SYSTEMS ENGINEERING RESEARCH CENTER

Verification and Validation (V&V) of System Behavior Specifications

Final Technical Report SERC-2018-TR-116

October 31, 2018

Principal Investigator:
Dr. Kristin Giammarco, Naval Postgraduate School

Co-Principal Investigators:
Ron Carlson, Naval Postgraduate School
Dr. Mark Blackburn, Stevens Institute of Technology

Research Team:
Dr. Mikhail Auguston, Naval Postgraduate School
Dr. Rama Gehris, Naval Postgraduate School
Marianna Jones, Naval Postgraduate School
Bruce Allen, Naval Postgraduate School
David Shifflett, Naval Postgraduate School
John Quartuccio, PhD Student, Naval Postgraduate School
Kathleen Giles, PhD Student, Naval Postgraduate School
Chris Wolfgeher, PhD Student, Naval Postgraduate School
Gary Parker, PhD Student, Naval Postgraduate School

Sponsor: Naval Air Systems Command (NAVAIR)

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY

Castle Point on Hudson, Hoboken, NJ 07030

Table of Contents

CHANGE RECORD

Interim Report Version 1	Added a change record Corrected minor typos Added description of Training Content and Delivery task
Interim Report Version 2	Expanded description of Training Content and Delivery task
Interim Report Version 3	Updated on MP development progress (version 3) Added description of installable MP tool
Final Report	Updated on MP development progress (version 3.5) Updated findings and recommendations Updated description of installable MP tool in Appendix D Updated patterns catalog and added queries and table summary in Appendix E Updated models and download instructions in Appendix F Added new section on Skyzer modeling in Appendix G

1. TASK MOTIVATION AND OBJECTIVES

This technical report summarizes the accomplishments for Research Task RT-176 – Verification and Validation (V&V) of System Behavior Specifications.

The NAVAIR workforce has a need for Model Centric Systems Engineering (MCSE) methods, processes and tools (MPTs) capable of assessing the goodness of system behavior specifications and other requirements earlier in the lifecycle of a system. In particular, the NAVAIR Systems Engineering Transformation (SET) initiative aims to leverage and extend existing research in the area of MPTs for performing early V&V of requirements and architecture models managed within its organization, and to educate its workforce in the use of automated tools for conducting early and continuous V&V across the entire lifecycle. Several Unmanned Aerial Vehicle (UAV) system models have been developed for use as a case study to test new and improved MPTs that have been developed as a result of this task. These MPTs are expected to apply to other systems in many domains throughout DOD and other government agencies.

The objectives of this work are aligned to NAVAIR SET tasks as follows:

Model-based V&V Demo (Task Lead: Kristin Giammarco)

- Formalize UAV behavior specifications into MBSE architecture tool(s)
  - using Monterey Phoenix for comprehensive use case scenario generation
  - Core Value Proposition: Manual drawing of a limited set of use cases is replaced with automatic and comprehensive scenario generation, enabling humans to spend more time on requirements analysis and V&V tasks that cannot be automated. Correction of errors in system behavior is then done much earlier.
- Demonstrate use of the UAV behavior model for early V&V analysis of requirements
  - using MP to expose positive and negative system behaviors permitted by the design
  - Core Value Proposition: Requirements gaps are identified and fixed early (before contracting), through inspection of a comprehensive set of use case scenarios.
- Formalize patterns of common design flaws or other model properties
  - using MP event grammar to store system behavior templates
  - Core Value Propositions: Save money by eliminating error-prone, labor-intensive, and expensive manual checking, and provide specification for testing contracted models for presence or absence of wanted and unwanted system behaviors discovered during early model-based V&V.

Training Content Development and Delivery (Task Lead: Ron Carlson)
The mission of this task is to create a full modeling methodology training curriculum. Training materials were developed to teach the NAVAIR workforce the necessary skills to perform in a model centric environment.

The goals of this task were to create an acquisition workforce transformed to understand and value model-based methods as well as be trained in model-based collaboration and technical review methods.

The objectives of this task were to create a series of classes and workshops that will be available as needed by the workforce.

Specific tasks for FY18 included:

1. Creation of a workshop that will serve as a kick-off for a program office to start a program as a model based project. This workshop will provide an overview of the processes, methods and tools needed in order to create a model based acquisition program. This workshop will cover the program from conception to approximately source selection.

2. Creation of course objectives for three courses:

    a. Basics of SysML. A course on the basics of SysML.

    b. Intermediate SysML. A course that will introduce how to develop models using SysML and MagicDraw Cameo.

    c. Advanced SysML. A course that goes further than the basics of SysML and delves deeper into the use of the Cameo tool to create architectures.

3. Creation of the Basics of SysML course.

Coaching & Mentoring (Task Lead: Kristin Giammarco)

- Create a catalog of typical architecture model views for behavior containing good practices (patterns[*]), poor practices (anti-patterns[†]), and pattern / anti-pattern examples.
    - See Appendix E for this catalog.
- Provide ongoing mentoring and coaching support as needed on pilot projects.

2. Related Work

Verification and validation (V&V) are distinct processes used for ensuring that a system meets its requirements and specifications, and satisfies the user's need. *System verification* is "the confirmation, through the provision of objective evidence, that specified requirements have been fulfilled" (SEBoK authors, 2016). Verification is performed throughout a system's lifecycle and involves performing tests to ensure the system continues to meet the requirements and specification as the system develops. *System validation* is the procedure used to ensure compliance of any system element with its intended purpose (SEBoK authors, 2016). Modeling and simulation are frequently used to support V&V activities throughout a system's lifecycle.

Model V&V should be conducted for each use of the model, as the model may be valid for one set of conditions but invalid for another (Sargent, 2015). *Model verification* ensures the implementation of the

[*] Tool-agnostic patterns with style guidance for certain critical and useful model views will be incorporated.

[†] Discovered practices that should be avoided are often called anti-patterns since they illustrate examples contrary to good practice.

conceptual model is logically sound, and that the conceptual model is programmed accurately into the computer model (Sargent, 2015). Two recognized verification approaches are model checking and theorem proving (Clarke et. al, 1996). *Model validation* confirms the model generates outputs that accurately reflect the model's purpose. Model V&V should be performed each time the model is modified. Model validation can be accomplished several different ways: independent V&V where a third party decides whether the model is valid (typically the most costly), validation by the model development team using test data, or by the user.

The fundamental purpose of V&V is usually to ascertain whether or not a modeled system will have the expected behavior when it is in operation, and identify any unexpected or unwanted behaviors early so that they can be dealt with in a controlled and least costly setting (Auguston et. al, 2015). Current industry standards for modeling system behavior include the Systems Modeling Language (SysML) viewpoints for sequence, activity, use case and state machine diagrams; system dynamics (SD) models depicting control and feedback in system processes; and agent-based models (ABM) that describe agent behaviors and interactions between agents and with the environment. The following paragraph goes into more depth on each approach and introduces Monterey Phoenix (MP), a new approach and tool for specifying system behavior.

SysML use case, sequence, activity and state transition diagrams provide different views on a system's behavior, and some automated tools enable discrete event simulation of SysML activity diagrams. A current challenge with these diagrams is the difficulty in representing all possible behaviors completely, concisely and legibly enough for inspection during V&V (Auguston et. al, 2015). *SD models* are mathematically rigorous; yet abstract enough for a wide variety of system applications. However, SD models are best used for closed-loop systems in which component dependencies must be considered at the global level (Borshchev & Filippov, 2004). MP[‡] augments a typical *ABM approach* by adding standardization for defining agents and events using formalized event grammar and structured syntax (Ruppel, 2016). MP also has a demonstrated ability to expose incorrect, hazardous, or otherwise undesirable behaviors in processes and system designs so that these unwanted behaviors can be removed or mitigated before they manifest in an actual system (Bryant, 2016; Nelson, 2015; Pilcher, 2015; Revill, 2016). MP addresses the SysML diagram challenges by advocating a separate diagram of behavior for each agent, whose events are interrelated to other events on other diagrams through event sharing and coordination (Giammarco & Auguston, 2013). This model structure allows the generation of a set of sequence diagrams, each describing one possible outcome of behavior at the System of Systems (SoS) level, which is useful content for inspection during V&V activities (Auguston, 2016).

In computer science and software engineering, formal methods (Clarke et. al, 1996; Guttag & Horning, 2012; Hoare, 1985; Jones, 1990; Rushby, 1993; Spivey, 1988) are mathematically grounded techniques including logic, semantics, and formal languages (Ruppel, 2016). Some use the term *lightweight* to characterize an approach used to analyze part of the specification document without re-baselining the entire specification, and the term *heavyweight* to describe a deeper, complete application of the methodology (Easterbrook et. al, 1998) (Agerholm and Larsen, 1998) (Woodcock et. al, 2009). Woodcock et al. (2009) propose that the use of formal methods looks to be increasing, but is mainly confined to critical systems development. MP's applicability as a lightweight formal method at various

‡ MP is a system architecture and workflow behavior modeling language and tool developed at NPS. The government-owned MP-Firebird tool is hosted publicly at https://firebird.nps.edu, has no licensing fees, and requires no installation.

levels of abstraction and ability to be used by non-technical stakeholders (Auguston et. al, 2015; Bryant, 2016) is expected to promote a more widespread use of formal methods in system behavior V&V, exposure of more design flaws earlier (Auguston et. al, 2015; Giammarco & Auguston, 2013), and a new capability to identify and detect patterns (Giammarco, 2014; Rodano & Giammarco, 2013) of good or bad behaviors in architecture models so that patterns discovered on one project may benefit other projects.

3. TECHNICAL ACCOMPLISHMENTS

This section of the technical report summarizes the project accomplishments by SET task. To maximize reuse and dissemination of the deliverables, they are organized into the following appendices to this report so that they can be separated into standalone content and shared with the interested parties.

- Appendix A: List of Publications and Invited Talks Resulted
- Appendix B: References Cited
- Appendix C: Collaborator Courses that Integrate or Contribute Research Results
- Appendix D: Monterey Phoenix Overview
- Appendix E: Catalog of Reusable Architecture Patterns and Anti-patterns
- Appendix F: Instructions for Downloading MP Models
- Appendix G: Model Based V&V (MCSE MPT) Demonstration

Spanning across all SET tasks has been the ongoing development and maintenance of the Monterey Phoenix modeling language and tool (Appendix D), as well as documentation and dissemination of the research (Appendix A). Progress on the evolution of MP has been reported in status updates throughout the year. At this point in time, MP version 3.5 is complete and implemented on the MP-Firebird server at https://firebird.nps.edu. A locally installable MP application was also developed for the sponsor and is further described in Appendix D.

Whereas Appendix A exclusively contains works that resulted from this research effort, Appendix B contains a broader set of prior and related work cited throughout this report and its appendices. Appendix C contains a description of the courses that integrated and contributed to this research; many NPS Master's students did project work utilizing the research materials, the best of which was leveraged back into the project. Appendix D contains an overview of MP that serves as an initial tutorial for the MP user community. The purpose of Appendix D is to teach the NAVAIR workforce and others the fundamental concepts of MP. Appendix E provides a catalog of reusable architecture patterns and anti-patterns to keep in mind throughout a modeling effort. Appendix F provides instructions for downloading the MP models developed for this research. Building on the fundamental ideas in Appendices D and E, Appendix G teaches how to verify and validate system behavior specifications using the Navy-developed MP-Firebird tool.

The Navy-developed Monterey Phoenix software is the backbone of this research. The code for MP has been updated to support new analysis questions for behavior model V&V. Event timing attributes make it possible to specify and test real time system architecture models and other time-dependent

behaviors. It becomes possible to generate performance estimates for process models as well, opening a whole new dimension of research in MP-enabled system behavior modeling.

Event attributes provide a foundation for building probabilistic MP models for different kinds of statistical estimates. This mechanism supports different kinds of probabilistic simulation, including different Bayesian inference rules. This kind of functionality is not available in most other system modeling tools. Event attributes also support integration of cost and other resource consumption into system behavior models, which is a new significant benefit for system behavior modeling.

User-defined relations implemented in MP v.3.0 combined with event attributes in MP v3.5 add new capabilities to system architecture and process modeling, making MP potentially a very powerful and useful tool for system design, both software and other. That may be a critical turn for the introduction of model-based systems engineering into practice.

The MP software is at the point where integration with UML/SysML/DoDAF -enabled tools is possible. MP brings scope-complete trace generation and quantitative analysis on / interrogation of scope-compete sets of scenarios.

The completed MP version 3.5 has the following essential new language features:

- Event attributes, including timing attributes, aggregate operations, extended MP expressions. It became possible to specify and test real time system architecture models.
- MP now supports various probabilistic simulation methods and estimates. Type 1 probabilistic models - independent alternative selection probabilities, and Type 2 - means to calculate Bayesian probabilities for traces and events.
- MAP construct for MP model reuse, the first step towards architecture template reuse.
- Significant update of trace layout algorithm – much better and more readable sequence layout for event trace rendering. The new trace layout algorithm also significantly improves trace generator's performance, eliminating relatively slow JavaScript code.
- MAY_OVERLAP predicate, language construct to support concurrent behavior specification and verification.
- $$EVENT generic event type facilitating generic modeling patterns and additional event trace views.
- Significantly larger pre-loaded examples set for the new MP features.

The MP v.3.5 compiler and trace generator have been completed and are in working condition on the public server at https://firebird.nps.edu/. The set of pre-loaded examples has been significantly extended and MP Manual document updated to include the MP v. 3.5 features.

Work continues on MP v. 4.0, in particular, on MP extensions for architecture view extraction and rendering. The v4.0 features (work in progress) are:

- New trace layouts – Gantt charts

- Customizable architecture and event trace views
- Queries, reports, and customizable graphs at the system architecture level
- More trace layout algorithm updates

The following subsections outline the mission, goals, and objectives of each SET task in scope of this RT, along with the appendices in which the related deliverables are found.

3.1. MODEL-BASED V&V DEMO

The mission of this SET task is to demonstrate how to conduct early and continuous V&V of requirements through behavior modeling & simulation.

The goal is to develop a behavior model for a UAV employed in an Intelligence, Surveillance, and Reconnaissance (ISR) mission.

The objectives are to:

1. Formalize UAV behavior specifications into MBSE architecture tool(s).
    - Delivered in Appendix F, Appendix G
2. Demonstrate use of the UAV behavior model for early V&V analysis of requirements.
    - Delivered in Appendix F, Appendix G
3. Formalize patterns of common design flaws or other model properties
    - Delivered in Appendix E

3.2. TRAINING CONTENT DEVELOPMENT AND DELIVERY

The mission of this task is to create a full modeling methodology training curriculum. The training materials and instructors developed will be able to teach the NAVAIR workforce the necessary skills to perform in a model centric environment.

The goals of this task are to create an acquisition workforce transformed to understand and value model-based methods as well as be trained in model-based collaboration and technical review methods.

The objectives of this task are to create a series of classes and workshops that will be available as needed by the workforce.

Specific tasks for FY18 completed include:

1. Creation of a workshop that will serve as a kick-off for a program office to start a program as a model based project. This workshop will provide an overview of the processes, methods and tools needed in order to create a model based acquisition program. This workshop will cover the program from conception to approximately source selection.
2. Creation of course objectives for three courses:

a. Basics of SysML. A course on the basics of SysML.

b. Intermediate SysML. A course on the the development of models using SysML and MagicDraw Cameo.

c. Advanced SysML. A course that goes further than the basics of SysML and delves deeper into the use of the Cameo tool to create architectures.

3. Creation of the Basics of SysML course.

3.3. COACHING & MENTORING

The mission of this task is to equip model-developing members of the workforce with good MBSE skills and practices for behavior modeling.

The goals of this task are to realize the claimed benefits of MBSE in behavior modeling practice, and to sustain effective MBSE implementation through ongoing mentoring and coaching.

The objectives of this task are to collect examples of good and poor behavior modeling practices, develop and teach patterns of good and poor practices, and provide ongoing coaching & mentoring for pilot teams.

Patterns (best practices) and anti-patterns (poor modeling practices) are cataloged in Appendix E.

4. CONCLUSIONS AND WAY AHEAD

RT-176 resulted in multiple advancements and contributions to the systems modeling community. Appendix A lists numerous papers and invited talks that were produced as a result of RT-176. The papers disseminate the major MCSE MPTs created during both years of the project. Appendix B organizes all other references cited throughout this report. Appendix C provides a brief description of graduate-level courses that have integrated, tested or contributed to the MCSE MPTs developed for this research task. Appendix D contains a general overview of the Monterey Phoenix (MP) approach used throughout this research task. Appendix E provides a catalog of reusable architecture patterns and anti-patterns that were created during the research task. Appendix F provides download instructions for the MP models developed for this research effort, and Appendix G demonstrates the application of MPTs for verification and validation of requirements with MP, drawing on the UAV models for examples.

The deliverables discussed in the status reports are mapped to the report appendices in the table below.

Table 4.1 Deliverables

Deliverable	A013 Technical Report Section
Conference papers, presentations, Master's theses, and textbook chapters	Appendix A
A013 Technical Report containing: • Findings, recommendations	Conclusions and Way Ahead

Deliverable	A013 Technical Report Section
• Instructions for applying demonstrated MCSE MPTs on other systems	Appendix G
• Catalog of discovered architecture patterns and anti-patterns that are reusable on other NAVAIR systems that use MCSE MPTs	Appendix E
MP models of UAVs with corresponding exhaustive set of use case variants	Appendix F
Final presentation and training materials for the NAVAIR workforce	Appendix D, Appendix G

4.1. FINDINGS

This subsection summarizes research findings supported by the body of work in the report appendices.

4.1.1. FIVE KEY CONCEPTS SUPPORT THE DEVELOPMENT OF SYSTEM AND SOS BEHAVIOR MODELS

Behavior logic modeling and simulation ought to be conducted prior to characterization of behavior so that the developer starts with a logical and appropriately refined description of the system. This behavior description naturally leads to the system architecture definition, activity and sequence diagrams, interfaces, and insight to modular building blocks for system design and implementation. Five key system of systems modeling concepts to guide this process were distilled from the modeling efforts: Separate behaviors and interactions, model system behaviors and environment behaviors, formalize models for automatic execution, properly allocate each task to a human or to a machine, and use abstraction and refinement to manage large models. See Appendix A research products [6] and [13] for more about these principles.

4.1.2. MODEL DEVELOPERS CAN EXPOSE BOTH WANTED AND UNWANTED BEHAVIORS WITHOUT SPECIALIZED SKILLS

Both wanted and unwanted behaviors can be discovered early in system models using the Monterey Phoenix approach and language. Execution of the model exposes both desirable and undesirable interactions that may not have been expected by the developer. A six-step methodology was proposed to make this new capability available to the NAVAIR workforce and others who are building their organization practice of MBSE (Appendix D). Program managers should expect to see more invalid behaviors in their systems being exposed earlier in design (at the architecture level) with MP as part of their tool suite. Students ranging from high school to graduate-level education were able to learn MP quickly and then use it to expose unspecified and potentially invalid behaviors on systems with which they were familiar, suggesting that this lightweight formal method approach for behavior model V&V is relatively user friendly and easy to learn [1].

4.1.3. MODEL DEVELOPERS CAN GENERATE ALL USE CASE SCENARIOS AUTOMATICALLY UP TO A SCOPE LIMIT

MP automatically generates a comprehensive set of use case scenarios. Every possible system or SoS-level behavior is generated up to a scope limit for verification, validation, and documentation of counterexamples. A major advancement in model-based verification is *assertion checking* with MP, which allows one to state requirements for behaviors that must be *avoided*, and prove that they are absent from the design up to a specified scope limit. The scope limit is the maximum number of iterations for loops in the behavior model. We expect most V&V issues to be exposed in our behavior

models at a small scope of about 3 given Jackson's (2006) Small Scope Hypothesis, further discussed in Appendix D. A major advancement to model-based validation is *counterexample documentation*, which may be used to demonstrate (e.g., to program leadership, or to new workforce members) why a certain requirement is needed – i.e., *what scenarios could emerge* if the requirement is relaxed removed. A main impact of MP on model-based V&V is its ability to make many more descriptions of emergent behaviors available to decision makers earlier in the system's design, so that they may be carefully planned for or removed in the design rather than haphazardly dealt with after they emerge in the actual system. Appendices D-G and research product [5] provide a starting guide for new MP modelers who seek to repeat the approach on their system or SoS of interest.

4.1.4. ALL BEHAVIORS NOT EXPLICITLY SUPPRESSED CAN EVENTUALLY EMERGE

In the course of this research, it became clear that MP could be used to steer the behaviors emerging from SoS models by relaxing or restricting control over component or system interactions. We view each system as having its own behaviors, all of which may eventually manifest unless we deliberately suppress those behaviors. We coax these behaviors out in simulation by adding constraints one at a time within and among comprehensive system models. Using MP, we consider positive emergence to be the acceptable behaviors and interactions that remain after the negative emergence has been thoroughly exposed and pruned.

MP-Firebird provides the developer with an interactive simulation environment to test whether constraints and coordination of events have the desired effects. Using assertions to propose a query and then marking the associated traces enables the user to quickly identify the model response. Within the scope of execution, all possible permutations of model output are derived and analyzed within the execution environment. This provides a level of assurance that an architecture that complies with the model definition and constraints will have known and bounded results. Constraint types include logical limitations, simplification decisions, and design requirements. Analysis of the model is necessary to avoid overly-constraining the model. An overly-constrained model could eliminate essential outcomes or the failure to identify emergent properties of the system.

4.1.5. MODEL DEVELOPERS CAN DETECT, CLASSIFY, PREDICT AND CONTROL EMERGENT BEHAVIORS WITH MP

MP has been used to detect, classify, predict and control emergent behaviors of SoS early in design, during modeling and simulation. MP automatically *detected* all possible combinations of system behaviors and interactions permitted by the model. During inspection, the emergent behaviors were *classified* as favorable or unfavorable, simple, weak, strong, and positive or negative. The human inspector used each generated scenario as a canvas for *predicting* future states of emergence, which influenced classification. The negative emergent behaviors were *controlled* through modification of the individual behavior models or relaxation / restriction of interaction constraints. In making this discovery, we found a need to refine and formalize the emergent behavior classification taxonomy, which the example models generated over the past year might be used to help shape. The essential features of MP enabling control of emergent behaviors are: ability to formalize interactions within the system and between the system and its environment at an appropriate level of abstraction, automated generation of exhaustive sets of scenarios within a given scope, and tools for the analysis of generated scenarios supporting assertion checking, event trace visualization and annotation. See Appendix A research product [21] for the research behind this finding.

4.1.6. PROBABILITIES FOR BEHAVIOR OUTCOMES CAN BE CALCULATED

The MP execution environment supports analysis of the results including the fundamental structure of a Bayesian belief network, facilitating a probability calculation of each possible outcome. During the

development of the system architecture, it may not be possible to avoid all undesirable outcomes, however the resulting system provides insight on ways to limit the effects of undesirable events. See Appendix A research products [9] and [22] for the research behind this finding.

4.1.7. TEMPLATES FOR BEHAVIOR PATTERNS CAN BE DERIVED INDUCTIVELY

Templates of prototypical behaviors enable an automated means to parse the resulting trace and characterize the positive and negative behaviors of the model. Interestingly, these templates are not imposed on the system but are derived inductively by first defining the system interactions and then by executing the model to identify the naturally occurring results. See Appendix A research products [9] and [22] for the research behind this finding.

4.1.8. ALL POSSIBLE INTERACTIONS FOR A MODEL CAN BE SUMMARIZED IN MATRIX FORM

The execution environment provides an automated means to populate a Design Structured Matrix (DSM), tabulating all possible interactions. Analysis of the DSM affords the system developer of a means to understand logical groupings of functions, and thereby supporting modularization and design synthesis. See Appendix A research product [22] for the research behind this finding.

4.1.9. SYSTEM BEHAVIORS COULD BE FORMALLY SPECIFIED CONSISTENTLY BY BOTH GOVERNMENT (IN REQUIREMENTS) AND CONTRACTORS (IN IMPLEMENTATION)

A model described through a high level of abstraction effectively defines system behaviors at a fundamental level, as characterized by the system internal functions and by system-to-system external interactions. Applying formal methods to the behavior description enables a precise description of the core interactions while eliminating the minutia of detail irrelevant to the functional description. MP is therefore just as useful pre-contract award as it is post-contract award, and may be used as a formal specification bridge between government sponsors as well as contractors/solution providers. Before writing requirements into a contract, MP can be used to test and debug those requirements at a solution-neutral level, exposing overlooked customer needs and expectations that ought to be included in the requirements that are handed to the solution developer(s). The scope-complete scenario generation capability of MP regards the independence of interacting systems or components to allow the requirements analysts to try out and inspect all combinations of behaviors, and employ a constraint (requirement) discovery process to shape the overall behavior. We impose or lift constraints in MP to observe the effects on the overall design, and to discover requirements that may not have been found otherwise until much later.

4.1.10. MANY ARCHITECTURE MODELING LESSONS LEARNED ARE FORMALIZABLE INTO ANTI-PATTERNS

The anti-patterns presented in Appendix E are also available for specification as part of a contract to encourage good MBSE practices in both solution-neutral and solution-oriented models developed for the implementing organization. A conceptual data model (CDM), which underlies every system model, is a powerful logical language for expressing typical architecture modeling anti-patterns (practices to avoid) in a tool-agnostic manner. The anti-patterns 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. Early testing in the Innoslate tool has codified these anti-patterns and picked up hundreds of overlooked deficiencies in dozens of different academic and real project models. Further testing, however, is needed to fine-tune the anti-patterns, common exception cases, and metrics to assist with model maturity tracking over time. The high-level specification of the anti-patterns enables model developers to delegate the checking and enforcement

of typical modeling best practices to the modeling software tools, allowing them more time to focus on the harder problems that human capital is needed for (e.g., verification, validation, refutation, pattern-finding activities).

4.1.11. Design Reference Missions (DRMs) are useful source data for model development

Development of Design Reference Missions (DRMs) has been found to be an effective tool for scoping short- to medium- term MBSE analysis efforts (such as pilot projects). A DRM provides a solution-neutral operational context and a thorough description of a problem space for a modeling team to use as source data. The process of developing a DRM answers many important questions about a problem space, and lays the groundwork for a successful model-based development or analysis effort. See sample DRMs developed by NPS students in Appendix A, research product [23] and Appendix G section 1.

4.1.12. MP may be mapped to SysML and other graphical notations

We conducted a preliminary crosswalk of MP with the System Modeling Language (SysML) and the Lifecycle Modeling Language (LML). This crosswalk shows a relatively straightforward mapping from graphical languages forward to MP to achieve basic executable MP models that can be modified and optimized for generating many more scenarios at higher scopes after translation from SysML or LML. These results are promising for graphical modeling language users who want to gain access to the comprehensive scenario coverage and V&V capability delivered by MP.

4.2. Recommendations

We recommend the following actions be taken based on the results of this research project:

4.2.1. Develop a body of evidence for the efficacy of new modeling methods

Adopting new modeling methods in an organization will be more socially than technically difficult. Workforce members who have been satisfied with older methods for a long time may resist changing their way of doing business. To fully support the newer methods, the workforce should be provided with convincing evidence that new modeling methods are better than current or previous methods being used with respect to their experience. MBSE pilot projects should continue to be used to develop a body of evidence on the efficacy of the modeling methods proposed for wider adoption.

