoD CIO). [Online]. Available: <http://cio-nii.defense.gov/sites/dodaf20/conceptual.html>. Accessed Nan. 2012.
- [42] K. Giammarco, “Architecture modeling software analytics: Model quality and maturity assessment using automated tools,” in *System of Systems Engineering Conference (SoSE), 2017, 12th Annual System of Systems Engineering Conference*. Waikoloa, HI: IEEE, 2017, pp. 1–6.
- [43] E. Rechtin and M. Maier, *The Art of Systems Architecting*, 3rd ed. Boca Raton, USA: CRC Press, 2009.
- [44] D. A. Dunaway, “Optevfor: Organizing to accomplish the ote mission (integrated testing through mission based test design),” *ITEA Journal*, vol. 32, pp. 146–148, 2011.
- [45] G. O. Langford, *Engineering systems integration: Theory, metrics, and methods*. CRC Press, 2012.
- [46] K. Giles, “Mission-based Architecture for Swarm Composability,” Ph.D. dissertation, Dept. of Sys. Eng., NPS, Monterey, CA, 2018.
- [47] K. Giammarco and P. Shebalin, “An instructional design reference mission: Tactical disc clearance system,” Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS-SE-18-002, Mar. 2018.

- [48] J. Quartuccio and K. Giammarco, *Engineering Emergence: A Modeling and Simulation Approach*. Boca Raton, FL: CRC Press Taylor Francis Group, accepted for publication, ch. 18.

Initial Distribution List

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Research Sponsored Programs Office, Code 41
Naval Postgraduate School
Monterey, California