4.2.2. Use Monterey Phoenix on MBSE pilot projects

Continue to deploy the proposed methodology described in Appendices D and E on pilot MBSE projects for testing and implementation on actual systems. Research product [27] applies MP to a NAVAIR pilot project and contributes evidence for the utility of MP. This process should be deployed with urgency to those tasked with identifying and purging a design of errors, vulnerabilities, safety hazards or other critical issues, or Independent Verification & Validation (IV&V).

4.2.3. Train model developers how to verify and validate SysML models using MP

Model developers with less modeling experience may initially have an easier time learning and wielding MP than very experienced model developers. Where possible, start the less experienced modelers on MP, having them use models developed by experienced SysML modelers as source data. This will effectively teach them two skills at once: analysis of SysML models, and verification and validation of those models using MP. This approach enables the MP modelers to serve as “smart ignoramus” (Berry,

1998), asking clarifying questions, exposing tacit assumptions, and providing constructive feedback that helps to improve the quality of the SysML model and of the design it describes.

4.2.4. GENERATE SEQUENCE DIAGRAMS AUTOMATICALLY FROM OTHER BEHAVIOR VIEWS

Utilize MP's automatic and scope-complete event trace generator to automatically create *formal specifications* for sequence diagrams that can be transcribed to and from SysML. Manual generation of many use case scenarios as event traces or sequence diagrams is time consuming, error prone, deficient in scenario coverage, and no longer necessary. Human capital should be focused on creating behavior models such as activity diagrams and state transition diagrams with logic constructs, and automated tools tasked with extracting sequence diagrams / event traces from those models.

4.2.5. SCRUB THE REQUIREMENTS WITH MP BEFORE WRITING CONTRACTS

Use MP to test and debug high-level system requirements before they are written into contracts. This activity can help to stabilize the requirements earlier, and significantly reduce program costs by resulting in fewer engineering change proposals and contract modifications later in the system's lifecycle. The new capability to control emergent behaviors with MP may have a strong positive influence the requirements analysis process. Be sure to record and track all V&V issues discovered (as counterexamples to good behavior) by MP analysis, so that the cost of exposing and fixing each issue early can be compared to that of finding and fixing the same behavior in a later lifecycle phase.

4.2.6. USE THE MODEL'S ONTOLOGY AS A LANGUAGE TO DESCRIBE MODEL ANTI-PATTERNS

A conceptual data model (CDM) / ontology underlies every system model, and is a powerful logical language for expressing typical architecture modeling anti-patterns in a tool-agnostic manner. Decide which of the anti-patterns from Appendix E the organization will adopt, map them to the organization's chosen ontology, and communicate them as standard practices to be avoided in models developed for them.

4.2.7. FORMALIZE THE TYPES AND DEFINITIONS OF EMERGENT BEHAVIOR FOR USE IN RISK ANALYSIS

The classification taxonomy for different types of emergent behaviors (e.g., simple, weak, strong, spooky, positive, negative, etc.) in current use may benefit from formalization based on the example models generated as research products of this project over the past year. There is practical value to having clear criteria for classifying different types of emergent behavior, such as aiding in the conduct of risk analysis (e.g., in assigning priorities to behaviors), and developing metrics for emergent behaviors in designs (e.g., to generate stoplight charts for behaviors of concern). Now that a collection of example models containing different types of emergent behaviors exists, the classification taxonomy should be tested and refined into a standard way to assign and substantiate the types of emergent behavior.

4.2.8. PROVIDE MODELERS WITH GOOD SOURCE MATERIALS FOR THE DEVELOPMENT EFFORT

Modeling teams should be provided with a Design Reference Mission (DRM) or similar document that summarizes the operational context and operating environment assumptions for the system they are being asked to model. The DRM answers many important questions about a problem space, and lays the groundwork for a successful modeling effort.

4.2.9. DEVELOP A GRAPHICAL GATEWAY TO MP

While analysts of critical systems typically have a mathematical or formal methods background that comes with coding experience, others may have strong preferences for graphical languages and tools.

Some may even have an aversion for coding languages that, if overcome, would enable them to take advantage of the full strength of the MP language. These users would benefit from a graphical gateway to MP analysis, whereby specially profiled graphical models in a notation of the user's choice are automatically translated into MP event grammar as a starting point for expansion and analysis natively in MP. To encourage those who are used to working exclusively in graphical languages to learn and use MP, extend or profile familiar views and notations in common use (such as SysML) to enable the first draft of MP code to be generated from them.

4.2.10. INFORM THE WORKFORCE OF THE RESULTS OF THIS RESEARCH

Have the research team summarize the content in this report into a workshop format that can be delivered to the NAVAIR workforce. This will ensure a good transition of the many research products resulting from this effort in a timely manner, and have a positive impact on Systems Engineering Transformation. Also, create a workshop that covers MBSE activities in a program after source selection throughout the remaining life of the program.

4.3. ACKNOWLEDGEMENTS

The RT-176 research team would like to thank our research sponsor, NAVAIR Systems Engineering Transformation (SET), for investing in the development and integration of the Monterey Phoenix (MP) behavior modeling approach and tool into its practice of MBSE. We would also like to acknowledge the contributions of the many NPS Master's students who contributed content and insight to the RT-176 project, especially LCDR Chris Krukowski, LCDR Richard Moebius, Mr. Thomas Moulds, LT Stephan Mathos, Mr. Stephen (Alex) Rambikur, Mr. Ernie Lemmert, Mr. Cody Reese, and Mr. Anthony Constable, who made direct and specific contributions that are enclosed in this report.

APPENDIX A: LIST OF PUBLICATIONS AND INVITED TALKS RESULTED

1. Giammarco, Kristin and Katy Giles. "Verification and Validation of Behavior Models using Lightweight Formal Methods." Proceedings of the 15th Annual Conference on Systems Engineering Research (CSER), Redondo Beach, CA, March 23-25, 2017. Received **Best Paper Award** for Transition in Systems Engineering Research sponsored by MITRE.
2. Giammarco, Kristin. "Verification and Validation of Behavior Models using Lightweight Formal Methods." Invited Talk, Engility Webconference, April 26, 2017.
3. Giammarco, Kristin. "Verification and Validation of Behavior Models using Lightweight Formal Methods." Invited Talk, MITRE Technical Exchange Meeting, McLean, VA, May 3, 2017.
4. Giammarco, Kristin. "Verification and Validation of Behavior Models using Lightweight Formal Methods." Invited Talk, NAVSEA M&S Forum Webconference, Dahlgren, VA, May 9, 2017.
5. Giammarco, Kristin and Clifford A. Whitcomb. "Comprehensive use case scenario generation: An approach and template for modeling system of systems behaviors." Proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.
6. Giammarco, Kristin. "Practical Modeling Concepts for Engineering Emergence in Systems of Systems." Proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.
7. Giammarco, Kristin. "Architecture Modeling Software Analytics: Model quality and maturity assessment using automated tools." Proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.
8. Quartuccio, John, Kristin Giammarco, and Mikhail Auguston. "Identifying Decision Patterns Using Monterey Phoenix." Proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.
9. Quartuccio, John, Kristin Giammarco, and Mikhail Auguston. "Deriving Stochastic Properties from Behavior Models Defined by Monterey Phoenix." Proceedings of the 12th Annual System of Systems Engineering Conference, Waikoloa, HI, June 18-21, 2017.
10. Giammarco, Kristin. "Verification and Validation of Behavior Models using Lightweight Formal Methods." Invited Talk, SOSECIE Webinar, August 8, 2017.
11. Mathos, Stephan. "A Model Based Systems Engineering Analysis of Anti-Submarine Warfare Tactics for Crew-Based Aircraft." Master's Thesis, DKL restricted collection. Naval Postgraduate School, September 2017.
12. Moulds, Thomas. "Analysis of Mission Effectiveness: Modern System Architecture Tools for Project Developers." Master's Thesis. Naval Postgraduate School, September 2017.
13. Auguston, Mikhail. "Monterey Phoenix: System and Software Architecture and Workflow Modeling Language." Naval Postgraduate School, modified: September 28, 2017.
14. Giammarco, K. "Verification and Validation (V&V) of System Behavior Specifications." Presented to the NASA Independent Verification & Validation (IV&V) Facility on invitational travel orders, Fairmont, WV, USA, October 18, 2017.
15. Giammarco, K. "RT-176: Verification and Validation (V&V) of System Behavior Specifications." Presented at the Annual SERC Sponsor Research Review (SSRR), Washington, DC, USA, November 8, 2017.

16. Giles, Kathleen, and Kristin Giammarco. (2017). "Mission-based Architecture for Swarm Composability (MASC)." *Procedia Computer Science* 114: 57-64.
17. Giles, Kathleen. "Mission-Based Architecture For Swarm Composability (MASC)." Doctorate in Systems Engineering (SE PhD). December 2017.
18. Giammarco, K. "Lessons Learned from Engineering Emergence Research." Presented to the INCOSE SoS Research Roundtable, online webinar, January 21, 2018.
19. Giammarco, K.; Giles, K. "Verification and validation of behavior models using lightweight formal methods." Presented to the INCOSE North Texas Chapter, online webinar, February 13, 2018.
20. Giammarco, K. "Practical Modeling Concepts for Engineering Emergence in Systems of Systems." Presented to the Presented to the System of Systems Engineering Collaborators Information Exchange (SoSECIE), online webinar, March 20, 2018.
21. Giammarco, Kristin and Mikhail Auguston. "Behavior modeling approach for the early verification and validation of system of systems emergent behaviors," In *Engineering Emergence: A Modeling and Simulation Approach*, edited by Larry Rainey and Mo Jamshidi. Boca Raton, FL: CRC Press Taylor & Francis Group, 2018.
22. Quartuccio, John and Kristin Giammarco. "A model-based approach to investigate emergent behaviors in systems of systems." In *Engineering Emergence: A Modeling and Simulation Approach*, edited by Larry Rainey and Mo Jamshidi. Boca Raton, FL: CRC Press Taylor & Francis Group, 2018.
23. Giammarco, Kristin, Spencer Hunt, Clifford A. Whitcomb. (2018). "Using the Design Reference Mission for Framing Naval Ship Design Problems." *Naval Engineers Journal* 180, no. 1:53—63.
24. Giammarco, K. "Monterey Phoenix Approach and Tool for Behavior Modeling: A Tutorial." Presented at the Military Operations Research Society Symposium (MORSS), June 18, 2018.
25. Giammarco, K. "The Monterey Phoenix Approach and Tool for Behavior Modeling." Presented at the Military Operations Research Society Symposium (MORSS), June 20, 2018.
26. Auguston, Mikhail. "Monterey Phoenix: System and Software Architecture and Workflow Modeling Language." Naval Postgraduate School, modified: August 30, 2018.
27. Krukowski, Christopher. "Requirements Verification and Validation for the MQ-25 Using Scope-Complete Behavior Models." MSSEM. September 2018.
28. Giles, Kathleen and Kristin Giammarco. "A Mission-based Architecture for Swarm Unmanned Systems." *Journal of Systems Engineering*, accepted for publication.

APPENDIX B: REFERENCES CITED

1. Agerholm, S., & Larsen, P. G. (1998, October). A lightweight approach to formal methods. In International Workshop on Current Trends in Applied Formal Methods (pp. 168-183). Springer, Berlin, Heidelberg.
2. Aizier, B., LIZY-DESTREZ, S., SEIDNER, C., CHAPURLAT, V., PRUN, D., & WIPPLER, J. L. (2012, July). 1.6. 1 xFFBD: towards a formal yet functional modeling language for system designers. In INCOSE International Symposium (Vol. 22, No. 1, pp. 170-183).
3. Auguston, M. (1995, May). Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In AADEBUG (pp. 277-291).

4. Auguston, M. (2009a). Software architecture built from behavior models. ACM SIGSOFT Software Engineering Notes, 34(5), 1-15.
5. Auguston, M. (2009b, October). Monterey phoenix, or how to make software architecture executable. In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (pp. 1031-1040). ACM.
6. Auguston, M., & Whitcomb, C. (2010). System architecture specification based on behavior models. In Proceedings of the 15th ICCRTS Conference (International Command and Control Research and Technology Symposium), Santa Monica, CA, June 22-24, 2010.
7. Auguston, M., & Whitcomb, C. (2012). Behavior Models and Composition for Software and Systems Architecture, Proceedings of the 24th ICSSEA Conference (International Conference on Software and Systems Engineering and their Applications), Paris, France, October 23-25, 2012.
8. Auguston, M. (2014) *Behavior models for software architecture*. Technical Report NPS-CS-14-003, Monterey, California. Naval Postgraduate School.
9. Auguston, M., Giammarco, K., Baldwin, W. C., & Farah-Stapleton, M. (2015). Modeling and Verifying Business Processes with Monterey Phoenix. Procedia Computer Science, 44, 345-353
10. Auguston, M. (2016). "System and Software Architecture and Workflow Modeling Language Manual." Available from https://wiki.nps.edu/display/MP/Monterey+Phoenix+Home
11. Berry, D. (1998, October). "Formal Methods: The Very Idea, Some Thoughts About Why They Work When They Work", *Proceedings of the 1998 ARO/ONR/NSF/ARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Monterey, CA, pp. 9-18.
12. Borshchev, A., & Filippov, A. (2004, July). From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools. In Proceedings of the 22nd international conference of the system dynamics society (Vol. 22).Box, G. E. (1976). Science and statistics. *Journal of the American Statistical Association*, *71*(356), 791-799.
13. Borshchev, A. (2013). The big book of simulation modeling: multimethod modeling with AnyLogic 6. AnyLogic North America.
14. Box, George EP. (1976). Science and statistics. *Journal of the American Statistical Association*, 71(356):791—799.
15. Bryant, J. (2016, June). *Using Monterey Phoenix to analyze an alternative process for administering Naloxone*. Capstone Research Project, Science and Math Academy, Aberdeen, MD. June 2016. Retrieved September 17, 2016 from http://www.scienceandmathacademy.com/academics/srt4/student_work/2016/bryant_jordan.pdf.
16. Buede, D.M. (2009). *The engineering design of systems: models and methods*. John Wiley & Sons, 2nd edition.
17. Calvano, C. N., & John, P. (2004). Systems engineering in an age of complexity. Systems Engineering, 7(1), 25-34.
18.
19. Clarke, E. M., & Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, *28*(4), 626-643.
20. Climate-data.org (2017). Climate: Port-au-Prince. Retrieved May 21, 2017 from https://en.climate-data.org/location/3571/.
21. Crawley, E., De Weck, O., Magee, C., Moses, J., Seering, W., Schindall, J., ... & Whitney, D. (2004). The influence of architecture in engineering systems (monograph.
22. Crawley, E., Cameron, B., & Selva, D. (2015). System architecture: Strategy and product development for complex systems. Prentice Hall Press.
23. Dahmann, J. S., & Baldwin, K. J. (2008, April). Understanding the current state of US defense systems of systems and the implications for systems engineering. In *Systems Conference, 2008 2nd Annual IEEE* (pp. 1-7). IEEE.

24. Dahmann, J. S., Bhatti, A. S., & Kelley, M. (2009, March). Importance of systems engineering in early acquisition. In *Systems Conference, 2009 3rd Annual IEEE* (pp. 110-115). IEEE.
25. Defense Acquisition University. (n.d.). Defense acquisition guide (DAG). 19:1248, Retrieved June, 2013, from https://www.dau.mil/tools/dag
26. DOD, Department of Defense (2008). Universal Naval Task List (UNTL) (OPNAVINST 3500.38B/MCO3500.26A/USCG COMDTINST 3500.1B Change 1). Washington, DC: Department of the Navy and U.S. Coast Guard. https://doni.daps.dla.mil/Directives/03000%20Naval%20Operations%20and%20Readiness/03-500%20Training%20and%20Readiness%20Services/3500.38B%20-%20Chapter%203%20CH-1.pdf
27. DOD, Department of Defense (2011). Unmanned Aircraft System Airspace Integration Plan. Retrieved June 2, 2017, from http://www.acq.osd.mil/sts/docs/DoD_UAS_Airspace_Integ_Plan_v2_%28signed%29.pdf
28. DODAF, Department of Defense Architecture Framework (2010). Version 2.02, Washington, DC. [Online], Retrieved Feb. 27, 2017 from http://dodcio.defense.gov/Portals/0/Documents/DODAF/DoDAF_v2-02_web.pdf.
29. Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., & Hamilton, D. (1998). Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, *24*(1), 4-14.
30. Eberhard, M. O., Baldridge, S., Marshall, J., Mooney, W., & Rix, G. J. (2013). Mw 7.0 Haiti Earthquake of January 12, 2010: USGS/EERI Advance Reconnaissance Team Report. BiblioGov.
31. Farah-Stapleton, M. (2016, September). *Executable behavioral modeling of system- and software-architecture specifications to inform resourcing decisions.* Doctoral Dissertation, Naval Postgraduate School.
32. Farah-Stapleton, M., Auguston, M., & Giammarco, K. (2016). Executable Behavioral Modeling of System and Software Architecture Specifications to Inform Resourcing Decisions. *Procedia Computer Science*, *95*, 48-57.
33. Fromm, J. (2005). Types and forms of emergence. *arXiv preprint nlin/0506028.*
34. Giammarco K. (2012, June). *Architecture model based interoperability assessment.* Doctoral thesis, Naval Postgraduate School, Monterey, CA.
35. Giammarco, K., & Auguston, M. (2013). Well, you didn't say not to! A formal systems engineering approach to teaching an unruly architecture good behavior. Procedia Computer Science, 20, 277-282.
36. Giammarco, K. (2014). A formal method for assessing architecture model and design maturity using domain-independent patterns. *Procedia Computer Science*, 28, 555-564.
37. Giammarco, K. & Auguston, M. (2015). Monterey phoenix main principles and advantages, viewed July 16, 2015. https://wiki.nps.edu/display/MP/Main+Principles+ and+Advantages.
38. Giammarco, K., & Shebalin, P. (2016). An Instructional Design Reference Mission: Tactical Disk Clearance System [Scholarly project].
39. Giammarco, K. & Giles, K. (2017a). Verification and validation of behavior models using lightweight formal methods." Proceedings of the 15th Annual Conference on Systems Engineering Research. Redondo Beach, CA. March 23-25, 2017. Best paper award
40. Giammarco, K., Giles, K., & Whitcomb, C. A. (2017b, June). Comprehensive use case scenario generation: An approach for modeling system of systems behaviors. In System of Systems Engineering Conference (SoSE), 2017 12th (pp. 1-6). IEEE.
41. Gleick, J., & Berry, M. (1987). Chaos-Making a New Science. Nature, 330, 293.
42. Greenfield, C. M., & Ingram, C. A. (2011). An analysis of US Navy humanitarian assistance and disaster relief operations (No. NPS-LM-11-009). Naval Postgraduate School, Monterey, CA: Graduate School Of Business And Public Policy.

43. Guide, U. D. S. (2008). Systems engineering guide for systems of systems. Version, 1, 20301-3090.
44. Guttag, J. V., & Horning, J. J. (2012). *Larch: languages and tools for formal specification.* Springer Science & Business Media.
45. Hoare, C. A. R. (1985) *Communicating sequential processes*. Vol. 178. Englewood Cliffs: Prentice-hall.
46. Hughes, J., (1989) Why Functional Programming Matters. Retrieved from http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html
47. IBM Corporation. (2010, June) "UPIA DoDAF 2.01 Mapping Reference." [Online], Accessed May 12, 2017. Available: http://www-01.ibm.com/support/docview.wss?uid=swg27019041&aid=1.
48. INCOSE (2007). Systems Engineering Vision 2020.
49. INCOSE (2010). Systems Engineering Handbook - A Guide for System Life Cycle Processes and Activities, Version 3.2. International Committee on Systems Engineering.
50. INCOSE (2015). Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, Version 4. International Committee on Systems Engineering, Wiley.
51. ISO/IEC/IEEE. Systems and Software Engineering - System Life Cycle Processes. International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), Institute of Electrical and Electronics Engineers (IEEE), Geneva, Switzerland, 2015. 15288:2015.
52. Jackson, D. (2002). Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2), 256-290.
53. Jackson, D. (2006). Software Abstractions: logic, language, and analysis. MIT press.
54. JCS, Chairman - Joint Chiefs of Staff (2014). Joint Publication 3-29 Foreign Humanitarian Assistance. Last Accessed: June 2, 2017. http://www.dtic.mil/doctrine/new_pubs/jp3_29.pdf.
55. JCS, Chairman – Joint Chiefs of Staff (2011). "Joint Publication 5-0, Joint Operation Planning." Last Accessed: June 10, 2011. http://www.dtic.mil/doctrine/new_pubs/jp5_0.pdf.
56. JEL+, Joint Electronic Library Plus (2017). Universal Joint Task List. Last modified: April 15. https://jdeis.js.mil/jdeis/index.jsp?pindex=43.
57. Jones, C. B. (1990). *Systematic software development using VDM* (Vol. 2). Englewood Cliffs: Prentice Hall.
58. Kossiakoff, A., Sweet, W. N., SEYMOUR, S., & BIEMER, S. (2003). Systems Engineering: Principles and Practice John Wiley & Sons. New York.
59. Kovács, G., & Spens, K. M. (2007). Humanitarian logistics in disaster relief operations. International Journal of Physical Distribution & Logistics Management, 37(2), 99-114.
60. Krukowski, C., & Giles, K. (2017). Design Reference Mission for Unmanned Aircraft System Conducting Humanitarian Assistance Disaster Relief Mission [Scholarly project].
61. Lifecycle Modeling Language Steering Committee. (2015, December). LML Specification, version 1.1. [Online], Retrieved Feb. 27, 2017 from: http://www.lifecyclemodeling.org/specification/.
62. Long, D., & Scott, Z. (2011). A primer for model-based systems engineering. Lulu. com.
63. Lynch, N. A., & Tuttle, M. R. (1988). An introduction to input/output automata.
64. Maier, M. W. (1996, July). Architecting principles for systems-of-systems. In INCOSE International Symposium (Vol. 6, No. 1, pp. 565-573).
65. Maier, M. W. (1998). Architecting principles for systems-of-systems. Systems Engineering, 1(4), 267-284. doi:10.1002/(sici)1520-6858(1998)1:4<267::aid-sys3>3.0.co;2-d
66. Maier , M. W., & Rechtin, E. (2002). The art of systems architecting (2nd ed.). Boca Raton, FL: CRC Press.
67. Maier, M. W. (2015). The Role of Modeling and Simulation in System-of-Systems Development. *Modeling and Simulation Support of System-of Systems Engineering Applications*, 11-41.
68. Manna, Z., & Pnueli, A. (2012). The temporal logic of reactive and concurrent systems: Specification. Springer Science & Business Media

69. Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychological review, 63(2), 81.
70. Miller, J. H., & Page, S. E. (2009). Complex adaptive systems: An introduction to computational models of social life. Princeton university press.
71. Mitchell, M. (2009). *Complexity: A guided tour.* Oxford University Press.
72. Nelson, C. (2015, November). Modeling a spacecraft communication system using Monterey Phoenix: a systems engineering case study. Master's Project, Stevens Institute of Technology, Hoboken, NJ.
73. NWDC, COMMANDER, N. W. (2006). Foreign Humanitarian Assistance/Disaster Relief Operations Planning.
74. ODUSD (A&T) SSE. (2008). Systems engineering guide for systems of systems. Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems Software Engineering, August 2008. Version 1.0.
75. Pilcher, J. D. (2015). Generation of Department of Defense Architecture Framework (DODAF) models using the Monterey Phoenix behavior modeling approach (Doctoral dissertation, Monterey, California: Naval Postgraduate School).
76. Port-au-Prince [Map]. (n.d.). In Google Maps. Retrieved May 15, 2017, from https://www.google.com/maps/@18.9824209,-72.8840285,229306m/data=!3m1!1e3.
77. Pyster, A., Olwell, D. H., Hutchison, N., Enck, S., Anthony Jr, J. F., & Henry, D. (2012). Guide to the systems engineering body of knowledge (SEBoK) v. 1.0. 1. Guide to the Systems Engineering Body of Knowledge (SEBoK). Accessed July 11, 2015.
78. Qualities of great models. [Scholarly project]. (n.d.). In MIT, Short Course titled Model-Based Systems Engineering: Documentation and Analysis. Retrieved June 11, 2017, from https://mitprofessionalx.mit.edu/courses/course- v1:MITProfessionalX SysEngx3 1T2017/
79. Quartuccio, J., & Giammarco, K. (2018.). Chapter 18 (L. Rainey & M. Jamshidi, Eds.). In *Engineering Emergence: A Modeling and Simulation Approach*, edited by Larry Rainey and Mo Jamshidi. Boca Raton, FL: CRC Press Taylor & Francis Group, 2018.
80. Rambikur, A., Giammarco, K., & O'Halloran, B. (2017a, June). Systems architecture in failure analysis (Applications of architecture modeling to system failure analysis). In System of Systems Engineering Conference (SoSE), 2017 12th (pp. 1-6). IEEE.
81. Rambikur, A., Giammarco, K., & O'Halloran, B. (2017b, November). Improving Fault Tree Analysis through MBSE. Accepted to the 2017 Complex Adaptive Systems Conference, Chicago, IL, October 31 – November 1, 2017.
82. Reese, C. (2017). "A Model-Based Evaluation of Unmanned Aerial Systems for Humanitarian Assistance and Disaster Relief Operations." (unpublished).
83. Revill, M. B. (2016). *UAV swarm behavior modeling for early exposure of failure modes* (Master's thesis, Monterey, California: Naval Postgraduate School).
84. Rodano, M., & Giammarco, K. (2013). A formal method for evaluation of a modeled system architecture. *Procedia Computer Science*, 20, 210-215.
85. Ruppel, S. R. (2016). *System behavior models: a survey of approaches* (Master's thesis, Monterey, California: Naval Postgraduate School).
86. Rushby, J. (1993). *Formal methods and the certification of critical systems*. SRI International, Computer Science Laboratory.
87. Sargent, R.G. (2015). A Tutorial on Verification and Validation of Simulation Models. Proceedings of the 2015 Winter Simulation Conference, 1729–40. http://dl.acm.org/citation.cfm?id=809441.
88. SEBoK authors (2016, March 23). "System Verification," in BKCASE Editorial Board. 2016. *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*, v. 1.6. R.D. Adcock (EIC). Hoboken, NJ: The

Trustees of the Stevens Institute of Technology. Retrieved on September 18, 2016, from *http://sebokwiki.org/w/index.php?title=System_Verification&oldid=50858*
89. Skolnick, F. R., and P. G. Wilkins (2000). "Laying the Foundation for Successful Systems Engineering." Johns Hopkins APL Technical Digest 21 (2): 208–16.
90. Spivey, J. M. (1988). Understanding Z: a specification language and its formal semantics (Vol. 3). Cambridge University Press.
91. Strogatz, S. H. (2012). Sync: How order emerges from chaos in the universe, nature, and daily life. Hachette UK.
92. United Nations (2017). Percentage of Total Population Living in Coastal Areas. Retrieved May 21, 2017, from: http://www.un.org/esa/sustdev/natlinfo/indicators/methodology_sheets/oceans_seas_coasts/pop_coastal_areas.pdf.
93. USN, United States Navy (2015). "A Cooperative Strategy for 21st Century Seapower." Last Accessed: June 2, 2017. http://www.navy.mil/local/maritime/150227-CS21R-Final.pdf.
94. Waldrop, M. M. (1993). Complexity: The emerging science at the edge of order and chaos. Simon and Schuster.
95. Weilkiti. (2013). What is a model? Retrieved July 5, 2017, from http://model-based-systems-engineering.com/2013/03/15/what-is-a-model/
96. Whitcomb, C.A., Auguston, A., and Giammarco, K. (2015) "Composition of Behavior Modeling for Systems Architecture," Chapter 14 in "M&S Support for System of Systems Engineering Applications", 2015, edited by Larry Rainey and Andreas Tolk. Hoboken, NJ: John Wiley & Sons.
97. Whitcomb, C., Giammarco, K., & Hunt, S. (2015). An instructional design reference mission for search and rescue operations. Monterey, California. Naval Postgraduate School.
98. Wikipedia, (2017). Köppen climate classification. Last Modified May 19, 2017. https://en.wikipedia.org/wiki/K%C3%B6ppen_climate_classification.
99. Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. ACM computing surveys (CSUR), 41(4), 19.
100. Yakimenko, O. A. (2011). Engineering computations and modeling in MATLAB/Simulink. American Institute of Aeronautics and Astronautics.
101. Zeigler, B. P. (1976). Theory of modeling and simulation. John Wiley & Sons. Inc., New York, NY.

Zeigler, B. P. (2016). A note on promoting positive emergence and managing negative emergence in systems of systems. *The Journal of Defense Modeling and Simulation*, *13*(1), 133-136.

APPENDIX C: COLLABORATOR COURSES THAT INTEGRATE OR CONTRIBUTE RESEARCH RESULTS

Course No.	Course Name	Description
SE4930	Model Based Systems Engineering	SE4930 presents Model Based Systems Engineering (MBSE) as a key enabler for humans to develop information, interrogate designs, and reason about alternatives using automated tools. This course provides an overview of MBSE tools and methods and how they relate. Students in the summer quarter of AY18 learned how to assess duration and cost for executable activity models based on the RT-176 UAV model, which was used to inform UAV MP modeling efforts.
SE4935	Formal Methods for Systems Architecting	SE4935 provides students with an introduction to the application of formal methods to system architecture model and design analysis. As a directed study, a student in this course applied systematic thinking to develop a user-friendly tutorial on Monterey Phoenix.
SE4151	System Integration and Development	SE4151 provides an advanced introduction to building products and services from the perspective of theory, frameworks, and practice. The aim is to instill a sense of perspective about the means and consequences of integrating system components with regards to cost, schedule, and performance issues. Students in the winter quarter of AY17 performed subsystem integration on the RT-176 UAV model, which was used to inform UAV MP modeling efforts.
SI4022	Systems Architecture for Product Development	This course provides an integrative forum for PD21 students to stimulate holistic, global, and innovative thinking, and to enable critical evaluation of current modes of architecture. We discuss physical systems and software systems, heuristic and formal methods, and the students complete research assignments that provide opportunities to further learn how systems architecture principles are applied in a variety of application areas. Students in the spring quarter of AY17 and AY18 refined models developed by previous students in Innoslate and in MP in order to inform the UAV MP modeling analysis for this SERC task.
SE0811	Thesis in Systems Engineering	This is a thesis course for students pursuing a systems engineering degree. PD21 students take this course in the last two quarters of the 721 curriculum to conduct independent research under the guidance of an advising team consisting of an advisor and a second reader. The primary thesis contributions resulting from this research are referenced as [11], [12] and [26] in Appendix A.

This section contains a version of work authored by John Quartuccio and Kristin Giammarco appearing as the chapter entited "A model-based approach to investigate emergent behaviors in systems of systems" in Engineering Emergence: A Modeling and Simulation Approach, edited by Larry Rainey and Mo Jamshidi. Boca Raton, FL: CRC Press Taylor & Francis Group, in press.

D.1. INTRODUCTION

A fundamental challenge in Systems Engineering is predicting the behavior of the operational system (Aizier et. al, 2012). Early identification of emergent behaviors—particularly those that are unexpected or unwanted—contributes to reduction of program cost and schedule risk (Auguston, 2009a), and enables failure analysis to promote mission success (Rambikur, 2017a). Studying system behaviors in advance of operations typically involves generating use cases using a graphical approach (as is done using architecture modeling tools and fault tree analysis (Rambikur, 2017b), an executable approach (as is done using simulation tools), or a combination of both. Creating executable models of systems increases human understanding of behaviors that emerge, as models can be executed over a number of conditions to study various outcomes. However, many of today's system behavior models are over-constrained (Auguston, 2010) (Auguston, 2012), suppressing behaviors that should be addressed prior to operation, and failing to expose the very behaviors we are seeking to prevent from occurring.

Monterey Phoenix (MP) is a Navy-developed systems engineering approach and tool with a demonstrated ability to automatically generate comprehensive sets of use cases containing many more emergent behaviors than other methods in use today (Auguston, 2009a; Giammarco, 2017a). The analysis value of these MP-generated use cases is the following: the scenarios not only contain wanted behaviors, but also unwanted behaviors including incorrect, hazardous, or otherwise undesirable behaviors in systems designs (Giammarco et. al, 2017b). Giammarco and Giles (2017a) summarize some examples of emergent behaviors that prompted the identification of new system requirements. These behaviors would have been difficult to find without using MP. The comprehensive sets of MP-generated use cases are scope-complete, which means they contain the exhaustive set of possible combinations of behaviors and interactions among the modeled systems up to a scope limit. Scope is the number of loop iterations permitted in the behavior models. The scope-complete aspect of the automatically generated use cases, further described in (Giammarco and Augston, 2013; Auguston et. al, 2015; Giammarco and Giles, 2017a), provides the ability to test the system model for the presence or absence of behaviors of concern up to the specified scope.

Prior to describing how MP works (section D.2), a significant discussion on the concepts leveraged by MP is presented for background and context. These concepts include abstraction, behavior, activity, events, modeling, simulation, system, system of systems, complex systems, and emergence.

D.1.1. BEHAVIORS, ACTIVITIES, ACTIONS, AND EVENTS

1. A **behavior** is a set of events or actions, typically leading to some observable end point.
2. An **event** has a beginning and end, and an order of precedence. Both the existence of a precedence relationship and the lack of a precedence relationship (or concurrency) are relevant relationships among events.
3. An **event** also has an inclusion relation that places the events within some hierarchy of other events.

4. An **action** is considered the same as an event.

5. An **activity** is a higher-level abstraction of an event. A process is an activity.

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

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

An event also has an additional relationship, *inclusion*. This relates the hierarchy of events; all events must have a root or source. One may think of a radio that would either send or receive a message. The event under consideration is the send or receive action, the operator is the radio, and the operand is the message. And so one can say that the send and receive events stem from, or are included in, the radio. Complex events may be described within multiple subordinate events that stem from a composite event. For example, there may be many functions necessary for the radio to send or receive, and so the operator and operand remain the same, but the send and receive events may be made up of many sub-events.

Further, actions and events are treated synonymously within this report, though when used in reference to the code to follow, events will be used exclusively. Furthermore, and more simply, an activity is a higher-level abstraction of an event. Following a defined process is an example of an activity.

D.1.2. ABSTRACTION, MODELS, AND SIMULATION

1. A **model** is an abstraction of the physical or instantiated system, developed in order to represent essential elements of the design.

2. The **model** must represent a complete set of behaviors, and analytically derive a means to enable intended behaviors, while restricting unintended behaviors.

3. A **simulation** is any instance of behavior extracted from the model.

The concept of an abstraction is critical to any discussion of a model. Simply put, a model represents only those functions necessary for a particular purpose. As an example, a training system may need to model a radio in order to provide instruction. However, the model radio does not need to perform the thousands of functions of an operational radio; the model simply has the look and feel of the operational system.

Many are familiar with the popular quote from Box that “all models are wrong, but some are useful” (Box, 1976). Buede developed a consistent thought as he defines a **model** as “any incomplete representation of reality, an abstraction” (Buede, 2009). He further elaborates that a model may take on the following forms:

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

Buede's key point is that the "essence of a model is the question or set of questions that the model can reliably answer for us" (2009). Though a model is an incomplete abstraction of reality, yet the model must enable a **complete set of answers to the questions of interest**. Thus in order to represent behaviors, a model needs to identify all aspects of these behaviors to include both positive and negative interactions.

Consistent with those definitions, consider the following: "A model is a representation of something. It captures not all attributes of the represented thing, but rather only those seeming relevant. The model is created for a certain purpose and stakeholders" (Weilkiti, 2013).

A **simulation** is any particular instance of a behavior model. Using the previous examples, a simulation of the wind tunnel aerodynamic model is one run of the wind tunnel, consisting of the initial set-up description and final results. The simulation of a physics-based computer model is one execution, based on a set of inputs, and the computed results.

D.1.3. A BEHAVIOR MODEL

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

A behavior model in the analytical or computational realm consists of some algorithm that employs a formal language to derive a set of events, representing essential interactions and relationships held within the model. This approach avoids the specification of parameters of the design for as long as possible; concentrated effort can evaluate the underlying and intrinsic interactions within the design and within the design interaction with other systems, the user, or the environment. The temporal (precedence) and hierarchical (inclusion) properties combine to enable concise behavior definition.

A formal language, based in set theory, is fundamental to the description of a behavior model. As opposed to a natural language, with possibilities for misinterpretation, a formal language is complete, consistent, and verifiable. Monterey Phoenix (MP) employs such a language, explicitly for the purpose of defining and modeling behaviors. As a lightweight formal method, models developed within MP rely upon investigation by the developer for verification in lieu of formal proofs. This approach encourages an interactive methodology for development.

Key Points:

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

D.1.4. A TAXONOMY OF COMPUTER-BASED MODELS

Within the construct of computer modeling and simulation of dynamic behaviors and interactions within a system, a proposed taxonomy of models and simulation including behavior models, system dynamics modeling, discrete-event modeling, and agent-based modeling, as summarized in Table 1 and discussed in the following outline (Borshchev, 2013; Buede, 2009). The latter three types require some level of parameterization, such as speed, range, timing, etc. in order to project the system dynamics. Behavior modeling produces a logically sound architecture and therefore it should be conducted prior to more detailed analysis.

Behavior models are developed at a high level of abstraction, prior to populating parameters to the model. Monterey Phoenix (MP) is a behavior model, employing lightweight formal methods, which develop all possible traces, or use cases, within a given scope of execution (Auguston, 2016). Behavior models focus on the essence of the interactions within a system and with external interfaces, producing a logically-sound architecture. MP is available for anyone to use at http:\firebird.nps.edu.

Discrete-event models are used for analysis of transactions such as those encountered in financial institutions, logistics and shipping companies, and other behavioral representations. Credit card transactions, commercial airlines transportation, and commercial cargo and shipping systems are examples of processes that are well suited for discrete event modeling. Many aspects of systems of systems are tied to such discrete events. The Discrete Event System Specification (DEVS) formulation was introduced by Zeigler (1976) enabling a mathematical representation of system dynamics based on event sequences.

System dynamic models are used to simulate classic time-based problems in engineering. Many of these models simplify the system dynamics to a set of first or second order differential equations, code these as difference equation with a variable or fixed time-step, and solve these equations using a numerical method such as Runge-Kutta or other applicable algorithm (Yakimenko, 2011). Most engineering disciplines use system dynamic models to study phenomena and their applications to include within dynamics of structures, particle dynamics, flight dynamics, computational fluid dynamics, and most scientific fields. Interestingly, systems theory as applied to other fields also gain benefit from this method, including economics, sociology, and psychology. MATLAB and Simulink (Yakimenko, 2011) and AnyLogic (Borshchev, 2013) are commercially available packages that model system dynamics. Higher order languages such as any instantiation of C or Java™ can also be used to develop these models.

Agent-based Models are more recently developed than system dynamic models and discrete event models. The agent-based formulation describes the individual parameters or variables within the agent, the expected interactions between agents, and the overall context of the environment including number of dimensions, number of agents, constraints of the problem. AnyLogic is a commercially available platform for agent-based modeling (Borshchev, 2013).

Hybrid models combine aspects of two or more types of models.

Table D.1: Types of modeling and simulation

Type	Applications
Behavior models logic-based	Early system architectures Cross-domain analysis of all behaviors
Event-based models sequenced by events	Production and manufacturing Transportation systems Logistics
Agent-based models sequenced by an agent	Organizational behavior Animal behavior Biological systems

	Crowd dynamics Search patterns
System dynamics models physics, time-based	Flight dynamics Weapon separation dynamics & trajectory Computational fluid dynamics Dynamic structural loading
Hybrid models	Autonomous platforms Combined effects

D.1.5. FURTHER DEFINITIONS PERTAINING TO SYSTEMS ENGINEERING

D.1.5.1. System

We use a general definition of a *system*, supported by International Council on Systems Engineering (INCOSE), International Organization for Standardization (ISO), International Electro- technical Commission (IEC), and Institute of Electrical and Electronics Engineers (IEEE). These definitions apply to "engineered" systems that have specifically and intentionally been developed by humankind, as opposed to natural systems (Kossiakoff & Sweet, 2003). From this perspective a **system** is defined as "a combination of interacting elements organized to achieve one or more stated purposes" (ISO/IEC/IEEE, 2015) where the elements may include "hardware, software, firmware, people, information, techniques, facilities, services, related natural artifacts and other support elements" (Pyster et. al, 2012).
Crawley et al. define a system as "a set of entities and their relationships, whose functionality is greater than the sum of the individual entities" (2015). The authors further indicate that this definition the system includes both "entities and their inter-relationships." This construct of a system has a natural extension to concept of Systems of Systems (SoS), in this case each entity is a system.

D.1.5.2. Model Based Systems Engineering

The INCOSE defines Model Based Systems Engineering (MBSE) as
"the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases." (INCOSE, 2007; INCOSE, 2010; INCOSE, 2015)

D.1.5.3. System of Systems

INCOSE further defines a *System of Systems (SoS)* as a system of interest "whose elements are managerially and/or operationally independent systems. These inter-operating and/or integrated collections of constituent systems usually produce results unachievable by the individual systems alone. Because an SoS is itself a system, the systems engineer may choose whether to address it as either a system or as an SoS, depending on which perspective is better suited to a particular problem" (INCOSE, 2015).
The Defense Acquisition Guide (DAG) adds context to this definition that is directly consistent with the ODUSD Systems Engineering Guide for Systems of Systems (Guide, 2008). The DAG reads as follows:
"Most DoD capabilities today are provided by an aggregation of systems often referred to as System of Systems (SoS). A SoS is described as a set or arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities. For complex SoS, the inter-dependencies that exist or are developed between and/or among the individual systems being integrated are significantly important and need to be tracked. Each SoS may consist of varying technologies that matured decades apart, designed for different purposes but now used to meet new

objectives that may not have been defined at the time the systems were fielded." (Defense, 2013, p. 167)

Dahmann et al. indicated that the system of systems framework and processes "push systems thinking beyond the traditional arena of new system development and acquisition to address the reality of today's system challenges of integrating and evolving existing systems to meet changing needs" (2009). Within the continuum of types of systems of systems, consistently, consideration has been given to basic categories of managerial control to include "directed, collaborative, virtual, and acknowledged" (Dahmann and Baldwin, 2008; Maier, 1996; (Maier, 1998), an emphasis on the importance of acknowledged systems have "recognized capability needs, management, and SE at the SoS level as well as autonomous objectives, management, and technical development approaches of the systems which contribute to the SoS capability objectives" (Dahmann et al., 2009).

D.1.5.4. Complex systems and emergence

Mitchell (2009, p. 13) defines a *Complex System* as "a system in which large networks of components with no central control and simple rules of operation give rise to complex collective behavior, sophisticated information processing, and adaptation via learning or evolution."

Taking into account the "self-organizing" aspects of unique "emergent" or collective behavior that can occur with no internal or external command and control of the individual elements or agents, Mitchell (2009, p. 13) also proposed a second definition of a *Complex System* as "a system that exhibits nontrivial emergent and self-organizing behaviors."

Complexity is closely aligned with Chaos Theory (Gleick, 1987; Strogatz, 2012; Waldrop, 1993) and its implications apply to many disciplines, including economics, psychology, sociology, medicine, and organization dynamics; within the traditional sciences of physics, chemistry, biology, and mathematics; and within engineered systems including network theory, computer science, integrated systems, and in general, systems of systems. Characteristics associated with complexity include emergence and self-organization. These aspects cannot be deduced by decomposing the system to its smallest components, but they are tied to the individual system behaviors, and are associated with interactions among the systems and their environment. Aspects that also contribute to those collective behaviors include, dependence on initial conditions, diversity within the population of agents, and the environment itself (Calvano & John, 2004; Miller & Paige, 2009; Mitchell, 2009, p.13; Strogatz, 2012; Waldrop, 1993). Considering these definitions, a complex system is special sort of system of systems that demonstrates a property known as *emergence*.

Zeigler (2016) identified the importance of both positive and negative emergence. Crawley et al. (2004) relate complex systems and emergence, including the concept of positive and negative as follows: "Complex systems have behaviors and properties that no subset of their elements have. Some of these are deliberately sought as the product of methodical design activity. While achieving these behaviors, the designers often accept certain undesirable behaviors or side effects. In addition, systems have unanticipated behaviors commonly called emergent. Emergent behaviors may turn out to be desirable in retrospect, or they may be undesirable."

Emergence is exhibited through collective behavior and self-organization of a system of systems that cannot be deduced by the investigation of an individual constituent system, element, or agent. This behavior depends not only upon the properties of an individual constituent system, but the collective behavior emanates from tightly coupled interactions among the individuals within the group (Calvano & John, 2004; Miller & Paige, 2009; Mitchell, 2009, p.13; Strogatz, 2012; Waldrop, 1993).

Maier defined an emergent property as "a property possessed by an assemblage of things that is not possessed by any members of the assemblage individually" (2015, p. 21). Maier (2015) further outlined types of emergence, summarized in Table D.2. Fromm (2005) had also developed a classification of emergence, summarized in Table D.3, based on the level of interaction of constituent systems within the

whole system. There exists some level of inconsistency between these descriptions; the former description may be better suited for an early assessment of predictability of the system response, while the latter description lends itself to a computational method that can enable automated classification based on the system response.

Table D.2: Types of emergence, derived from Maier (2015, p. 21-22).

Type	Emergent property mapping to modeling and simulation
Simple	Behavior readily predicted, model abstracted with lower complexity than the actual system
Weak	Behavior consistent with known properties of the system and readily and reproduced by simulation; however, system interactions must be included in the model
Strong	Behavior consistent with known properties, but unable to reliably predict where [or when] emergent properties occur
Spooky	Behavior inconsistent with known properties of the system components, even with model equivalent to complexity of the system

Table D.3: Classification of emergence, indicating a particular designation based on the level of interaction and feedback, developed by Fromm (2005).

Class	Title	Description
Type I	Simple	Interaction of systems producing some global response, but without feedback to the systems.
Type II	Weak	Inclusive of the interactions of Type I, but with feed-back from the global response to the systems.
Type III	Multiple	Inclusive of Type II, but with multiple systems.
Type IV	Strong	Inclusive of Type III, with the interaction of these multiple systems with each other creating additional global responses.

Having established this foundation in terms and concepts, the next section provides a general description of how MP works by introducing the MP event grammar and the MP-Firebird implementation (tool) available publicly at firebird.nps.edu.

D.2. Fundamentals of Monterey Phoenix

MP is a formal approach and language for modeling system behaviors and business processes. Its unique model partitioning strategy enables its novel scenario generation capability: component behaviors are separated from one another, and also component behaviors are separated from component interactions. The event trace generator computes all possible combinations of events modeled within each component, then filters out invalid combinations using the interaction constraints as rules defining behaviors that should be disallowed from a logical, simplification, or design point of view (Quartuccio). The fewer constraints imposed, the more behaviors appear in the model output. Each automatically generated event trace scenario represents one possible outcome of behavior, and it is these resulting event trace scenarios that are inspected for unwanted behaviors arising as a result of missing logical, simplification, or design constraints.

D.2.1. MP EVENT GRAMMAR

Using a concise pseudo-code language, MP employs event grammar rules to define the model. The two main binary relations used to construct event traces (particular instances of behavior) are ***precedes*** and ***includes***. Sequencing of events is done using a PRECEDES relation, and decomposition of events is done using an IN relation. An event grammar rule specifies the structure for a particular event type in terms of these two relations, and has the form

```
A:    right_hand_part;
```

where A is an event type name, the colon after A represents the IN relation (A *includes* right_hand_part), and the semi-colon indicates the end of a grammar rule. An event to the left of the colon may be root (having no parent) or composite (having a parent and at least one child), and events on the right hand part of the rule may be composite or atomic (having a parent but no children). A typical convention in MP is to associate each system or major component with a root event, as in the following example.

```
ROOT System_A:    Event_1
                    Event_n;
```

The above code specifies a root event called System A. Equivalently, we might call this event "System_A_Events", but as a convention, we abbreviate root events that contain other events belonging all to the same system as simply the system name. In this case, `System_A` includes two events `Event_1` followed by `Event_n`. A space or return carriage indicates a precedence relation between composite or atomic events within a root event.

A grammar rule may be composed of any combination of composition operations from Table D.4.

Table D.4 MP Event Grammar Concepts.

A: B C;	Ordered sequence of events (A includes B followed by C)
A: (B | C);	Alternative events (A includes B or C)
A: [B];	Optional event (A includes B or no event at all)
A: (* B *);	Ordered sequence of zero or more occurrences of event B in A
A: (+ B +);	Ordered sequence of one or more occurrences of event B in A
A: {B, C};	Unordered set of events B and C in A (B and C may happen concurrently)
A: {* B *};	Unordered set of zero or more occurrences of event B in A
A: {+ B +};	Unordered set of one or more occurrences of event B in A

Behavior is modeled in the event grammar as an algorithm for each component, describing the step-by-step procedure by which it achieves a well-defined goal. See the following MP code for an example event grammar. There are three distinct grammar rules in this code: one for the Car race (defined as a set of one or more Cars), one for Car (defined as the possible behaviors that a car does), and one for go straight (defined as possible different behaviors in going straight).

```
ROOT Car_race     :     {+ Car +};
     Car:       go_straight
                (*  ( go_straight | turn_left | turn_right )  *)
              stop;
```

```
     go_straight: ( accelerate | decelerate | cruise );
```

In this example, `Car_race` is a root event, `Car` and `go_straight` are composite events, and `turn_left`, `turn_right`, `stop`, `accelerate`, `decelerate`, and `cruise` are atomic events.

D.2.2. MODELING COMPONENT BEHAVIORS SEPARATELY

A key feature of MP analysis is the modeling of component behaviors separately. To maximize prediction of emergent system of systems behaviors, we model the behaviors of the system under design, the behaviors of external systems, and behaviors in the environment, separately. The MP event grammar is employed to create separate grammar rules for each component or system, as in the example grammar rules below.

```
ROOT System_A:    Do_System_A_actions;
ROOT System_B:      Do_System_B_actions;
ROOT Environment:   Do_Environment_actions;
```

D.2.3. MODELING DATA FLOW AS EVENTS

Because MP uses an event grammar, data inputs and outputs are not explicitly modeled in the same way they are in data flow-oriented languages. Instead, they are represented by actions (captured as events) that may be performed on that data. For example, a data input/output would be modeled implicitly in the MP events, and look like "Send_data" or "Receive_data" rather than "data," as in the code below.

```
ROOT System_A:    Send_data;
ROOT System_B:      Receive_data;
```

D.2.4. MODELING COMPONENT INTERACTIONS SEPARATELY

In addition to separating component behaviors from each other as in the `System_A` and `System_B` code example above, MP uniquely separates the definition of the behaviors within a system from the definition of the interactions of the events among multiple systems, the user, an environment (Auguston, 2016; Giammarco and Auguston, 2013). This model partitioning strategy allows the model developer flexibility and control in describing the intended behavior of the overall SoS through the addition and removal of interaction constraints. This is a missing concept in other modeling methodologies, in which system interactions are often found hard-coded – that is, tied to the specific instances of related actions in a given scenario – in activity diagrams and sequence diagrams. In MP, a COORDINATE statement is used to impose an interaction constraint upon events in different roots, separately from the component grammar rules. The code example below coordinates the `Send_data` and `Receive_data` events from the different roots in the code example above, since sending actions generally precede receiving actions (a logical constraint).

```
COORDINATE  $a: Send_data     FROM System_A,
                  $b: Receive_data  FROM System_B
            DO ADD $a PRECEDES $b; OD;
```

The literal interpretation of the code is an instruction to pick up each instance of `Send_data` and `Receive_data` present in each scenario, and add a precedence relation to each pair. When this constraint is added, any traces containing just one and not the other will be rejected, meaning in this case there would be no scenarios of sending but not receiving, or receiving but not sending. A rationale for rejecting these traces is provided in Appendix E, which describes such scenarios as anti-patterns to good architecture practice (see in particular E.3.3.5 and E.3.3.6).

Event sharing, where two or more systems have some events in common, is yet another way of behavior coordination. The SHARE ALL composition is used to specify events to be shared. Shared events may appear in the root event at any hierarchical level. For example, a constraint may be written to explicitly state that `System_A` and `System_B` share all instances of certain events when they occur. For example:

```
System_A, System_B      SHARE ALL   Common_event;
```

The `Common_event` must be present in each root event for which event sharing is specified. The below code shows this, and puts together all of the introduced concepts so far with a SCHEMA name. The schema is a user-defined name for the whole MP model.

```
SCHEMA Simple_Data_Flow

ROOT System_A:    Send_data
                    Common_event;

ROOT System B:      Receive_data
                    Common_event;

COORDINATE  $a: Send_data     FROM System_A,
                  $b: Receive_data  FROM System_B
            DO ADD $a PRECEDES $b; OD;

System_A, System_B      SHARE ALL   Common_event;
```

Event sharing is often used as a device for abstracting interactions that are not essential to the modeling effort. In this example, `Common_event` might be something like `Close_session`.

D.2.5. EVENT ITERATION

Composition operations for event sequencing, concurrency, optional events, alternate events, and recurring events were defined in Table D.1. Event repetition can be used to specify iteration of certain behaviors. For example, the following model demonstrates how to create a simple recurring message flow from a Sender to a Receiver. Figure D.1 shows an example of what one of the traces from this model could look like.

```
SCHEMA Simple_message_flow

ROOT Sender:      (* send *);

ROOT Receiver:    (* receive *);
```

```
COORDINATE  $x: send          FROM Sender,
               $y: receive     FROM Receiver
        DO ADD $x PRECEDES $y;       OD;
```

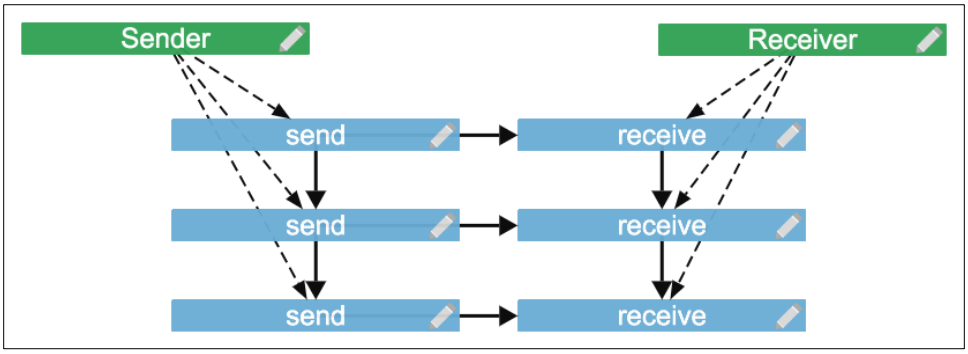


Figure D.1. An example composed traces for a simple pipe/filter architecture pattern. The green boxes represent root events, the blue boxes represent atomic events. The dashed arrows represent inclusion relations and the solid arrows represent precedence relations.

More examples can be found in research products [4] [5] [6] [8] [9]. The MP language is fully described in (Auguston 2016).

D.2.6. The Small Scope Hypothesis

The Small Scope Hypothesis (Jackson, 2006) postulates that most errors in computer code can be found within a relatively small number of iterations or execution cycles. Modeling objects, as opposed to behaviors, Jackson implemented this concept through the Alloy Analyzer tool (Jackson, 2002). The Small Scope Hypothesis and the associated Alloy Analyzer tool provided inspiration to apply this practice from the software domain in the systems engineering domain with Monterey Phoenix and the MP-Firebird tool (Giammarco, 2013). MP-Firebird, or Firebird, employs the Small Scope Hypothesis to find most errors in MP behavior specifications with relatively few loop iterations through the code. For example, the scenario depicted in Figure D.1 depicts three iterations of the send-receive events. Iteration in MP code is defined by loops `(*…*)` `(+…+)` that contain the events to be repeated, like `(* send *)` and `(* receive *)` in the case above. Events may occur zero times, one time, two times, etc., up to the specified scope. Since the model execution time grows exponentially as the scope increases, using the Small Scope Hypothesis is a practical concession we make when we do not need to provide a 100% behavior scenario coverage guarantee of all possible behaviors at *infinite* scope. Instead, MP provides a 100% behavior scenario coverage guarantee of all possible behaviors at a reasonably *small* scope. For example, if the loop in the model above Figure D.1 were permitted to run for an infinite number of times, Figure D.1 would grow infinitely tall, and it is very unlikely we will gain any more information running at say, scope 100, than we would running at scope 3. Therefore, we limit the scope at which we run the model, not only for practical reasons, but also because the small scope is usually sufficient to spot behavior patterns taking shape. Using the Small Scope Hypothesis keeps the run times brief for near-immediate user feedback that exposes a great number of errors or behaviors of concern without waiting for lengthy computations to complete.

D.2.7. EXHAUSTIVE SCENARIO GENERATION, UP TO THE SCOPE LIMIT

In MP, exhaustive scenario generation up to the scope limit provides a 100% coverage guarantee of all possible behaviors within the specified scope. A reasonable scope limit for most models is 3. Running an MP model at a scope higher than 3 will take longer, but may be desired to accumulate confidence and evidence for certain critical systems, such as the absence of certain behaviors of concern. *Assertion checking* is an automated process that can be used to detect known or suspected behaviors, especially at higher scopes and other cases where the number of scenarios may be far too large to undergo manual inspection.

Exhaustive, of course, also refers to the model content present, and like any model, cannot overcome errors of omission. In other words, MP does not provide entirely new and different events in the scenario output that were not already present in some form in the MP model. What it does provide is many permutations of the events that are present, but potentially not considered before they were explicitly laid out as possible event traces. Errors of omission may, however, be realized by the modeler upon inspection of the output. For instance, Nelson's example in (Giammarco & Giles, 2017a) demonstrates an error of omission that came to light by its stark absence against the backdrop of all other combinations of events, resulting in a model revision to correct the omission as well as a new requirement for a communications subsystem on an International Space Station resupply spacecraft. Realization of the omission error was a result of human cognition aided by the automated tool, neither of which realized the presence of the error without ability from the other.

D.2.8. HOW TRACE DERIVATION WORKS

In execution, MP generates all possible outcomes of the model. Within MP, these instances are called event traces (or use cases) and are based on the scope of execution of the model.

The event traces emerge as a result of a derivation process. It is guided by the grammar rules and composition operations created by the model developer. The event trace derivation determines the MP semantics. Trace derivation for schemas, roots, and composite events is performed top-down and from left to right following the event grammar rules. Composition operations act like "crisscrossing" derivation rules that may add new events and relations to the trace under derivation.

As John Hughes (1989) has put it in his influential paper: "The way in which one can divide up the original problem depends directly on the ways in which one can glue solutions together." Event grammar rules in MP provide a way of modularization, or dividing the system's behavior model into a hierarchy of component behaviors. Composition operations are separated from the grammar rules, and define interactions (or dependencies) between behaviors of components. This separation and supporting mechanism of "gluing" behaviors together are the core features of MP.

For the schema:

```
SCHEMA S

ROOT R1: …;
ROOT R2: …;
…
ROOT Rn: …;
```

event trace derivation starts with an implicit grammar rule

```
S: { R1, R2, … , Rn };
```

and deploys the composition operations interlacing the root rules and defined in BUILD blocks attached to the schema and to other event grammar rules. See (Auguston 2016) for a description of how to use BUILD blocks and other language features, and for a more complete description of the derivation process.

D.2.9. USING MP-FIREBIRD

MP is an executable architecture modeling approach and language. Figure D.2 shows the graphical user interface of the MP-Firebird tool, available publicly at https://firebird.nps.edu. Event traces (use cases, or examples of behavior) are generated by executing MP pseudo-code, or schema, performing the trace derivation process described in the previous subsection in order to represent all possible behaviors within the model. Each section of code is written and edited in the designated text-box. The schema is executed by the run "button," while the scope of execution controls the number of iterations computed in accordance with the schema. Upon execution, diagrams of an exhaustive list of traces are produced.

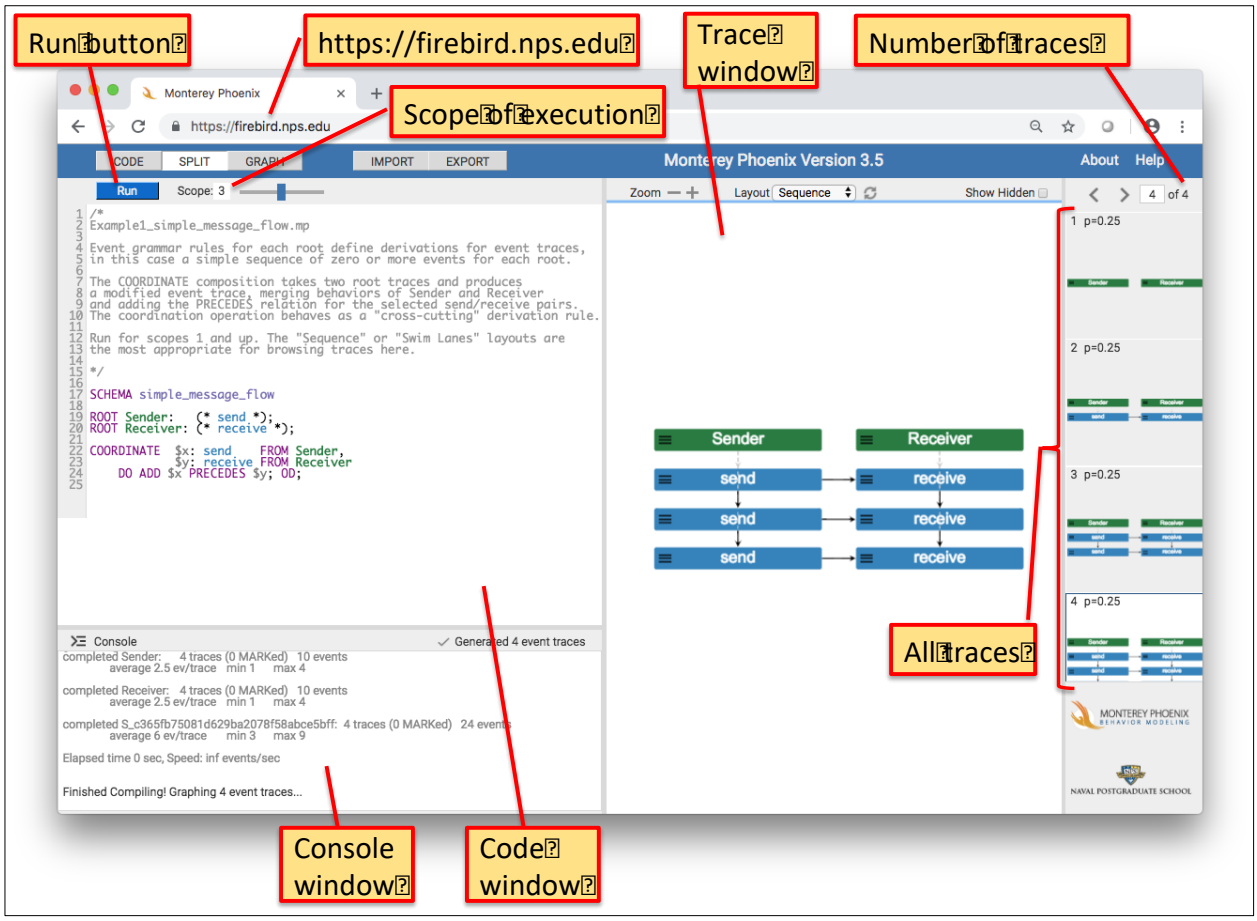


Figure D.2: The Monterey Phoenix (MP) modeling environment, illustrating the code window (left), scope of execution (top left), the current trace (right), and all traces produced by executing the schema (far right). MP-Firebird is available for anyone to use at firebird.nps.edu.

Events are visualized as boxes, and dependencies between pairs of events as arrows marked by the relation type (Figure D.3).

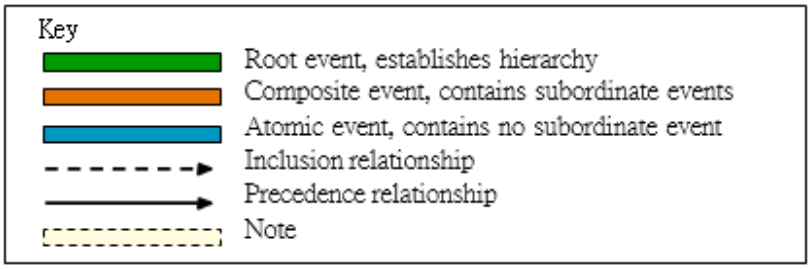


Figure D.3: MP diagram key.

Other MP-Firebird features currently deployed include:

- Syntax highlighting and real time syntax checking
- Import of user-defined code as well as readily accessible, preloaded examples (Import menu)
- Drag and drop manual arrangement of boxes and arrows on graphs
- Zoom in / out
- Node collapsing / expanding
- Node hiding / showing
- Edge repositioning
- Adjustable size navigation pane
- Sequence, Force, and Swim Lanes layouts (additional architecture views to be defined)
- Export high resolution images of full or user-cropped trace diagrams
- Export code only (.mp) or code together with customized diagrams (.wng)
- Keyboard shortcuts as described in the Help menu

To *browse example models*, open the Import menu and select any example under “Load example:”.
To *load a model* that you downloaded or saved to your hard drive, open the Import menu and select “Load .mp or .wng file:”.
To *start a new MP model*, simply load an example model and type directly over it, reusing as needed.
To *save a model* you developed, open the Export menu and select Code (for an .mp text file) or Code and Graph (for a JSON .wng file containing any adjustments you made to box and arrow positions).
To *save a picture of a trace*, open the Export menu and select Current Trace.
To *save only a portion of a trace*, open the Export menu, select Define Crop Area, and then use the selection tool to highlight the portion of the graph to be included in the export.

D.2.10. DEVELOPMENT PROGRESS ON INSTALLABLE MP IMPLEMENTATION

An experimental installable version of the MP implementation uses a Python user interface application to accept user input (MP code) and display the event traces generated from it.
Events and their dependencies are visualized similarly to MP-Firebird (Figure D.4). Features as of this final report include most features in MP-Firebird plus the following:

- Export high resolution images of full trace diagrams (active trace or all traces)
- User-defined and preloaded color schemes for boxes, arrows, and background colors
- Close MP model
- Stop trace generation
- Option to run trace generation locally or through remote server
- Status bar

Development on the MP implementation in Python is planned to continue to support integration with NAVAIR's MBSE modeling framework.

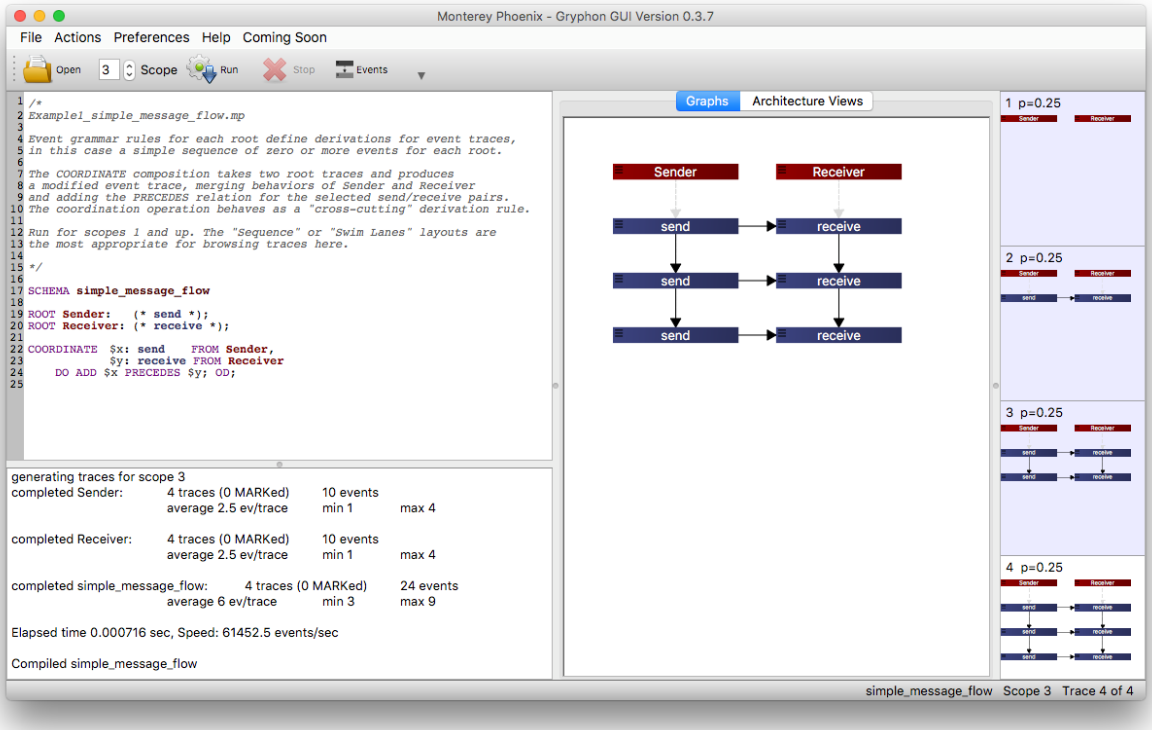


Figure D.4: The Python user interface for the installable MP implementation, illustrating the text editor and console panes (left), scope of execution (top left), the current trace (right), and all traces produced by executing the schema (far right). NAVAIR color scheme shown.

D.3. MP MODELING BEST PRACTICES

This section contains some MP modeling best practices, and conventions that have made MP modeling a pleasant and productive experience for its users. More advanced modeling heuristics are provided in Appendix G, section 4.3.

- **Read and refer to the MP Manual** to learn all of the language concepts available to you (Auguston, 2016). This appendix is a brief overview to get you started.

- Save/backup your MP file frequently.

- Sometimes it makes sense to copy/paste from, or directly edit, an existing MP model to create a new MP model, rather than start a new MP model from scratch. In this case, make sure that you are aware of and fix any copy/paste errors, and remove all irrelevant comments from the new model. Wrong comments are dangerous!

- Proceed with the MP model building incrementally. Start with a simple (bare bones) model of the main actors/behaviors. Add actors (root events), events, and coordination for them one at a time. Introduce composite events to encapsulate the logically complete actions. Run your code with scope 1 to weed out obvious errors.

- After each update, run the MP model to detect/fix coding errors and omissions in the MP code, and to decide about the appropriate scope, which is an essential aspect for non-trivial MP models. This may include adding specific iteration scope descriptions for selected iterators, like `(*<1..4> A *)` to steer the number of iterations. This way you can get around the scope limits imposed by Firebird, if needed.

- Style practices. Application of the following conventions may seem like a waste of time at first, but it is not. Standard conventions greatly increase the speed of comprehension and provide structure to minimize ambiguity in natural language. Whatever the naming conventions you use, consistency is paramount.

    - Always begin every model with some introductory comments. Include at least the name of author, date, and short description of the purpose of this MP model.

```
/* File_name.mp
Start with a brief description of the model purpose, and how to use it.
Also include any interesting analysis highlights or results.

created 2017-09-30 by J.Doe
modified 2017-09-30 by K.Doe updated the comments, corrected typos in the
model
*/
```

    - Use indentation to emphasize the order of events and main control structures (alternatives and iterations). Try to have one event name per line of MP code. Usually we read MP code top-down, and it always helps when while reading the code we can figure out what will be the order of events evolving from that code.

    - Use indentation and blank lines to make the MP code more readable, and to help readers follow the top to bottom stream of events. This practice will become very

critical when the size of the MP model starts to grow. Use blank lines to separate composite event definitions.

- Make sure that event names are readable and informative. In many cases event names will be pseudo-code statements describing activities or actions within the system and its environment.
- MP is case sensitive. In all cases, use an underscore to separate words of the phrase. This is not required by MP, but is strongly suggested. Use of CamelCase is not recommended.
- MP keywords are UPPERCASE in most instances with a few exceptions. (Refer to keywords listing in Section 7 of (Auguston, 2016) – the MP manual). Therefore usage of ALL_CAPS for event names is not recommended, in order to avoid confusion.
- Most events in a model are usually actions, but some high level (composite) events can be equated with components, processes, states, and even time phases. One can use good language grammar to help a reader determine which type of these event types you are representing:
  - Choose a noun-oriented name for an event that represents a **thing**, component, object, system, person, or organization (e.g., Operations_Team, UAV). Use title case for capitalization.
  - Choose an imperative present-tense verb or verb phrase for an event name that represents an **action** (e.g., Provide_command, Analyze_object). Use sentence case for capitalization.
  - You may choose an indicative past-tense verb or verb phrase to represent zero-duration or **instantaneous events** (e.g., Authorization_received, Object_detected). This format can be especially useful for naming shared events. Use sentence case for capitalization.
  - Choose an indicative present continuous verb or verb phrase for an event name that represents a **state** (e.g., Conducting_Maintenance, Powering_Up). Use title case for capitalization.
  - Choose a process-oriented phrase for an event name that represents an **activity**, **process** or **time phase** (e.g., Mission_Preparation, On_Station). Use title case for capitalization.
  - Root events are often named for the component that embodies the event pattern listed on the right hand side of its grammar rule. Composite events may be named for high-level processes that contain a sequence of composite or atomic events, or for time phases that contain a sequence of events.

- Validation begins when MP model starts to run smoothly. Now it is time to select the graph visualization mode (Sequence, Swim Lane, etc.) and to start browsing generated scenarios/use cases with respect to how well it captures your or your stakeholders' intent.

- During inspection on Firebird using the Sequence view, you may need to move some boxes to see all of the inclusion relations. The graphing algorithm currently used stacks the inclusion relation behind precedence relations, so they can be hidden.

- Though the relationships are correct, the current graphing algorithm on Firebird does not always stack events inside composite events in the correct position on the artificial timeline. If you have orange composite events in your traces and some events appear out of order, move the misplaced event to the correct position on the "timeline." The arrows are correct, as is the MP output file, but in the current implementation of the Firebird GUI, the box placement sometimes is not.

- If you move any boxes, to save your changes you need to export a .wng file.

- The number of generated traces may start to grow into hundreds and thousands. It becomes cumbersome to find traces satisfying particular properties, for instance, you'd like to see whether there are traces containing the event `Bell_Rings`. To help, MP has IF and MARK operations, and adding to your model something like

```
IF #Bell_Rings > 0 THEN MARK; FI;
```

will mark traces satisfying the condition, and make it easier to find them in the trace browsing scrollbar.

- After the initial draft of the model is stabilized, add incrementally more actors/behaviors, repeating the test/debug activities after each increment.

- Insert comments in your MP code explaining your major design decisions and rationale for MP constructs used in the code. The best comments answer the question "Why?", e.g., why this alternative or iteration has been placed in this particular position in the model, or why this coordination construct is here.

- Models are built by humans and usually need testing and debugging. For this purpose MP has CHECK construct for assertion checking (or for counterexample rendering) and SAY clause for annotating generated traces with messages. It is similar to the print statements used for debugging source code in traditional programming languages. After all, MP is executable modeling language. SAY clauses may provide answers to queries about number of events of interest already assembled into the trace segment, or indicate reaching certain point in the derivation process. This can be done both in BUILD blocks and in the schema's code. The MARK construct combined with IF composition operation makes it possible to highlight traces of interest for testing/debugging purposes.

- MP is not a programming language, like C, C++, or Java. It is a behavior specification language. MP grammar rules are supposed to define all behaviors you want to have. So, if you want to describe behaviors with and without `Bell_Rings` event, make sure that your event grammar contains all alternatives - if the grammar does not contain a behavior you want to have, there is no way how you will obtain it. Now, it may contain also many unwanted behaviors, but we can weed them out using ENSURE, COORDINATE, and SHARE ALL. Consider these constructs as a

powerful event trace filters and devices for identifying important requirements. These are the principles of writing MP code.

D.4. A METHODOLOGY FOR GETTING STARTED WITH MP

There are six steps in a proposed methodology to build behavior models that support SoS requirements development and analysis (Quartuccio). The first four are presented in this last section of Appendix D, and the last two are presented in the first section of Appendix E.

D.4.1. STEP 1: DEVELOP A NARRATIVE OF THE BEHAVIOR

It is typical that the model developer is not the end-consumer of the model and associated analysis, and so SME input is critical to the process. The SME may be an end-user of the product, the funding sponsor of the project, or perhaps the developer of a legacy or predecessor system. The model developer begins the methodology by interviewing the relevant SMEs necessary to formulate a complete, analyzable, internally consistent, and elegant, in accordance with qualities of great models (Qualities, 2017).
Criteria: This step is complete when the SMEs have described the relevant behaviors, constraints, assumptions, and limitations of the SoS. A written and approved document is the recommended practice, especially if significant resources are spent in order to develop the model.
Decision: The developer needs to decide to proceed to next step, repeat this step, or go back to a previous step.
Proceed if documentation provides an internally consistent starting point, then proceed with the model development. Further input will be needed at the next steps, and so the description may be incomplete during the first cycle of the methodology.
Repeat if the description is not consistent in its description and definitions; clarification will be necessary.
Go Back if the SME input cannot be reconciled; consideration needs to be given to address conflicts with sponsoring authority as necessary.

D.4.2. STEP 2: IDENTIFY MP EVENTS

A simple example is used throughout each of the next sections in order to illustrate how to describe events in MP. This model includes two root events A and B, with corresponding atomic events “task a” and “task b.” Root events are typically the environment, a user, a system, or a component of the system. Subsequent events are typically actions taken by the root.
The model was executed with scope of two, producing two traces. Trace 1 has a single atomic event for both roots A and B, while trace 2 has two atomic events for root A and a single atomic event for root B.

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

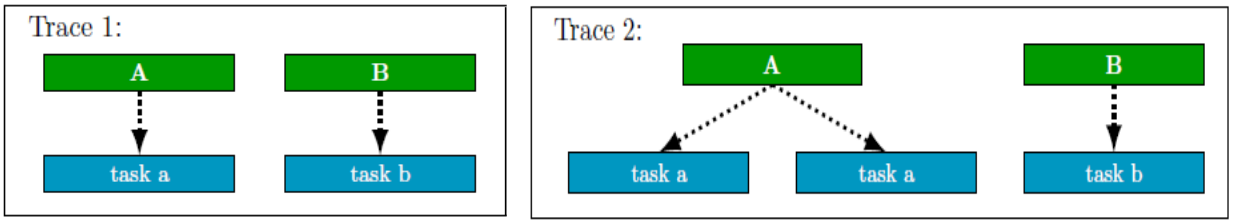


Figure D.5: A simple MP model illustrating root events A and B, with corresponding atomic events "task a" and "task b" (top). The inclusion relationship is represented with a dotted line. The model was executed with scope of two, producing two traces (bottom). Trace 1 has a single atomic event for both roots A and B, while trace 2 has two atomic events for root A and a single atomic event for root B.

The code for this model used the curly brackets `{ … }` to indicate concurrent events, and the plus sign, "+," surrounding `task_a` indicates that the number of events is at least one and up to the scope of execution. Since this schema was executed at a scope of two, the output includes a trace with one "task a" event and a second trace with two "task a" events. Both traces have a single "task b" event. Each root description ends with a semicolon, ";".

Criteria: This step is complete when the relevant environment, users, systems, and components are defined as root events, and all functions are captured as atomic or composite events.
Decision: Proceed to next step, repeat this step, or go back to previous step, as follows:
Proceed when coordination with the SMEs affirms that all events are captured.
Repeat if the SMEs identify significant errors.
Go Back if fundamental problems exist in the narrative.

D.4.3. STEP 3: IDENTIFY COORDINATION

Coordination establishes precedence relationships between events within roots A and B. In the simple illustration we can establish that in all cases, "task a" precedes "task b", as illustrated in Figure D.6. The "DO...OD" command establishes a loop for asynchronous coordination so that all occurrences of "task a" precede "task b." Events may also be shared by multiple roots using the "SHARE ALL" command.

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

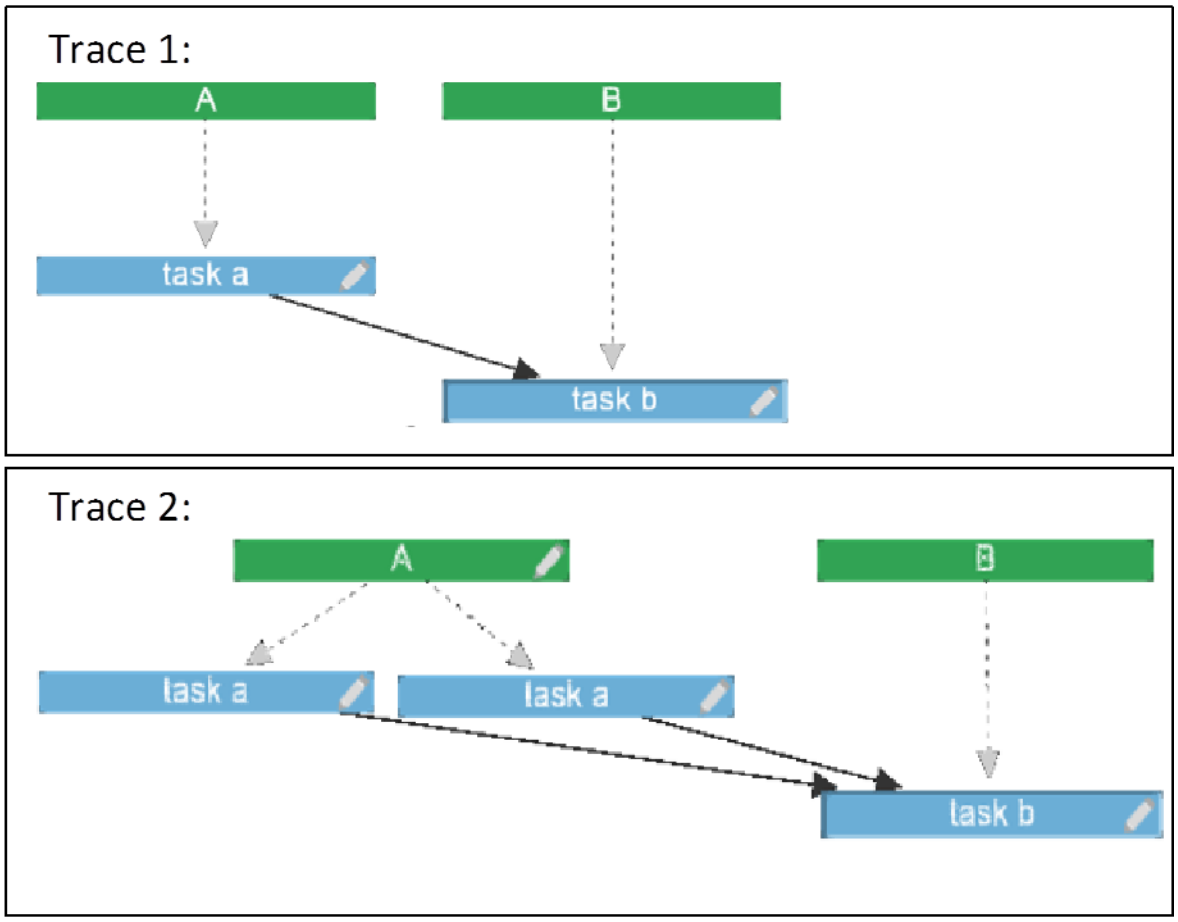


Figure D.6: A simple MP model with coordination between atomic events from root A and B. Coordination from task a to task b for both traces are illustrated as solid lines, representing precedence relationships. The dotted lines are inclusion relations, so instances of task a are unordered. As such, all occurrences of task a must be accomplished before task b.

Criteria: This step is complete when coordination across all root events are complete.
Decision: to proceed to next step, repeat this step, or go back to previous step.
Proceed if the developer and SMEs agree with the coordination structure.
Repeat if errors are found.
Go Back if the root events need to be restructured.
MP separates the behavior definition within a system (modeled as a root event) from the interaction among the SoS (modeled as multiple root events).

D.4.4. STEP 4: DEFINE CONSTRAINTS

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

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

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

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

Implementing constraints for our simple example: if the schema were executed without the coordination of precedence relationships at the top of Figure D.6, the developer would find the occurrence of a trace shown in Figure D.7, where "task b" occurs without a logical predecessor of "task a". Examples include a read function that cannot occur before a write function, or a receive function that cannot occur before a send function. Clearly in these cases, a logical-type of constraint is needed for the model. The code in Figure D.8 ensures that the number of "trace a" events is greater than the number of "trace b" events.

Examples of each type of constraint are available in Appendix A, research product [8].

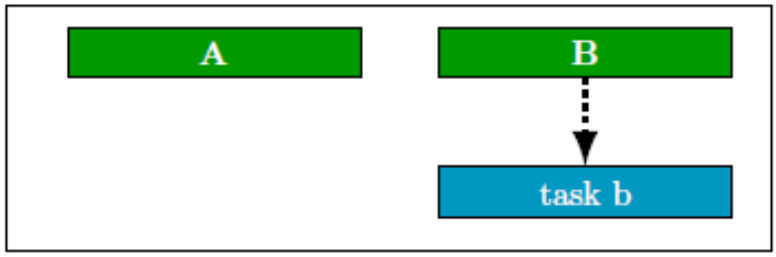


Figure D.7: A simple Monterey Phoenix (MP) model with a missing constraint, allowing a trace in which "task b" exists without a corresponding "task a."

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

Figure D.8: MP code for a logical-type constraint on the simple model

Criteria: This step is complete when all logical, simplification, and design constraints are implemented in the schema.
Decision: Proceed to next step, repeat this step, or go back to previous step.
Proceed when the developer and SMEs agree that the constraints are an appropriate representation.
Repeat if constraints are missing or overly used, restricting desirable or actual behavior.
Go Back if any part of the event structure needs to be reworked in order to satisfy the needed behaviors.
Appendix E continues the description of this method with Steps 5 and 6.

D.5. SUMMARY

The following are fundamental concepts of Monterey Phoenix (MP):

- MP models behaviors of a design, as the developer systematically defines the events, adds temporal and hierarchical interactions, applies constraints, evaluates the results, and adjusts the definition as needed.
- MP builds a model at a high level of abstraction, prior to adding parameters to the system description. Evaluating the model with MP early in the development enables the use of a mature structure for follow-on simulations employing event-based models, agent-based models, time-based physical models including system dynamics, or hybrids of these methods.
  - MP employs a powerful language, or pseudo-code, that formally describes the behaviors within the system. The complete model is referred to as a schema.
  - All blocks in the system are considered as events.
  - A root event initiates a hierarchy of subordinate events. These subordinate events are established by an *inclusion* relationship with the root event.
  - Events also exhibit *precedence* relationships both within a root hierarchy and coordinated among multiple root hierarchies.
  - MP separates the definition of behaviors within the system from interaction with other systems, users, or the environment through coordination of inter-related events.
  - Atomic events are comprised of no constituent event, meaning that these events have a parent in the hierarchy, but no child events.
  - Composite events are comprised of one or more constituent events, meaning that these events have a parent in the hierarchy and at least one child event.
- MP applies the Small Scope Hypothesis (Jackson, 2006) such that most issues in the model are found within relatively few iterations.

- Execution of the MP schema (model) produces all possible outcomes within the scope of execution, thereby enabling a means to identify emergent behaviors. As seen in Appendix E, these results expose patterns of behavior leading to the need for analytic tools to interactively identify these patterns, both positive and negative.

This appendix captures preliminary work in cataloging reusable architecture model patterns and anti-patterns of various types. First, the steps for documenting Monterey Phoenix (MP) behavior model patterns is described with references to example MP behavior patterns. Next, a crosswalk of MP, System Modeling Language (SysML), and Lifecycle Modeling Language (LML) behavior model patterns is presented. Finally, some general anti-patterns to be avoided in architecture modeling are presented in the language of the Department of Defense Architecture Framework (DODAF) meta model.

E.1. IDENTIFICATION OF PATTERNS IN MP BEHAVIOR MODELS

This section contains a version of work authored by John Quartuccio and Kristin Giammarco appearing as the chapter entited "A model-based approach to investigate emergent behaviors in systems of systems" in Engineering Emergence: A Modeling and Simulation Approach, edited by Larry Rainey and Mo Jamshidi. Boca Raton, FL: CRC Press Taylor & Francis Group, in press.

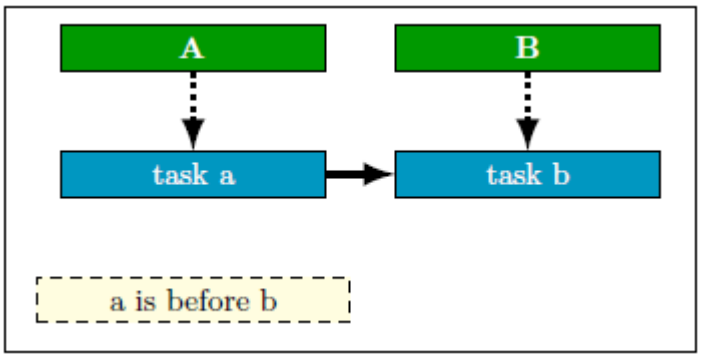
E.1.1. STEP 5: IDENTIFY PATTERNS

Appendix D describes four MP model construction steps that are prerequisite steps to identifying patterns in the MP results. Readers should become familiar with those steps prior to using this section of the report.

Patterns in MP behavior models can be identified using MP's assertion checking functions along with the "MARK" and "SAY" commands. Identifying patterns in MP models has several purposes that are critical to the system developer. Some ways these patterns can be used include:

- Quickly investigate large sets of automatically generated event traces, many of which may contain hundreds, thousands, or tens of thousands of traces.
- Ensure that all generated traces contain only expected output, and are absent of instances of uncontrolled emergence.
- Develop a repository of effective architectures that can be employed in future designs, including instances of desired emergence, thereby capturing positive lessons learned.
- Develop a repository of ineffective or problematic architectures, including instances of unwanted emergence, thereby capturing negative lessons learned.

Figure E.1 illustrates how behavior pattern detection can be done using MP. The code listed below the graph checks for scenarios where "task a" from a System A occurs before "task b" in a System B. If the condition is met, the trace is marked and annotated with the "SAY" command. Each of these checks can be formulated as templates of unique behavior in the model.




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

Figure E.1. Top: A simple MP model showing an example task_a that precedes task_b. Bottom: MP code for detecting behaviors such that each trace that satisfies the condition that "a is before b" is marked and labeled. This capability establishes the means to identify templates, or patterns, in the results.

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

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

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

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

Go Back if additional constraints are needed in the model.

Examples of preliminary MP behavior model templates are contained in Appendix A research products [8] and [13]. Long term objectives include creating a repository of these templates to enable re-use of working and validated architectures, as well as abandoned ineffective or problematic architectures best avoided.

E.1.2. STEP 6: EVALUATE THE RESULTS

Identifying and cataloging behavior patterns is a major step in and of itself, but is still the tip of the iceberg in behavior modeling research. Given the behavior patterns and other results available from executed MP models, several types of analyses may be of interest to the developer, such as:

- Determining the probability of occurrence of any trace or outcome
- Generating a summary of all interactions in the form of a Design Structure Matrix (DSM) or N-squared Diagram
- Computing the throughput of data flow

- Determining the sequencing and timing of events
- Acertaining requirements for computing power

Analysis of probability of each trace may be straight-forward for a very simple model, but the developer needs to ensure that the fundamental principles of statistics are followed closely. Not all architectures lend themselves to a viable solution. As described in Appendix A research product [9], the authors worked out certain conditions where an MP model can be used to define a Bayesian belief-network, and then probabilities of each trace could derived from a given set of assumptions. The conditions include the following:

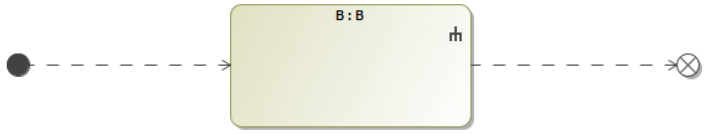
1. Each trace is an instance of the architecture and is represented as a directed graph without loops. This is a given for MP.
2. Each trace has the same topology, such that a consistent Bayesian belief network may be applied to all traces. This is not typical for many models, since the scope tends to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

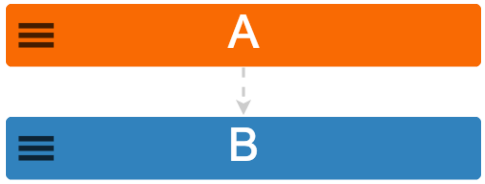
Other analysis uses of MP models and the behavior patterns contained therein is a topic of future work, subject to sponsor interest.

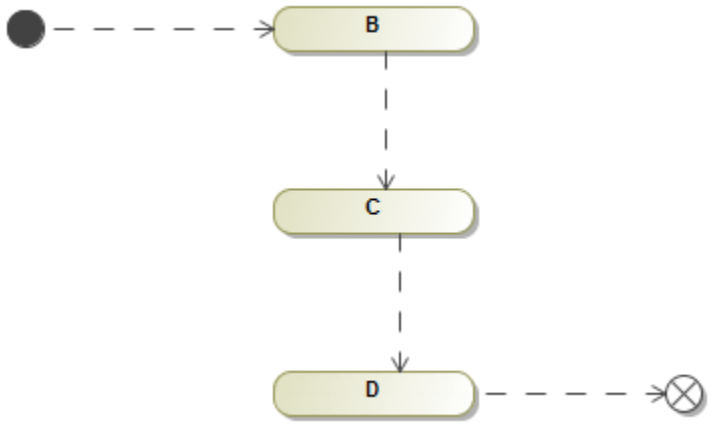
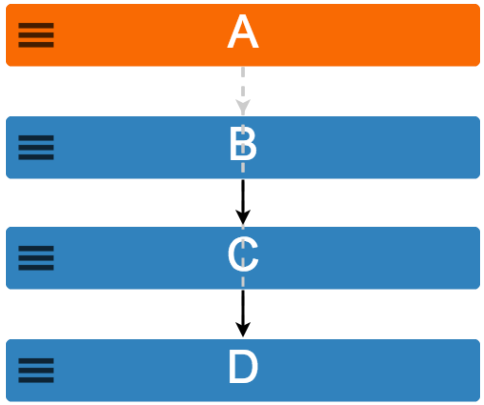
E.2. A Crosswalk of MP, SysML, and LML Behavior Model Patterns

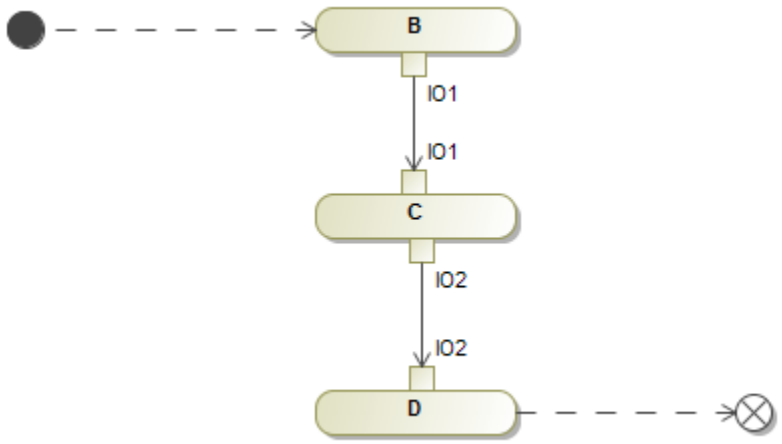
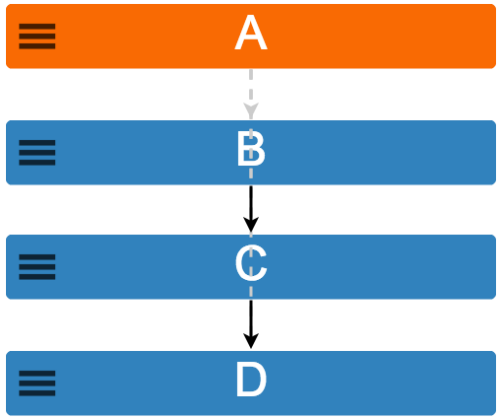
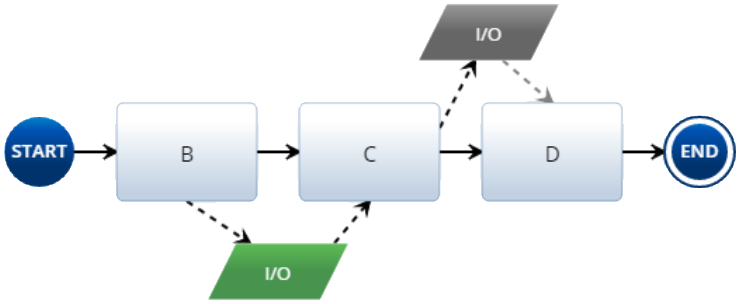
The table contained in this section are a result of a preliminary crosswalk performed among simple logical constructs available in several different modeling languages including SysML, LML and MP. The mappings provide the beginnings of a formal specification that may eventually inform automated conversion from graphical languages into MP. NPS PD21 student Ernie Lemmert contributed the SysML mappings to MP code as part of his ongoing thesis research.

Table E.1. Crosswalk of MP, SysML and LML Behavior Model Patterns

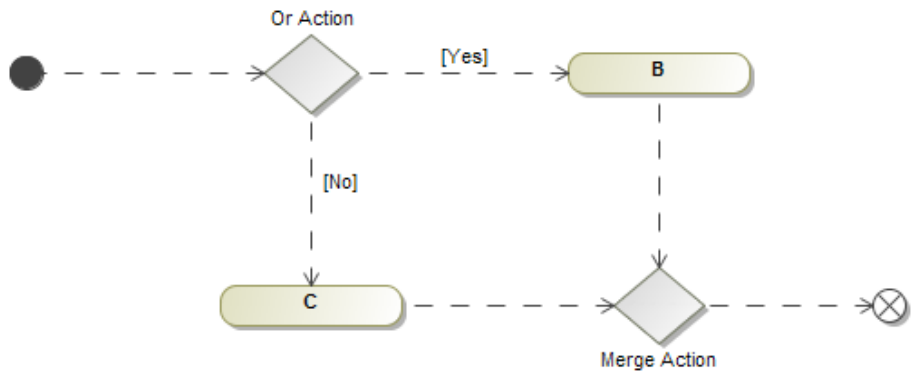
<table>
<tr><th>Activity Diagram</th><th>Corresponding MP Code and Graph(s)</th></tr>
<tr><td>SysML:
In MagicDraw, a node “A” has a decomposed diagram named “B”.
 
</td><td>/* A includes B */

A: B;
 
</td></tr>
<tr><td>LML: A decomposed by B.
In Innoslate, “A” is the name of the diagram containing “B”.
 
</td><td></td></tr>
<tr><td>SysML:</td><td>/* A includes: B followed by C followed by D. */</td></tr>
</table>

Activity Diagram	Corresponding MP Code and Graph(s)
B, followed by C, followed by D. 	A: B C D; /* Precedence may be denoted by a space between events, or a carriage return. The following rule is equivalent to the one above. */ A: B C D ; 
LML: B, followed by C, followed by D. 	

Activity Diagram	Corresponding MP Code and Graph(s)
SysML: B, followed by C, followed by D, shown with SysML Pins/ObjectFlow  	/* A includes: B followed by C followed by D. */ A: B C D; /* Data flow is not a separate concept in MP event grammar. For example, there is no separate concept for items like “status report”. The data flow around a status report would be modeled as “Send_status_report” and “Receive_status_report” with a plain precedence relation between these two events. */ 
LML: B, followed by C, followed by D, shown with required input/outputs (green triggers) and optional input/outputs (gray).  	
Activity Diagram	Corresponding MP Code and Graph(s)

SysML:
B or C. The “Or Action” can have custom named branches.



/* A includes: B or C */

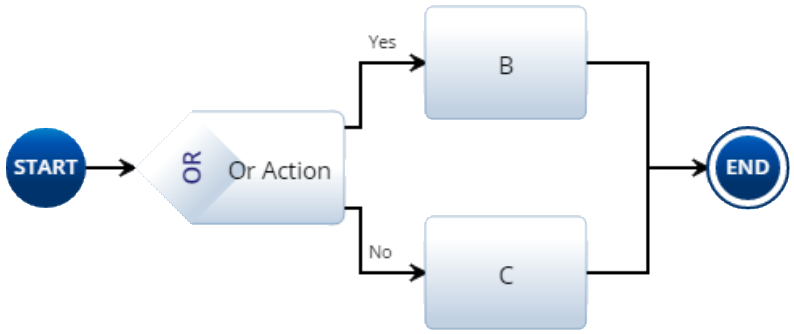
A: (B | C);

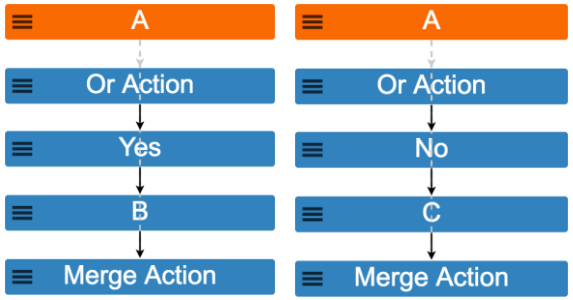
/* An alternate expression can be used to name the specific decision point branches as events in their own right.

A includes: Or_Action, followed by B if Yes or C if No, then Merge_Action. */

A: Or_Action
(Yes B | No C)
Merge_Action;

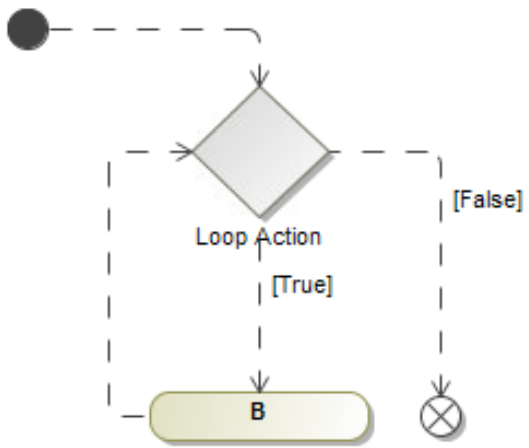
LML:
B or C. The “Or Action” can be a named block phrased as a question (with Yes/No branches as in the example) or a decision point / event with custom named branches.



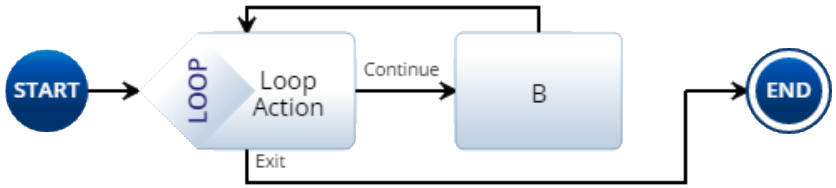


/* Whether or not to include the action and branch names as separate events in MP can be based on user preference. */

Activity Diagram	Corresponding MP Code and Graph(s)
SysML: B occurs for as long as some predefined condition holds true.	/* B occurs zero or more times */

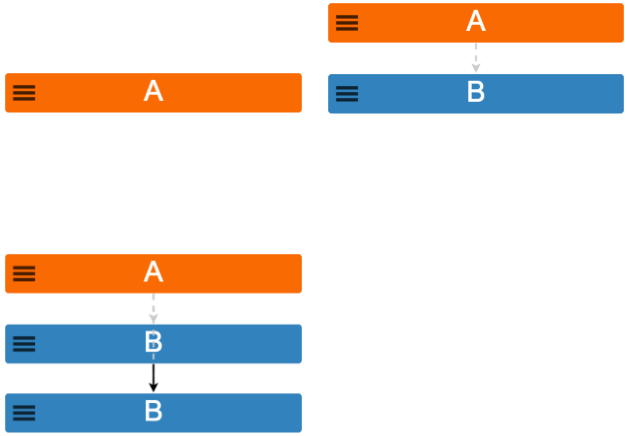


LML:
B occurs for as long as some predefined condition holds true.

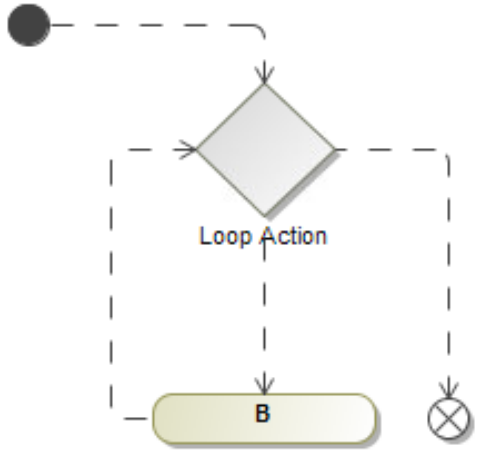


A: (* B *);

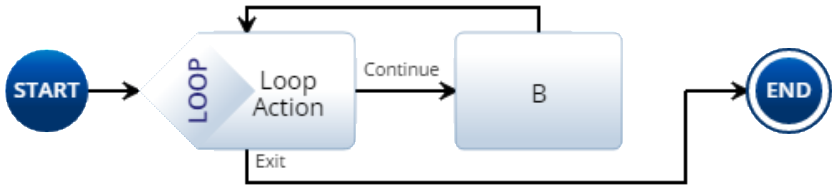
/* The model's global scope setting controls the number of loop iterations. For example, a scope of 2 will include all scenarios where B occurred zero, one or two times (illustrated in the three distinct scenarios below). */



Activity Diagram	Corresponding MP Code and Graph(s)
SysML: B occurs for as long as some predefined condition holds true.	/* B repeated between m and n times */ A: (* <m..n> B *);



LML:
B occurs for as long as some predefined condition holds true.



Activity Diagram

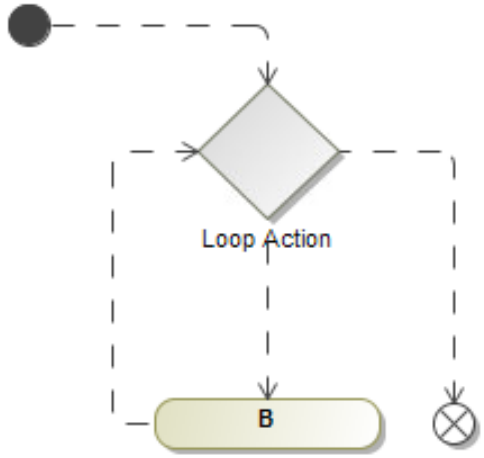
SysML :
B occurs for as long as some predefined condition holds true.

/* The local scope setting used here can control the number of loop iterations separately from the global scope. For example, a local scope of 1..2 will admit only scenarios where B occurs once or twice (illustrated in the two distinct scenarios below). */

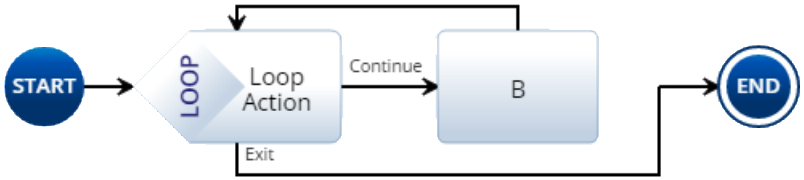
Corresponding MP Code and Graph(s)

/* B occurs one or more times */

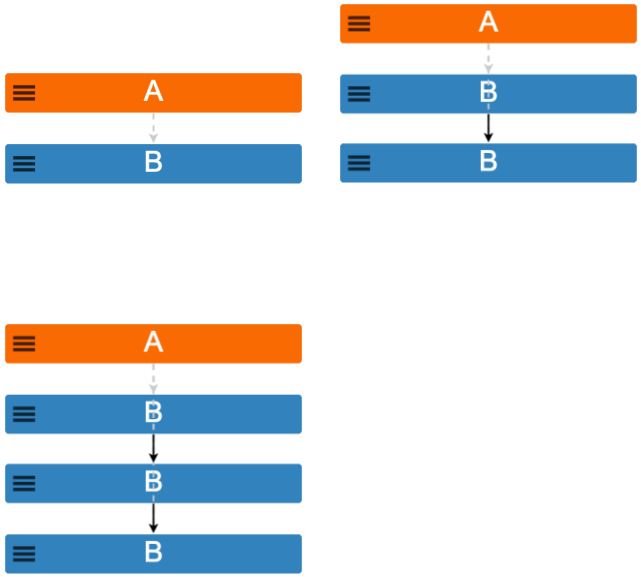
A: (+ B +);



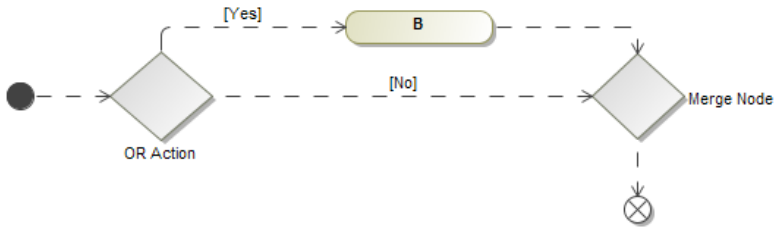
LML:
B occurs for as long as some predefined condition holds true.



/* The iterator plus composition requires at least one instance of the event(s) contained. For example, a scope of 3 on this model will include all scenarios where B occurred one, two, or three times (illustrated in the three distinct scenarios below). */



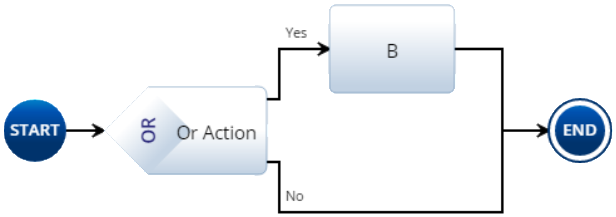
Activity Diagram	Corresponding MP Code and Graph(s)
SysML: B occurs if the OR Action outcome is Yes, or does not occur if the OR Action outcome is No.	/* B is optional */ A: [B];

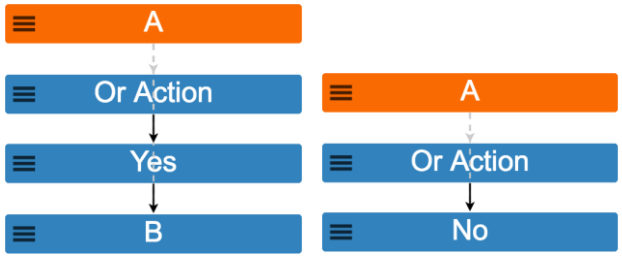


/* Alternatively, Or_Action is followed by Yes then B, or No */

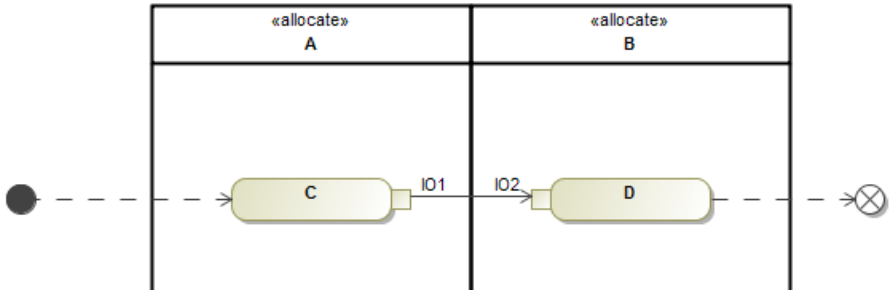
LML:
B occurs if the OR Action outcome is Yes, or does not occur if the OR Action outcome is No.

A: Or_Action (Yes B |
No);

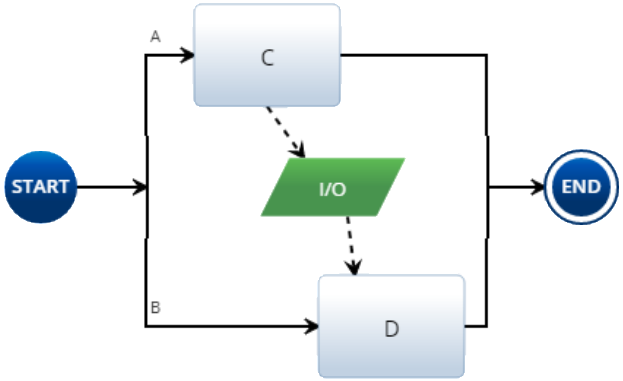




Activity Diagram	Corresponding MP Code and Graph(s)
SysML: A and B are components, each with its own swim lane. Activity C is allocated to A, activity D is allocated to B, and C precedes D.	/* A and B are root events (components). Event C is in root event A, event D is in root event B. */ ROOT A: C; ROOT B: D;



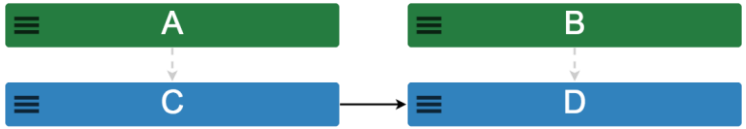
LML:
A and B are assets, each with its own parallel branch.
Action C is allocated to A, action D is allocated to B, and C precedes D.



Activity Diagram

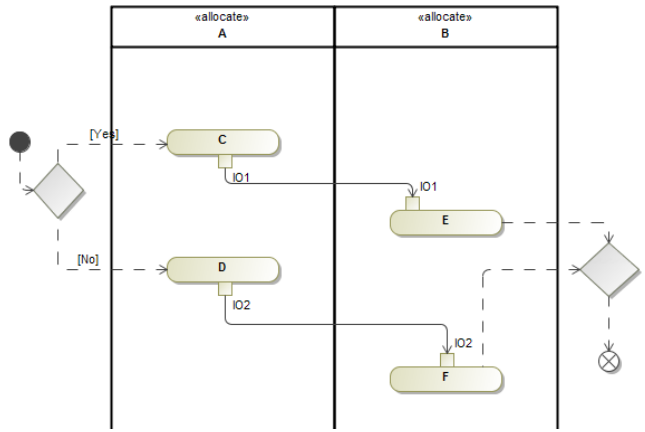
SysML:
A and B are components, each with its own swim lane.
Activities C and D are allocated to A, activities E and F are allocated to B.
C precedes E and D precedes F. If a Yes decision is made, C will occur, if not, D.

```
/* The constraint that C precedes D is added using a coordinate
composition since the events are in different roots. */

COORDINATE $x: C FROM A,
        $y: D FROM B
    DO ADD $x PRECEDES $y;
    OD;
```

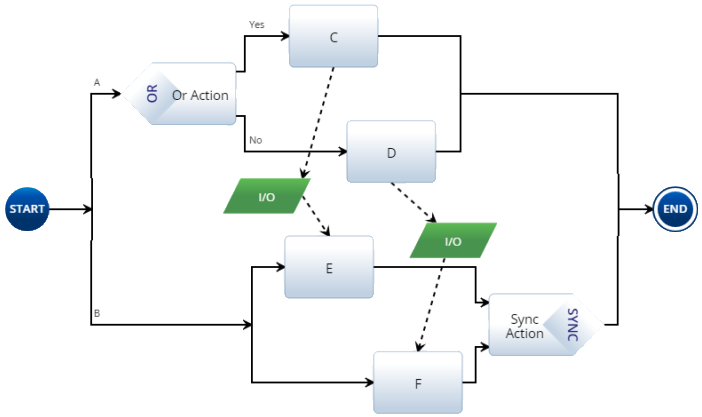


Corresponding MP Code and Graph(s)

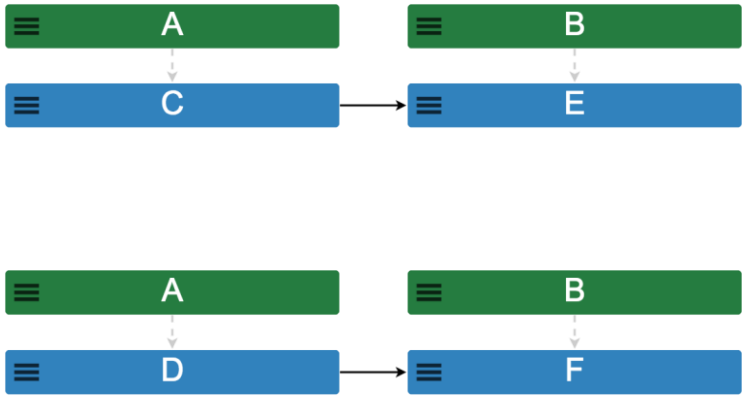
```
/* A and B are root events.
Event C or event D occurs in A, and event E or event F occurs in B.
*/
```

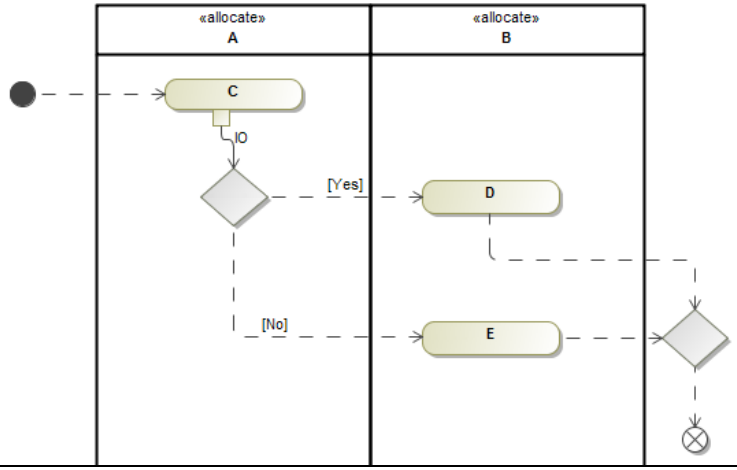
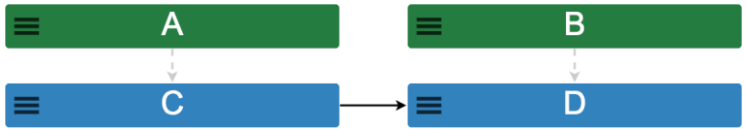
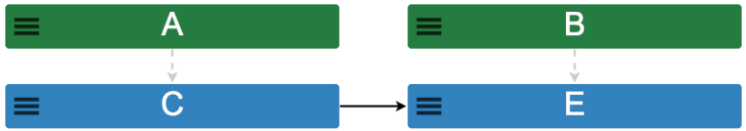
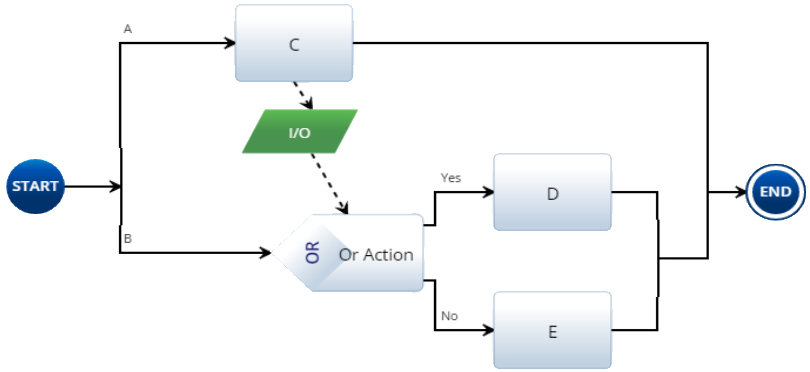


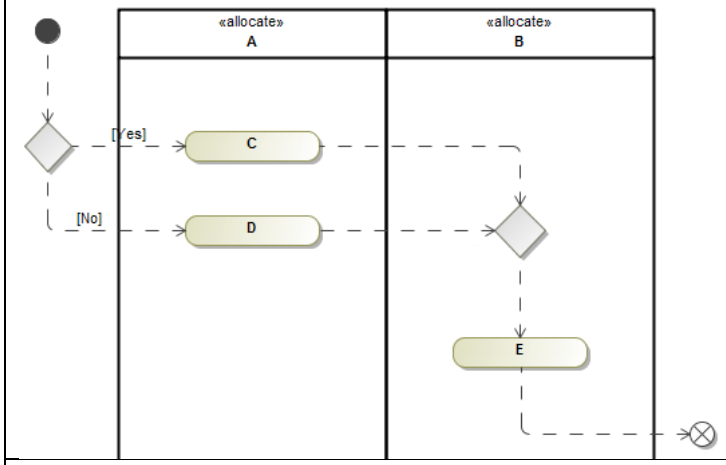
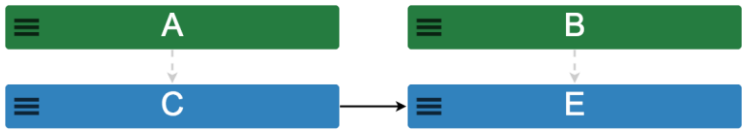
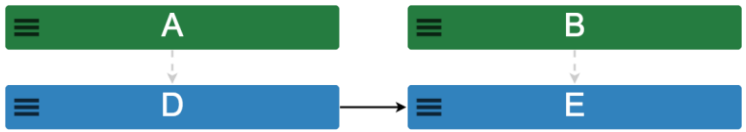
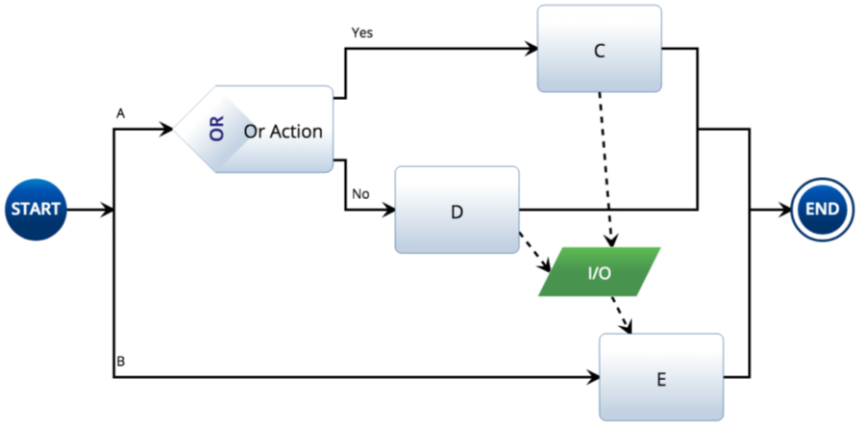
LML:
A and B are assets, each with its own parallel branch.
Actions C and D are allocated to A, actions E and F are allocated to B.
C precedes E and D precedes F. If a Yes decision is made, C will occur, if not, D.

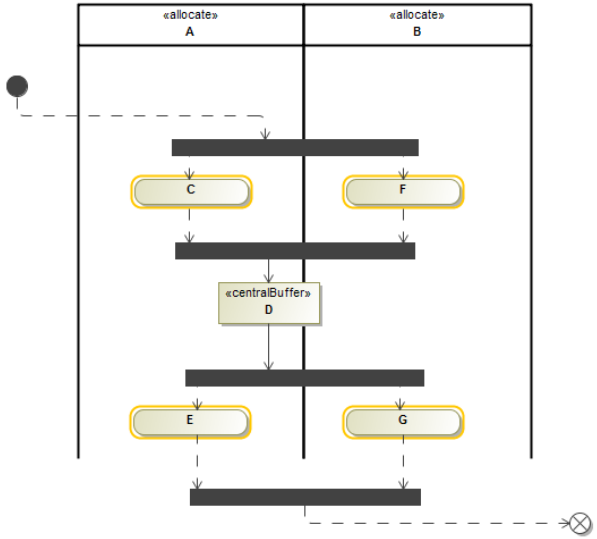
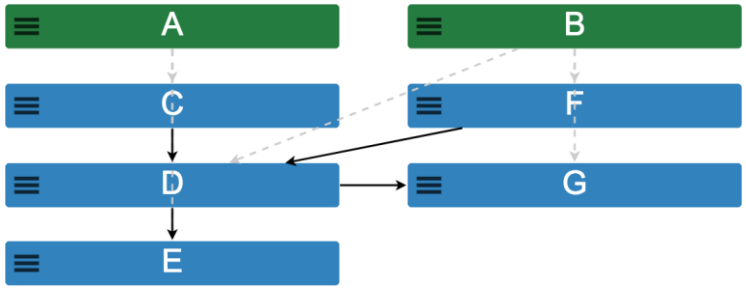
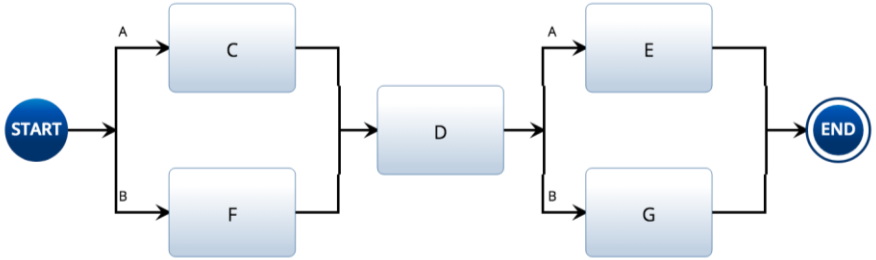



```
ROOT A: ( C | D );
ROOT B: ( E | F );


/* C precedes E and D precedes F through inter-root coordination.
*/

COORDINATE $x: C FROM A,
        $y: E FROM B
    DO ADD $x PRECEDES $y;
    OD;

COORDINATE $x: D FROM A,
        $y: F FROM B
    DO ADD $x PRECEDES $y;
    OD;
```



Activity Diagram	Corresponding MP Code and Graph(s)
SysML: A and B are components, each with its own swim lane. Activity C is allocated to A, and activities D and E are allocated to B. C precedes D if Yes, or E if No.  	/* A and B are root events. Event C or event D occurs in A, and event E or event F occurs in B. */ ROOT A: C; ROOT B: (D | E); /* C precedes E or D through inter-root coordination. */ COORDINATE $x: C FROM A, $y: (D | E) FROM B DO ADD $x PRECEDES $y; OD;  
LML: A and B are assets, each with its own parallel branch. Action C is allocated to A, and actions “Or Action,” D and E are allocated to B. C precedes the Or Action. D occurs if Yes, or E occurs if No.  	

Activity Diagram	Corresponding MP Code and Graph(s)
SysML: A and B are components, each with its own swim lane. Activities C and D are allocated to A, and activity E is allocated to B. C or D precedes E.  	/* A and B are root events. Event C or event D occurs in A, and event E occurs in B. */ ROOT A: (C | D); ROOT B: E; /* C or D precedes E through inter-root coordination. */ COORDINATE $x: (C | D) FROM A, $y: E FROM B DO ADD $x PRECEDES $y; OD;  
LML: A and B are assets, each with its own parallel branch. Actions “Or Action,” C and D are allocated to A, and action E is allocated to B. C or D precedes E.  	

Activity Diagram	Corresponding MP Code and Graph(s)
SysML: A and B are components, each with its own swim lane. Activities C and E are allocated to A, and activities F and G are allocated to B. An object flow with an ObjectNode was used to represent D.  	/* A and B are root events. Event C or event D occurs in A, and event E occurs in B. */ ROOT A: C D E; ROOT B: F D G; /* All instances of event D are shared by root events A and B. */ A, B SHARE ALL D; 
LML: A and B are assets, each with its own parallel branch. Actions C and E are allocated to A, and activities F and G are allocated to B. An action D is allocated to both A and B.  	

E.3. General Anti-Patterns to Avoid in Architecture Modeling

This section contains a version of work authored by Kristin Giammarco appearing as the conference 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. The heusitics have been recast as anti-patterns, extended with corresponding queries, and translated into the DoDAF / UPDM meta model language.

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 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 ontology for 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. Maier and Rechtin (2002) describes a data model as that which "specifies data that a system retains, and the relationships among the data."

Figure E.1 shows four small conceptual data model examples used for architecture modeling: the Department of Defense Architecture Framework Meta Model (DM2) (DODAF, 2010), the Unified Modeling Language (UML) Profile-Based Integrated Architecture (UPIA) (IBM, 2010), the System Description Language (SDL) (Long & Scott, 2011), and the Lifecycle Modeling Language (LML) (LML Steering Committee, 2015). 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.

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 analytics in a tool-agnostic manner. The anti-patterns 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. Automatic detection of the anti-patterns has already been implemented and tested in the Innoslate tool, demonstrating that automatic detection of these types of patterns is possible and useful. Testing with automation has turned up hundreds of model deficiencies across dozens of different academic and real project models. Further testing will fine-tune the automated detection with common exception cases, as well as metrics to assist with model maturity tracking over time.

As the Navy and other DOD agencies transition to MBSE, a list of anti-patterns like the one that follows can be provided as a precise yet tool-neutral specification for models developed for the customer organization. Finally, the high-level specification of anti-patterns enables the heavy lifting of enforcing such modeling lessons learned to be delegated to the modeling software tools, freeing the developers to focus on creative and pattern-finding tasks that are best suited for biological processors.

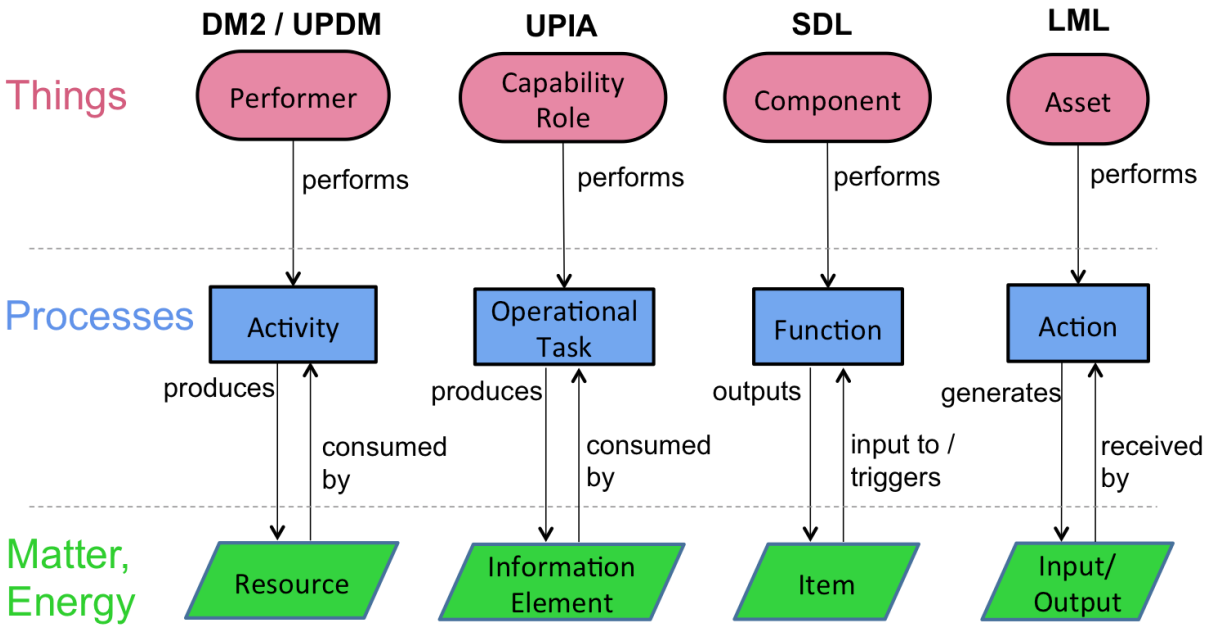


Figure E.2. Four simple CDM examples. Things doing processes exchanging matter and energy, in example taxonomies of DM2, UPIA, SDL, and LML. From research product [7].

The list of anti-patterns that follows is a catalog or menu of practices from which experienced architects can select general guidelines for assessing their models for well-formedness. They are expressed in natural language and in the DODAF /UPDM vernacular. The entity terms referred to in the anti-patterns (e.g., activity, performer, resource) are defined in (DODAF, 2010). The anti-patterns are poor practices to be generally avoided; they are experienced-based, and some have certain exception cases included with their description. Some anti-patterns are more or less restrictive than others, allowing the architect to use a combination that best fits the intent. The method used for developing the CDM-based heuristics upon which these anti-patterns are based is described in (Giammarco, 2014) and (Giammarco, 2012).

Each anti-pattern is followed by one or more corresponding queries that can be used to detect the presence of the anti-pattern in a model. 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. 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 anti-patterns, as well as provide experienced architects with a means to double-check their work.

E.3.1. Hierarchy anti-patterns

H.1. An activity, performer, or rule (requirement) having no children and no parent.

As the model matures, activities, performers and rules (including 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.

Queries to detect the number and list of:
H.1.1 Activities with no child and no parent
H.1.2 Performers with no child and no parent
H.1.3 Requirements with no child or parent

H.2. An activity, performer, resource, or rule (requirement) having more than one parent.

It is possible that an entity has more than one parent by mistake. However, an entity may belong to multiple hierarchies as a deliberate decision. Multiple parents may be desired in such cases as when entities are being reused in modular, composable architectures, or being mapped to multiple taxonomies. In some cases it can happen accidentally, which results in ambiguity about which is the correct path of parentage.

Queries to detect the number and list of:
H.2.1 Performers with more than one parent
H.2.2 Activities with more than one parent
H.2.3 Resources with more than one parent
H.2.4 Requirements with more than one parent

H.3. A performer or activity having exactly one child.

Performers, activities, 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 may be a redundant equivalency. This anti-pattern 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 approved by the lead architect.

Queries to detect the number and list of:
H.3.1 Performers with exactly one child
H.3.2 Activities with exactly one child

H.4. A performer or activity having more than [#userdefined] children.

For model comprehension, it is strongly recommended to limit the number of entities that appear at a given level. Buede (2009) recommends the optimum number to be between 3 and 6 activities; Miller (1956) uses a 7 plus or minus 2 heuristic. Regrouping a large number of entities at one level in a hierarchy often helps to break up the description into manageable chunks. For example, decompose a performer with 24 child performers into 4 new performers, each with 6 child performers. The 4 new performers are used to group the otherwise large number of entities.

Queries to detect the number and list of:
H.4.1 Performers with more than [userdefined#] children
H.4.2 Activities with more than [userdefined#] children

H.5. A performer, activity, resource or 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.

Queries to detect the number and list of:
H.5.1 Performers having itself as a child
H.5.2 Activities having itself as a child
H.5.3 Resources having itself as a child
H.5.4 Requirements having itself as a child

E.3.2. FUNCTIONAL/PHYSICAL ALLOCATION ANTI-PATTERNS

FPA.1. An activity that is not performed by any performer.

Activities that have not been allocated to a performer lack a physical embodiment. The physical object that will perform the activity should eventually be specified. If an activity cannot be mapped to a single performer, consider regrouping the activities / performers to support the assignment of activities to specific performers. This regrouping will enable a clear work breakdown structure that delineates "who or what" (performer) is responsible for "doing what" (activity).

Query to detect the number and list of:
FPA.1.1 Activities that are not performed by any performer

FPA.2. A performer that does not perform any activity.

Performers that have not been allocated any activity lack functionality. If a performer exists without an associated activity, it may be idle or unnecessary, or it may be that its activity has been overlooked. Each physical form should be allocated to a corresponding function.

Query to detect the number and list of:
FPA.2.1 Performers that do not perform any activity

E.3.3. FUNCTIONAL INTERACTION ANTI-PATTERNS

FI.1. An activity that produces some resource, but does not consume any resources.

To preserve equilibrium, an activity should not be able to produce a physical output without having consumed some physical input at some point in the past. A couple of recognized exception cases in modeling are 1) the use of "stub" activities, during executable modeling development or for simulation debugging, and 2) on a top-level context diagram where an unmodeled resource is assumed, and not explicitly shown because it is beyond the scope of the model.

Query to detect the number and list of:

FI.1.1 Activities that produce a resource without consuming any resource

FI.2. An activity that consumes a resource, but does not produce any resources.

As a corollary to the preceding anti-pattern, an activity should not be able to consume an input without producing some output. This is an alternate expression of the law of conservation of matter and energy. The same exception cases for modeling also apply to this anti-pattern.

Query to detect the number and list of:
FI.2.1 Activities that consume a resource without producing any resource

FI.3. An activity that does not produce or consume any resources.

An activity that does not produce or consume any resources is either idle, or may be intended to define a closed process. More realistically, all activities will interact with at least one other activity at some point. Even undesired interactions should be modeled, to aid the specification of counteractions.

Query to detect the number and list of:
FI.3.1 Activities that do not produce or consume any resources

FI.4. An activity that produces and consumes the same resource.

A "looping" resource is one that is produced by the same activity that consumes it. In an abstract model this may not be a problem, but at a simulation level this concept is not executable. The looping resource should be moved one level down into the decomposed view of the activity it is leaving and re-entering, to show distinct sub-activities that produce and consume it.

Query to detect the number and list of:
FI.4.1 Activities that produce and consume the same resource

FI.5. A resource that is not produced by some activity.

A resource that is not produced by any activity has an unspecified source. If there is no source for the resource, the implication is that it appears from nowhere or is otherwise never sent by any source activity.

Query to detect the number and list of:
FI.5.1 Resources not produced by any activity

FI.6. A resource that is not consumed by some activity.

A resource that is not consumed by any activity has an unspecified destination. If there is no destination for the resource, the implication is that it disappears to nowhere or is otherwise never received by any destination activity.

Query to detect the number and list of:
FI.6.1 Resources not consumed by any activity

FI.7. A resource that is neither produced nor consumed by some activity.

No resource should be isolated; each should have at least one relationship to some activity.

Query to detect the number and list of:
FI.7.1 Resources that are not produced by or consumed by any activity

E.3.4. PHYSICAL INTERACTION ANTI-PATTERNS

PI.1. A performer that does not connect to at least one connector.

To support interactions with other performers of any sort, a performer needs to connect to at least one connector. (In UPDM, a Connector is an abstraction of a Needline, which is a concept present in earlier versions of the DoDAF meta model.)

Query to detect the number and list of:
PI.1.1 Performers that are not connected to any connectors

PI.2. A connector that connects to fewer than two disjoint performers.

A connector is a point-to-point concept that applies a pairwise relationship between connected performers. If a connector is connected to less than two performers, its specification is incomplete.

Query to detect the number and list of:
PI.2.1 Connectors that connect fewer than two different performers

PI.3. A connector that connects to more than two disjoint performers.

A connector is a point-to-point concept that applies a pairwise relationship between connected performers. If a connector seems to need more than two connection points, consider modeling the connector as a performer instead.

Query to detect the number and list of:
PI.3.1 Connectors that connect more than two different performers

PI.4. A connector that does not transfer any resources.

A connector with no resources assigned to it may be unnecessary or incomplete.

Query to detect the number and list of:
PI.4.1 Connectors that do not transfer any resources

PI.5. A resource that is not transferred by any connectors.

A resource that has not been assigned to a connector may be overlooked in the requirements for that connector.

Query to detect the number and list of:
PI.5.1 Resources that are not transferred by any connectors

PI.6. Two performers that exchange some resource, but are not connected by any common connectors.

Performers that interact through their performed activities should have at least one connector in common.

Query to detect the number and list of:
PI.6.1 Performers that exchange some resource, but are not connected to any common connectors

PI.7. A resource exchanged between two performers that is not transferred by any common connector that connects those performers.

All resources exchanged between performers need to be assigned to a logical or physical connection between those performers.

Query to detect the number and list of:
PI.7.1 Resources exchanged between performers that are not transferred by a connector that connects the two performers

PI.8. A performer that does not produce or consume any resources to or from any other disjoint performer.

Performers that do not have any interactions are either idle, or intended to define a closed system. More realistically, all performers will interact with at least one other performer at some point. Even undesired interactions should be modeled, to help the specification of counteractions.

Query to detect the number and list of:
PI.8.1 Performers that do not interact with any other performers through exchange of resources

E.3.5. TRACEABILITY ANTI-PATTERNS

T.1. An entity that is not related to any 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.

Query to detect the number and list of:
T.1.1 Entities that are not related to any other entity

T.2. An activity that is not subject to any rule (requirement).

Any activities that are not subject to some rule (requirement) may be missing a tracing, or unnecessary. Exceptions may apply to modeled activities 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 activity at the top of an activity or function hierarchy, since such an entity is typically used for context only.

Query to detect the number and list of:
T.2.1 Activities that do not trace to any requirement

T.3. A performer that performs an activity that is not subject to a rule (requirement) that also constrains the performed activity.

Performers are related to requirements through the activities they perform. If a performed activity is not constrained by the same rules as its performer, necessary behaviors may be overlooked.

Query to detect the number and list of:
T.3.1 Performers that are not traceable to any requirement

T.4. A resource that is not subject to any rule (requirement).

Any resources that are not subject to some rule (requirement) may be missing a tracing, or unnecessary.

Query to detect the number and list of:
T.4.1 Resources that do not trace to any requirement

T.5. A connector that is not subject to any rule (requirement).

Any connectors that are not subject to some rule (requirement) may be missing a tracing, or unnecessary.

Query to detect the number and list of:
T.5.1 Connectors that do not trace to any requirement

T.6. A leaf-level rule (requirement) that does not constrain any entities in the Performer, Activity, Resource, or Connector class.

A rule (requirement) that does not constrain any of these modeled entities is either unnecessary or has not yet been traced in the architecture model.

Query to detect the number and list of:
T.6.1 Leaf-level requirements that are not satisfied by any activity, performer, resource or connector

E.3.6. STANDARDIZATION ANTI-PATTERNS

S.1. Each performer is constrained by some rule (standard).

A performer should be constrained by at least one rule (standard) to denote rules, policies, guidelines or protocols to which it is expected to adhere.

Query to detect the number and list of:
S.1.1 Performers that are not subject to any standard

S.2. Each activity is constrained by some rule (standard).

An activity should be constrained by at least one rule (standard) to denote rules, policies, guidelines or protocols to which it is expected to adhere.

Query to detect the number and list of:
S.2.1 Activities that are not subject to any standard

S.3. Each resource is constrained by some rule (standard).

A resource should be constrained by at least one rule (standard) to denote rules, policies, guidelines or protocols to which it is expected to adhere.

Query to detect the number and list of:
S.3.1 Resources that are not subject to any standard

S.4. Each connector is constrained by some rule (standard).

A connector should be constrained by at least one rule (standard) to denote rules, policies, guidelines or protocols to which it is expected to adhere.

Query to detect the number and list of:
S.4.1 Connectors that are not subject to any standard

S.5. Two performers that exchange some resource through performed activities, but are not constrained by a common rule (standard).

Performers that interact through their performed activities should have at least one rule (standard) in common, so that they are consistently constrained to support the interaction.

Query to detect the number and list of:
S.5.1 Performers that interact with each other through exchange of resources, but are not subject to a common standard

S.6. A resource is produced by some activity a_A of some performer p_A and consumed by some activity a_B of another performer p_B, but a_A and a_B are not subject to any common rule (standard) that constrains both p_A and p_B.

Activities that interact through exchange of resource(s) should have at least one rule (standard) in common with the performers that perform them, so that they are consistently constrained to support the interaction.

Query to detect the number and list of:
S.6.1 Activities that exchange some resource that are not subject to any common standard with the performers performing those activities

S.7. An exchanged resource between two performers that is not subject to some rule (standard) that constrains both performers.

A resource exchanged between interacting performers should have at least one rule (standard) in common with its source and destination performers, so that the resource and the performers are consistently constrained to support the interaction.

Query to detect the number and list of:
S.7.1 Resources exchanged between performers that are not subject to any common standard with those performers

S.8. A connector that connects two performers and transfers a resource between those performers that is not subject to some rule (standard) that constrains both performers.

A connector that connects interacting performers should have at least one rule (standard) in common with the performers that they connect, so that the connector and connected performers are consistently constrained to support the interaction.

Query to detect the number and list of:

S.8.1 Connectors that connect performers transferring some resource that are not subject to any common standard with those performers

E.3.7. Summary of Queries for Anti-patterns in Four Conceptual Modeling Langauges

This section summarizes the anti-patterns presented in the previous sections and provides translations into LML, UPIA, and SDL to demonstrate their conceptual universality. The translations represent the best matches available. Translations may exist for additional CDMs beyond the scope of this summary.

Table E.2. Anti-Patterns in Four Languages

No.	DM2 / UPDM	UPIA	SDL	LML
H.1.1	Activities with no child and no parent	Operational tasks with no child and no parent	Functions with no child and no parent	Actions with no child and no parent
H.1.2	Performers with no child and no parent	Capability roles with no child and no parent	Components with no child and no parent	Assets with no child and no parent
H.1.3	Requirements with no child or parent	Requirements with no child or parent	Requirements with no child or parent	Requirements with no child or parent
H.2.1	Performers with more than one parent	Capability roles with more than one parent	Components with more than one parent	Assets with more than one parent
H.2.2	Activities with more than one parent	Operational tasks with more than one parent	Functions with more than one paren	Actions with more than one parent
H.2.3	Resources with more than one parent	Information elements with more than one parent	Items with more than one parent	Input/outputs with more than one parent
H.2.4	Requirements with more than one parent	Requirements with more than one parent	Requirements with more than one parent	Requirements with more than one parent
H.3.1	Performers with exactly one child	Capability roles with exactly one child	Components with exactly one child	Assets with exactly one child
H.3.2	Activities with exactly one child	Operational tasks with exactly one child	Functions with exactly one child	Actions with exactly one child
H.4.1	Performers with more than [userdefined#] children	Capability roles with more than [userdefined#] children	Components with more than [userdefined#] children	Assets with more than [userdefined#] children
H.4.2	Activities with more than [userdefined#] children	Operational tasks with more than [userdefined#] children	Functions with more than [userdefined#] children	Actions with more than [userdefined#] children
H.5.1	Performers having itself as a child	Capability roles having itself as a child	Components having itself as a child	Assets having itself as a child
H.5.2	Activities having itself as a child	Operational tasks having itself as a child	Functions having itself as a child	Actions having itself as a child
H.5.3	Resources having itself as a child	Information elements having itself as a child	Items having itself as a child	Input/outputs having itself as a child
H.5.4	Requirements having itself as a child	Requirements having itself as a child	Requirements having itself as a child	Requirements having itself as a child
FPA.1.1	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	Actions that are not performed by any asset
FPA.2.1	Performers that do not perform any activity	Capability roles that do not perform any operational task	Components that do not perform any function	Assets that do not perform any action
FI.1.1	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	Actions that generate an input/output without receiving any input/output

No.	DM2 / UPDM	UPIA	SDL	LML
FI.2.1	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	Actions that receive an input/output without generating any input/output
FI.3.1	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	Actions that do not generate or receive any input/outputs
FI.4.1	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	Actions that generate and receive the same input/output
FI.5.1	Resources not produced by any activity	Information elements not produced by any operational task	Items not produced by any function	Input/outputs not generated by any action
FI.6.1	Resources not consumed by any activity	Information elements not consumed by any operational task	Items not consumed by any function	Input/outputs not received by any action
FI.7.1	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	Input/outputs that are not generated by or received by any action
PI.1.1	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	Assets that are not connected by any conduits
PI.2.1	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	Conduits that connect fewer than two different assets
PI.3.1	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	Conduits that connect more than two different assets
PI.4.1	Connectors that do not transfer any resources	Needlines that do not transfer any information elements	Links that do not transfer any items	Conduits that do not transfer any input/outputs
PI.5.1	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	Input/outputs that are not transferred by any conduits
PI.6.1	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	Assets that exchange some Input/output, but are not connected by any common conduits

No.	DM2 / UPDM	UPIA	SDL	LML
PI.7.1	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	Input/outputs exchanged between assets that are not transferred by a conduit that connects the two assets
PI.8.1	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	Assets that do not interact with any other assets through exchange of input/outputs
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	Activities that do not trace to any requirement	Operational Tasks that do not trace to any requirement	Functions that are not based on any requirement	Actions that do not satisfy/verify/trace to any requirement
T.3.1	Performers that are not traced from any requirement	Capability Roles that are not traced from any requirement	Components that are not specified by any requirement	Assets that are not traced from any requirement
T.4.1	Resources that do not trace to any requirement	Information Elements that do not trace to any requirement	Items that are not specified by any requirement	Input/outputs that do not satisfy any requirement
T.5.1	Connectors that do not trace to any requirement	Needlines that do not trace to any requirement	Links that are not specified by any requirement	Conduits that do not satisfy any requirement
T.6.1	Leaf-level requirements that are not traced to any activity, performer, resource, or connector	Leaf-level requirements that are not traced to any operational task, capability role, information element, or needline	Leaf-level requirements that are not the basis of or do not specify any function, component, item, or link	Leaf-level requirements that are not satisfied by any action, asset, input/output, or conduit
S.1.1	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	Assets that do not reference any standard-labeled artifact
S.2.1	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	Actions that do not reference any standard-labeled artifact
S.3.1	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	Input/outputs that do not reference any standard-labeled artifact

No.	DM2 / UPDM	UPIA	SDL	LML
S.4.1	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	Conduits that do not reference any standard-labeled artifact
S.5.1	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	Assets that interact with each other through exchange of input/outputs, but satisfy no common standardizing requirement
S.6.1	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	Actions that exchange some input/output that reference no common standard-labeled artifact with the assets performing those actions
S.7.1	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	Input/outputs exchanged between assets that reference no common standard-labeled artifact with those assets
S.8.1	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	Conduits that connect assets transferring some input/output that reference no common standard-labeled artifact with those assets

APPENDIX F: INSTRUCTIONS FOR DOWNLOADING MP MODELS

The following MP models were developed to support this research effort. They demonstrate some of the MP features available in MP-Firebird and are referenced in Appendix G. Behaviors specified in the MP models include both regular (or "normal" scenarios), and "irregular" (or abnormal, like bingo fuel) scenarios. All models when run for scopes 1 or 2 on the Firebird provide an exhaustive set of use cases, or scenarios, for browsing and exploration. The .mp models are text files that may be downloaded from the project website at https://sercuarc.org/project/?id=35&project=Verification+and+Validation+%28V%26V%29+of+System+Behavior+Specifications. They can be run by following the steps below:

1. Go to https://firebird.nps.edu
2. Use the Import menu to "Load .mp or .wng file" by clicking on the Choose File button
3. Select the downloaded .mp model you are interested in browsing
4. Click Open and the model will load into the text window on the left. Use the Run button and Scope slider at the top left to browse traces from the model on the right.

F.1. UAV IN HUMANITARIAN ASSISTANCE AND DISASTER RELIEF (HADR) MISSION SCENARIO

These behavior models follow the UAS HADR mission narratives in Appendix G section 1.

- Phase_1_UAV_Ingress.mp - Demonstrates basic MP event grammar rules: sequence, alternative, simple coordination with ADD commands.
- Phase_2_UAV_Onstation.mp - Demonstrates iteration, coordination of events within iteration, simple conditional operation and MARK command.
- Phase_3_UAV_Egress.mp - Event set pattern (concurrency), optional event patterns, conditional coordination, SHARE ALL coordination, simple SAY clause.
- Phase_4_UAV_Postflight.mp - Multiple coordination with ADD command.

F.2. UAV IN SEARCH AND RESCUE MISSION SCENARIO (SKYZER MODEL)

These behavior models follow the Skyzer mission narrative and SysML behavior model in Appendix G section 5.

- Non_Combatant_Operations_Scenario_1_Phase1.mp – Demonstrates model segmentation of phase 1 (Prepare / Configure) of the Skyzer SysML behavior model.
- Non_Combatant_Operations_Scenario_1_Phase2.mp – Demonstrates model segmentation of phase 2 (Take Off) of the Skyzer SysML behavior model.
- Non_Combatant_Operations_Scenario_1_phase3.mp – Demonstrates model segmentation of phase 3 (Transit / Navigate) of the Skyzer SysML behavior model.
- Non_Combatant_Operations_Scenario_1_Phase4.mp – Demonstrates model segmentation of phase 4 (Post Mission Task) of the Skyzer SysML behavior model.
- Non_Combatant_Operations_Scenario_1_Phase3_expanded.mp – Demonstrates model expansion of phase 3 (Transit / Navigate) of the Skyzer SysML behavior model.
- Av7f_phase3.mp – Demonstrates modeling errors in the phase 3 (Transit / Navigate) model resulting in suppression of desired emergent behaviors.
- AV_Temp.mp – Demonstrates model debugging of phase 3 (Transit / Navigate), Air Vehicle only.

This appendix provides a preliminary demonstration of the methodology introduced in Appendix D. Since the methodology is a doctoral work in progress, this demonstration utilizes the first three steps, and then provides an example and discussion of how unexpected emergent behavior may present in Monterey Phoenix (MP) models. For the purposes of this model-based V&V demonstration, a US Navy Humanitarian Assistance and Disaster Relief (HADR) operational context is used to model the employment of an Unmanned Aircraft System (UAS). The HADR mission, authored by NPS students with operational and aviation experience, is first presented in the form of a Design Reference Mission (DRM) to define the operational context and behavior narratives, aligning with Step 1 in Appendix D. Next, we show how to dissect the written Ingress Phase mission narrative into distinct actors and activities for composing MP grammar rules, consistent with Step 2. We then show how to define coordination for the events in each actor's grammar rule, as described in Step 3. An example of an early version of the Ingress model is next presented to demonstrate the discovery of errors including an analysis of unexpected emergent behavior. The appendix concludes with a summary of MP modeling heuristics, and common modeling errors to avoid.

G.1. COMPOSING A DRM TO FRAME A PROBLEM

The Design Reference Mission (DRM) defines the projected threat and operating environment baseline for a rigorous systems engineering process to help ensure that future Navy systems can meet 21st century challenges and uncertainties (Skolnick and Wilkins, 2000).

Design Reference Missions (DRMs) provide an initial operational context description which provides context and acts as a guide when instantiating systems. System operators, stakeholders and technological subject matter experts (SMEs) provide input which shapes the primary capability need, a description of the proposed system and its boundaries (operational concept), and aids in understanding the projected operating environment in which mission functions take place. As such, a DRM provides a meaningful system concept which provides a basis for further Systems Engineering (SE) work. Taking the time to frame a problem well is the first and most critical step in successful system architecture and design (Giammarco & Shebalin, 2016).

The DRM provides a common framework to link the SE efforts to stakeholder needs and provides an analytic basis for further comparison between instantiated systems meeting the stakeholder needs (Skolnick & Wilkins, 2000). Figure G.1 below shows the DRMs location and importance in the SE process (Skolnick & Wilkins, 2000).

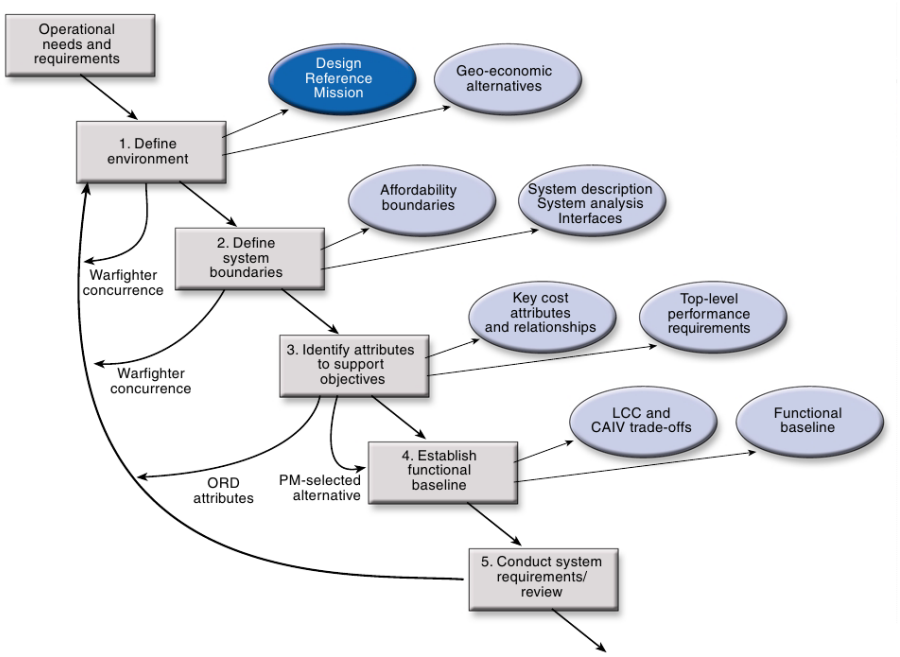


Figure G.1 - DRM in the SE Process (Skolnick & Wilkins, 2000)

A DRM also supports project planning and synchronization by establishing a common reference for definitions, description of the demonstration environment and objectives, and expectations on measuring success. Preliminary models of the pursued system design is outlined, however it only intends to establish a draft blueprint that will be refined during project execution.

Architecture design and analysis is an iterative, nonlinear process that unfolds based on information as it becomes available. This project is exploratory project by nature; this DRM is expected to mature as more stakeholder inputs are received and integrated, and is not intended to represent an allencompassing specification of requirements etched in stone. The objective here is to acknowledge that the use of UASs in HADR is an evolving concept; the DRM intends only to put forward an initial understanding of the problem space for solution providers and stakeholders. Solution providers can refer to the DRM to understand expectations and manage requirements. Additionally stakeholders may utilize the DRM to aid decision making concerns regarding candidate operational and solution architectures.

The remaining subsections of G.1 contain a version of work authored by LCDR Christopher Krukowski and CDR Kathleen Giles entitled “Design Reference Mission for Unmanned Aircraft System Conducting Humanitarian Assistance Disaster Relief Mission,” a student project from the SI4022 System Architecture course taught at the Naval Postgraduate School , Monterey, CA, June 2017.

G.1.1. INTRODUCTION

Krukowski and Giles sought to outline a DRM for a UAS in the participation of an HADR operation. The DRM seeks to validate potential solutions and provide a basis for the development of a concept of operations (CONOPS) and system design for UAS support during a HADR mission (2017). The language

was kept as solution neutral as possible to allow for varied potential concept generation. The main questions this DRM is looking to answer are:

- Can a UAS support a HADR mission?
- What role will a UAS play in supporting a HADR mission?
- How effective will the UAS be in its support of a HADR mission?

The scenario in this DRM will focus on the support provided by the UAS during the aftermath of an earthquake in a foreign country. From this basic scenario, communication and coordination tactics can be explored to influence the UAS architecture. Additional scenario variations will provide basis for follow-on analysis from the use cases generated.

G.1.1.1. Mission Background

HADR is one of the seven core mission of the United States Navy as outlined in the 2015 National Maritime Strategy. Falling within the essential functional area of power projection, the HADR mission allows the Navy to project what is described as "smart power" ashore (USN, 2015). Humanitarian assistance mission are conducted in response to foreign disasters either natural such as earthquakes, droughts, or floods, or man-made such as riots, civil strife, or epidemics (JCS, 2014). Recent examples of the Navy employing this "smart power" projection include post-earthquake in Haiti in 2010, post-tsunami in Japan in 2011, and post-typhoon in the Philippines in 2013 (USN, 2015). With forces forward deployed throughout the world, the Navy is able to respond very rapidly and provide relief without the benefit of shore-based facilities. Navy assets provide a Joint Force Commander with a unique set of capabilities including medical facilities, search and rescue, and reconnaissance capability (USN, 2015). During the 2010 Haiti earthquake relief, remote sensing data, such as the aerial imagery collected and disseminated by a UAS, were effectively used to assess the degree of landslides, the extent of blocked roadways, infrastructure damage assessment, and guiding search and rescue (SAR) teams (Eberhard, 2010). By comparing data gathered during remote sensing missions to previous data, mission coordinators can develop rescue plans to focus their efforts on the hardest hit areas after a disaster. Natural disasters can also leave an affected area's communication infrastructures in pieces after an incident, and there is often a need to establish impromptu communication networks to enable effective communication between the multiple entities participating in the disaster relief operation. U.S. Navy assets are typically employed in HADR mission within close proximity to the shoreline, however, with approximately 40% of the world's population living within 100 km of the ocean; the Navy expects its role in HADR missions to remain substantial (United Nations, 2017).

G.1.1.2. Operational Concept

Although the U.S. Military might be the first on scene due to the forward deployed aspect of military forces around the world, the United States Agency for International Development (USAID) or Department of State (DOS) normally lead HADR activities (JCS, 2014) (NWDC, 2006). It is the role of the US ambassador to the country in need of support to manage the relationship with the host nation (HN) (NWDC, 2006). Military support is requested by the Department of State and then orders are passed along to the regional combatant commander. The regional combatant commander will then organize and coordinate the appropriate joint task force (NWDC, 2006). For situations involving an multinational response, the lead agency is typically the United Nations.

There are many different operational models for HADR missions. Civilian agencies typically simplify HADR into three phases: *preparation, immediate response* and *reconstruction* (Kovacs & Spens 2007)

while the military uses the joint operational planning processes such as those found in JP 3-29 Foreign Humanitarian Assistance, JP 5-0 Joint Operation Planning and TACMEMO 3-07.6-06 Foreign Humanitarian / Disaster Relief Operations Planning.

The military planning process typically follows the joint operational planning process (JOPP). This process normally will follow seven specific steps:

1. Initiation
2. Mission Analysis
3. Course of Action (COA) Development
4. COA Analysis and Wargaming
5. COA Comparison
6. COA Approval
7. Plan or Order Development (JCS, 2011)

HADR is unique in that there is limited time between notification and execution, and the location where a HADR mission will occur is not known until the disaster strikes. This does not make the planning process any less important, though. Military planners develop generic HADR plans that can be put in place in the event of a disaster, but until the disaster occurs, specific environment and circumstances are not known. During the *planning* phase, the HN, USAID, other US government agencies, non-governmental organizations (NGOs), and inter-government agencies (IGOs) conduct needs assessments to determine the capability of each participating agency. The relief system and mission statement are developed based on the required support needed, within the context of the operational environment. Once the relief system has been established, the joint force structure is developed to provide coordination and communication processes between agencies (JCS, 2014).

The *execution and assessment* phase begins with deployment, which includes joint reception, staging, onward movement and integration (JCS, 2014). Once the forces are in place, command and control (C2) and sustainment operations bring communication, transportation, and logistics support to the HN. *Intelligence and information gathering* and dissemination help the appropriate relief agency prioritize requests for aid and deliver supplies. Finally, *assessment* is conducted throughout the operation and is a key component for a smooth transition of control back to the HN and NGOs at the appropriate time. The expected duration of a HADR mission can vary widely. The relief efforts for the Indian Ocean tsunami, Hurricane Katrina and the Haitian earthquake lasted 81, 42 and 72 days respectively; however, the USN part of the mission was completed in 41, 38, and 41 days respectively (Greenfield, 2011). In those three examples, NGOs and the HN were able to take over the relief effort after the immediate response and stabilization efforts made by USN assets in the initial five to six weeks.

This DRM will focus on the immediate response phase, and the following assets, representative of assets that would be found in an amphibious ready group (ARG) may be used in the scenario present in this DRM:

- US Navy ships:
    - LHD (landing helicopter dock) amphibious assault ship – with medical support, CH-53 and MH-60 variants for transport, lift, and SAR, and LCAC for ship-to-shore supply delivery

- LHA (landing helicopter assault) amphibious assault ship - with medical support, CH-53, MH-60 variants, and MV-22 for transport, lift.
- LPD (landing ship dock) amphibious assault ship – with medical support, launch and land capability for CH-53, and MH-60 variants supporting transport, lift, and SAR, and LCAC for ship-to-shore supply delivery

- **JTF C2** – Joint Task Force Command and Control node: tactical air control squadron (TACRON), joint force air component commander (JFACC), or other JTF asset who will be providing air traffic control. Responsible for coordination between military and NGO assets.
- **Helicopters** – MH-60 variants and CH-53, for SAR and transport of ship-to-shore personnel and supplies.
- **UAS** – consists of a Group 3 UAV launched from the deck of an LHD, LHA, LPD, or shore facility, a ground control station (GCS), launch and recovery systems. Group 3 UAVs have a maximum gross take-off weight of less than 1,320 lbs. and operate at medium altitudes with medium to long range and endurance (Navy 2008). Potential Payloads may include:
  - Streaming infrared (IR), video for detecting, classifying and identifying targets in the IR spectrum during wide-area, day or night search
  - Streaming electro-optic (EO) video for detecting, classifying, and identifying targets in the visible light spectrum during wide-area, day-time search in clear atmosphere
  - Synthetic aperture radar for all-weather detection and classification of stationary objects, and for determining the status of infrastructure such as roads, bridges, and buildings. IR and EO sensors can be cross-cued to an initial synthetic aperture radar target detection.
  - Simultaneous voice relay and data-link communication over VHF, UHF, and military and commercial satellite

While Military Sealift Command (MSC) cargo and hospital ships (T-AH) are useful in HADR missions for carrying large quantities of cargo and functioning as floating hospitals, they are not included as assets for this immediate response phase scenario. MSC cargo ships are manned with small crews, and may not have embarked helicopters, limiting their SAR and other immediate response mission utility. Hospital ships are not kept in a ready status (medical personnel are pulled from Navy active duty hospital staff or from the Navy's Reserve), which delays their arrival.

G.1.2. Projected Operating Environment

The Projected Operating Environment (POE) is the environment in which the UAS is expected to operate. This section provides details that describe the environmental conditions, types of locations, and threats to which the system will be subject. The POE establishes a context within which interactions and interfaces of the system may be modeled to produce measurable outcomes to enable physical architecture design tradeoffs (Whitcomb et al., 2015).

G.1.2.1. Environmental Conditions

The HADR mission domain is the nation of Haiti, specifically the capital city of Port-au-Prince and the surrounding area. The expected operating area is depicted in Figure G.2.

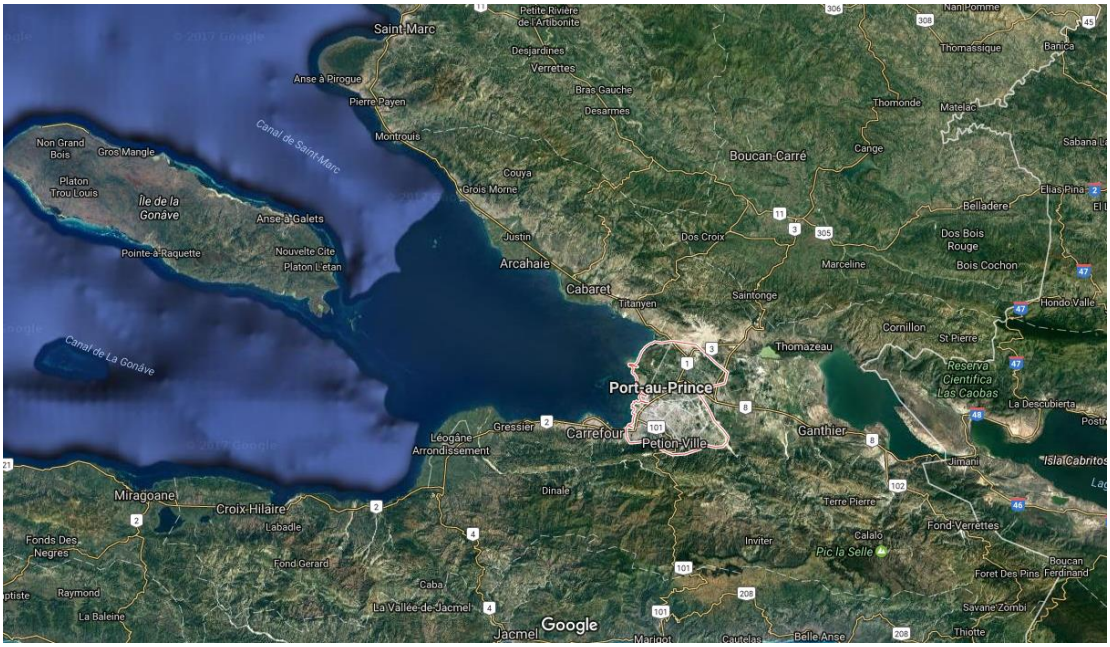


Figure G.2 - Map of Port-au-Prince and surrounding area. Adapted from Google Maps (2017)

The climate of Haiti is classified as a Tropical Wet climate on the Köppen Climate Classification Scale (Climate-data.org, 2017) (Wikipedia, 2017). This means the UAS should expect to operate under the following environmental climate conditions (Climate-data.org. 2017):

- Temperature:
    - 65 – 95 degrees Fahrenheit
    - Average: 80 degrees Fahrenheit
- Precipitation:
    - 1.6 – 9.6 inches of rain per month
    - Wind gusts less than 20 kts

In addition to the climatic environment, the operating environment for the UAS is defined as:

- Daytime, visual meteorological conditions
- Maximum operating altitude: <18K' MSL
- Mountainous terrain - Highest peak: Pic la Selle (8,793 ft)

G.1.2.2. Threat Details

G.1.2.2.1. Assumed Threat Environment

The UAS is expected to operate in a permissive environment. The only threats to the success of the HADR UAS mission are expected to come from convective weather, mountainous terrain, and unintentional electromagnetic interference. Threats from human actors, groups, or governments are not considered in this DRM.

G.1.2.2.2. Assumed Threat Mitigation

A mobile weather station on board the host platform (LHD, LHA, or LPD) will be used to monitor weather conditions and flights will not be conducted in or around convective weather. If a mission is airborne and convective weather develops, consideration should be given to returning the UAV to base.

The terrain to the north and south of Port-au-Prince is mountainous, with the highest peak being Pic la Selle at 8,793 feet. The threat from the mountainous terrain will be mitigated through proper mission planning on the part of the GCS Operator.

The potential UAS is expected to operate in the vicinity of both a rural and urban environment. To mitigate the potential problem of unintentional electromagnetic interference, the GCS Operator will need to ensure proper mission planning is conducted to coordinate frequency separation in an effort to minimize interference with other assets assisting in the HADR mission.

G.1.3. Mission and Measures

G.1.3.1. Mission success Requirements

In order for the mission to be successful, the UAS under consideration must meet the following high-level requirements:

- Embark on and operate from U. S. Navy ships (LPD-19, LHD–5, LHA–6, or LHA-8 class), or embark on a U.S. Navy ship and disembark from a U.S. Navy ship at operational location, and operate from an unprepared shore location
- Collect and disseminate imagery data to military and civilian units to improve timeliness of humanitarian need prioritization, and decrease response time to deliver relief supplies or conduct search and rescue operations
- Provide communication relay to other military and civilian units in order to improve information dissemination among participating units, and decrease response time to deliver relief supplies

G.1.3.2. Mission Definition

The main reference mission provides a level of detail necessary for collecting measures to assess mission success requirements. As technology and tactics develop, additional mission variations will be added to allow for increased aircraft, incorporation of sensors and weapons, different tactics, other unmanned or manned assets, and different environmental conditions. The capability needs statement for the main UAS HADR mission is:

The US Navy needs a cost-effective means to rapidly conduct reconnaissance, and support network centric communication to support immediate response to HADR missions, freeing crews to conduct other necessary missions.

The main mission will have an operational situation (OPSIT) that describes the assumptions being made about the mission's operational environment, which have implications for logistics, deployment, and time required to complete the mission. The simulation results for the OPSIT influence the design, and may provide test cases for future Developmental or Operational Test & Evaluation (DT&E or OT&E) (Skolnick & Willkins, 2000). Two similar OPSITs are presented in this DRM; ship based and shore based.

The following OPSITs pertains to a US Navy HADR task force supporting the Government of Haiti (GOH). Both OPSITs are fictional scenarios, based on the cataclysmic magnitude 7.0 earthquake which occurred on 12 January 2010. In these scenarios, Haiti will sustain a magnitude 6.8 earthquake on 15 January 2018, at 1603 EST. Over the following 5 days, more than 10 aftershocks greater than magnitude 3.5 will be recorded. In order to increase the effectiveness of this mission, the GOH and USAID have requested the assistance of the US Navy to provide relief in the form of medical support, temporary communication infrastructure, airborne reconnaissance, SAR, relief supply delivery, and berthing capacity for an expected 40-day period.

G.1.3.2.1. Ship-Based OPSIT

The primary mission for the UAS is to provide remote sensing data in order to assist infrastructure damage assessment and to guide SAR operations. The UAS will deploy as part of an amphibious ready group (ARG) onboard an LPD class ship. Once the ARG is on scene, the UAS launches from the LPD when tasked, and establishes communication with the JTF C2 node. The UAS proceeds with the briefed tasking and target deck, but may receive in-flight re-tasking based on the dynamics of the relief effort. The remote sensing data, imagery, and target positions will be provided to other assets assisting the HADR mission via common data link and Link-16. A secondary mission is for the UAS to act as an interim airborne communications relay node over the area of operation until more permanent communications can be established. The primary requirement will be relaying UHF and VHF voice and data communications between geographically separated ground elements that cannot establish direct line-of-sight communications. Once the UAS reaches bingo fuel or mission conclusion is commanded, it egresses to the host platform, where it is recovered.

G.1.3.2.2. Shore-Based OPSIT

Similar to the ship-based OPSIT, the primary mission of the UAS will be to deploy with an ARG to provide remote sensing data in order to assist infrastructure damage assessment and to guide SAR operations. Upon arrival on scene, the UAS will disembark from the host platform and set up operation ashore at an unprepared location. From a shore-based location, the UAS will complete all the same tasking as detailed in the ship-based OPSIT. At the conclusion of the ARG's HADR mission support, the UAS will re-embark on the host platform to return to base.

G.1.3.3. Mission Execution

The following mission narrative describes a general UAS HADR aerial reconnaissance reference mission. This mission narrative can be applied to both the ship-based and shore-based OPSITs detailed above. Mission start time (T+00) occurs at the beginning of the staging phase. The mission will be composed of

the following phases: staging, mission planning, preflight, ingress, on-station, egress, and postflight. The staging and mission-planning phase need not occur immediately preceding the remaining phases and may have a significant time separation from the preflight, ingress, on-station, egress, and postflight stage. For subsequent HADR missions, the staging phase may not be required if conducting the mission from the same location. The system will enact an operational failsafe mode if a system error (such as loss of command and control information or loss of GPS signal) occurs. Two parties will operate the UAS:

1. The GCS Operator, who is in charge of operating the GCS, controlling the UAV while airborne, and coordinating with external units
2. The Ground Crew, who is responsible for UAV pre-flight, launch, recovery, and postflight.

G.1.3.3.1. UAS HADR Mission Narrative for Aerial Reconnaissance Mission

G.1.3.3.1.1. Staging Phase:

Pre-Conditions: Staging phase begins once the UAS in its travel configuration arrives at the designated deployment site (ground or shipboard)

- Ground Crew unpack and assemble UAS from its travel configuration
- GCS Operator and Ground Crew configure system in preparation for executing a flight mission upon receipt of orders
- GCS Operator and Ground Crew test UAS to ensure an operational status
- If UAS is not in an operational status, Ground Crew conducts maintenance to get them to an operational status
- Ground Crew reports all components in an operational status to the GCS Operator
- GCS Operator ensure GCS has up-to-date digital charts
- If the GCS does not have up-to-date charts, the GCS Operator uploads up-to-date charts to the GCS
- GCS Operator establishes communications with the JTF C2

Post-Conditions: Staging phase ends when the UAS has been assembled, components tested, communications established, and the crew is ready to receive and execute missions

G.1.3.3.1.2. Mission Planning Phase:

Pre-Conditions: The Mission Planning Phase begins when the GCS Operator receives a tasking order to be prepared to execute an aerial reconnaissance mission

- GCS Operator reviews air tasking order (ATO), special instructions (SPINS), and any other relevant operational tasking orders (OPTASKS) from the JTF C2. These documents contain aircraft callsigns, mission types, coordination frequencies, airspace boundaries, ingress/egress corridors, failsafe rally waypoints and other relevant mission information.
- GCS Operator checks weather
- GCS Operator powers on GCS
- If there are any changes to the communication frequencies, the GCS Operator updates any communication frequencies
- If any there are any changes from previous digital chart configurations, the GCS Operator updates the digital charts on the GCS
- GCS Operator plans ingress and egress routes to and from the reconnaissance area, on-station waypoint, recovery waypoint, and failsafe rally waypoint
- GCS Operator plans search pattern to cover assigned tasking order
- GCS Operator downloads mission plan from GCS, provides mission plan to Ground Crew, and Ground Crew uploads mission plan to UAV
- If UAV launch is not imminent, GCS Operator powers off the GCS
- GCS Operator reviews data collection plan
- GCS Operator and Ground Crew conduct mission brief

Post-Conditions: Mission Planning Phase ends following completion of the mission brief

G.1.3.3.1.3. Preflight Phase:

Pre-Conditions: Preflight phase begins when UAS is powered on for launch

- Ground Crew powers on the launcher
- Ground Crew positions the launcher into the wind
- Ground Crew performs the dry launch checkout
- Ground Crew installs data storage devices in UAV
- GCS Operator installs data storage devices in GCS
- Ground Crew loads UAV on launcher

- If the GCS is not already powered on, the GCS Operator powers on the GCS
- GCS Operator verifies correct mission plan is loaded in the GCS
- If incorrect plan is loaded, GCS Operator loads correct plan into GCS
- Ground Crew powers on the UAV
- Ground Crew conducts engine run up
- Ground Crew verifies the correct mission plan is loaded in the UAV
- If incorrect mission plan is loaded into UAV, GCS Operator downloads correct plan from GCS, provides the plan to the Ground Crew, and Ground Crew uploads correct plan to UAV
- GCS Operator verifies connectivity with the UAV
- GCS Operator ensures GCS and UAV have synced mission plans
- If shipboard, GCS Operator establishes communications with host ship
- GCS Operator establishes communications with JTF C2
- UAV reports all systems flight ready to the GCS Operator
- GCS Operator verifies UAV system is in a flight ready status
- If UAV is not in a flight ready status, Ground Crew performs maintenance on the UAV and reports when maintenance is complete to the GCS Operator

Post-Conditions: Preflight phase ends when the UAS is in flight ready status

G.1.3.3.1.4. Ingress Phase:

Pre-Conditions: Ingress Phase begins when the GCS Operator receives a launch clearance from the JTF C2

- GCS Operator receives launch command from JTF C2
- If shipboard, the GCS Operator requests host ship maneuver to achieve launch parameters
- GCS Operator checks launch parameters for safety
- GCS Operator receives launch clearance from host platform
- GCS Operator commands Ground Crew to launch UAV

- Ground Crew commands UAV to launch
- UAV launches
- UAV maneuvers to clear obstacles then maneuvers to reach ingress altitude
- UAV levels off at ingress altitude
- UAV transmits status (payload and systems) and position messages to GCS Operator (on-going throughout each phase of mission)
- If status and position are acceptable, GCS Operator commands UAV to proceed on ingress flight path, otherwise the GCS Operator commands UAV to return to base. Proceed to egress phase.
- UAV follows ingress flight path to reach on-station area and altitude
- GCS Operator monitors UAV status and position during flight path to on-station area and altitude
- The UAV reaches initial on-station waypoint and reports position to GCS Operator

Post-Conditions: The Ingress Phase ends when the UAV reaches the initial on-station waypoint

G.1.3.3.1.5. On-station Phase:

Pre-Conditions: The On-station phase begins when the UAV reaches the assigned on-station area

- UAV reports arrival at on-station waypoint to the GCS Operator
- GCS Operator verifies correct position
- GCS Operator checks payload status
- If at the correct position and payload status is still good, the GCS Operator commands UAV to commence on-station tasking. If not the GCS Operator commands the UAV to return to base. Proceed to Egress Phase.
- GCS Operator establishes communication with Other Assets
- UAV conducts on-station tasking

G.1.3.3.1.5.1. Conduct Aerial Reconnaissance

- GCS Operator reports commencement of aerial reconnaissance to JTF C2 and Other Assets

- UAV collects data on assigned search area
- UAV transmits data to GCS Operator, JTF C2, and Other Assets. Ongoing throughout the aerial reconnaissance mission
- JTF C2, and Other Assets assess UAV-provided data. If they determine a SAR mission is required, they communicate with GCS Operator.
- If SAR mission is required, the GCS Operator commands the UAV to deviate from assigned flight path, transit to, and orbit over SAR area.
- If the GCS Operator receives request from Other Assets or JTF C2 to set up a communications relay node
    - Then GCS Operator configures UAV for communications relay
    - Then UAV performs communication relay for JTF C2 and Other Assets
- Concurrent with the Aerial Reconnaissance Mission, GCS Operator monitors UAS provided system health, UAV flight path, and fuel load

Post-Conditions: The On-station phase ends when UAV reaches bingo fuel, the on-station time period has concluded, or the JTF C2 has commanded the GCS Operator to return the UAV to base

G.1.3.3.1.6. Egress Phase:

Pre-Conditions: The egress phase begins when egress criteria have been met or the GCS Operator commands a return to base and the UAV is on a flight path to return to base (or ship)

- GCS Operator commands UAV to commence egress
- GCS Operator monitors system health
- GCS Operator monitors egress flight path
- Ground Crew prepares recovery device for recovery
- UAV reports arrival at recovery way point
- GCS Operator checks recovery parameters
- If shipboard and recovery parameters are unacceptable, the GCS Operator requests host platform maneuver to achieve safe recovery parameters
- GCS Operator executes auto-land or manual landing
- If auto-landing, UAV executes recovery profile

- If manual landing, the GCS Operator controls the UAV on recovery profile
- If recovery profile exceeds safety parameters, UAV executes autonomous wave-off
- GCS Operator monitors recovery profile
- If recovery profile exceeds safety parameters and UAV has not executed autonomous wave-off, the GCS Operator executes a wave-off
- If the UAV has waved-off, the GCS Operator maneuvers the UAV to intercept recovery waypoint again
- If no wave-off has occurred, UAV executes recovery
- If the UAV is unable to land and bingo fuel is approaching, the GCS Operator will command the UAV to proceed to an alternate landing site if available. If unavailable, the GCS Operator will intentionally ditch the aircraft

Post-Conditions: Egress completes once the UAV is safely recovered

G.1.3.3.1.7. Postflight Phase:

Pre-Conditions: Postflight begins after UAV is recovered

- Ground Crew conducts UAS post-flight checks and services
- Ground Crew assesses UAV and launch and recovery devices (as applicable) for damage
- Ground Crew removes data storage devices from UAV
- Ground Crew power off UAV and launch and recovery devices
- GCS Operators remove data storage device from GCS
- GCS Operators power off GCS
- GCS Operator and Ground Crew debrief mission
- GCS Operator and Ground Crew generate after-action report
- GCS Operator transmits after action report and applicable data from data storage to JTF C2, HN, NGOs, or Other Assets

Post-Conditions: The postflight phase ends once the mission after action report has been completed and transmitted

G.1.3.3.1.8. Failsafe Mode:

The failsafe mode is triggered during any flight phase (Ingress, On-station, or Egress) when either communications with the GCS are lost, when GPS navigation is lost, or both.

- If command and control communication is lost between the UAV and GCS for more than 1 minute: the UAV flies to operator-specified failsafe rally waypoint, and orbits until communication is reestablished. If communication not reestablished within 5 minutes, UAV conducts auto-land near failsafe rally waypoint.
- If loss of GPS occurs, UAV orbits at constant altitude to reestablish a GPS link. If GPS does not return after 1 min, UAV dead reckons to rally waypoint and conducts an auto-land.
- GCS Operator may command a UAV destruction at any time for safety of flight reasons or to prevent enemy from capturing system.

The failsafe mode ends when any one of the following occurs:

- Command and Control Communications are reestablished
- GPS navigation becomes available
- The UAV has completed an auto-land
- The UAV has been command-detonated for safety or to prevent capture.

G.1.3.4. Measures

This DRM is designed to provide the necessary context for a system under development to assess current system capabilities and tactics. Notional key system characteristics include interoperability, reliability, and maintainability. The following measures from the Universal Joint Task List and Universal Naval Task List (JEL+, 2017; DOD, 2008), may be useful for evaluating the effectiveness of the system for meeting mission requirements:

OP 2.7.1 Manage Intelligence, Surveillance, and Reconnaissance (ISR): "Direct, supervise, and guide operational control (OPCON) of intelligence, surveillance, and reconnaissance (ISR) operations supporting the joint force. Intelligence support to operations may be derived from any number and variety of intelligence sources and sensors employed within the operational environment (OE). Full-motion video (FMV) and motion imagery provided by unmanned and manned assets are two examples that may contribute to mission effectiveness of ISR operations. Maintaining cognizance of the availability and capabilities of all sources and sensors employed within the OE, coincidental to ISR operations, and ensuring timely dissemination of collected information in order to affect operational decision-making are essential elements of this task. Implement the ISR CONOPS based on the collection strategy and ISR execution planning. Coordinate ISR operations with the joint force directorate,

intelligence plans section, joint force collection manager, and asset controlling authority to ensure ISR operations are executed in accordance with the intelligence collection strategy" (JEL+, 2017).

M1	Percent	Of unanalyzed information made available to support personnel recovery.
M2	Percent	Of unanalyzed information made available to support time-sensitive targeting.
M3	Percent	Of unanalyzed information made available to joint task force commander for time-critical decision-making.
M4	Percent	Of unanalyzed information made available to joint intelligence support element analysts for production of current intelligence.
M5	Minutes	For FMV asset to establish communications with higher headquarters.
M6	Hours	To identify shortfalls in reconnaissance platforms.
M7	Minutes	To coordinate redirection of ISR assets to meet new collection requirement.
M8	Minutes	To coordinate redirection of ISR assets to meet combatant commander or national collection requirement.

Measure detailed in the Joint Universal Task List to support Task OP 2.7.1 are written specifically to gathering ISR data during a combat engagement, however many of the measures would apply even if not in a combat situation. The above measures have been modified to remove reference to the combat situations.

OP 8.10 Conduct Foreign Humanitarian Assistance: "Relieve or reduce human suffering, disease, hunger, or privation outside the United States and its territories. This task may include surveying the disaster area, prioritizing needs, conducting health assessments, and providing health services, communications, shelter, subsistence, water, engineering support, transportation, firefighting, mass care, urban SAR, hazardous materials response, and energy distribution" (JEL+, 2017).

M1	Percent	Of affected area under control of legitimate authorities
M2	Percent	Of populace that has basic needs met within 24 hours after disaster/incident

OP 8.18 Coordinate with NGOs: "NGOs operate in most conflict areas, areas of instability, and under-governed territory. Cooperation with NGOs can improve Joint Force effectiveness and minimize conflict between the DoD and NGO" (JEL+, 2017).

M1	Number	Of formal contacts with relevant NGOs.

NTA 1.1.2.5 Employ Remote Vehicles: "To operate vehicles such as robots, drones, unmanned underwater vehicles (UUVs), unmanned aerial vehicles (UAVs), and other devices from a local control station. This task includes deployment, launch, control, and recovery operations" (DOD, 2008).

M1	Hours	To respond to emergent tasking
M2	Percent	Of mission time controller remains in communication with remote vehicle
M3	Number/day	Of remote vehicle missions conducted successfully

NTA 1.2.8.2 Conduct Helicopter Landing Zone Reconnaissance: "To confirm historical data through on-site reconnaissance of a proposed helicopter landing zone (HLZ), site, or point" (DOD, 2008).

M1	Time	Force delayed due to late reconnaissance
M2	Number	HLZ's confirmed
M3	Time	Force delayed due to inadequate reconnaissance

NTA 2.2.3 Perform Tactical Reconnaissance and Surveillance: "To obtain, by various detection methods, information about the activities of an enemy or potential enemy or tactical area of operations. This task uses surveillance to systematically observe the area of operations by visual, aural, electronic, photographic, or other means. This includes development and execution of search plans" (DOD, 2008).

M1	Days	From receipt of tasking, unit reconnaissance/surveillance assets in place.
M2	Percent	Of collection requirements fulfilled by reconnaissance/surveillance assets.
M3	Percent	Of time able to respond to collection requirements.

NTA 2.2.3.1 Search Assigned Area: "To conduct a search/localization plan utilizing ordered search modes/arcs" (DOD, 2008).

M1	Hours	From receipt of tasking until search force is in place.
M2	Hours	To respond to emergent tasking(s).
M3	Percent	Of time able to respond to collection requirements.

NTA 4.8.3 Provide Interagency Coordination: "To coordinate all civil affairs with the appropriate U.S. agencies and follow their direction as appropriate" (DOD, 2008).

M1	Number	Of incidents/situations requiring coordination

M2	Hours	To assess situation and define assistance needed
M3	Number	Incidents of failed/ineffective coordination

NTA 4.8.4 Coordinate with Non-Governmental Organizations: "To coordinate civil affairs with appropriate NGOs, including private voluntary organizations" (DOD, 2008).

M1	Number	Of incidents/situations requiring coordination
M2	Hours	To assess situation and define assistance needed
M3	Number	Incidents of failed/ineffective coordination

NTA 5.1.1.1.2.2 Relay Communications: "To pass information which cannot reach its targeted audience directly. This includes the use of aircraft for tactical relay" (DOD, 2008).

M1	Number	Messages relayed.
M2	Minutes	To relay required messages.
M3	Percent	Correct messages received.

Other potential mission measures include: number of sorties, pounds of provisions delivered, number of survivors located, number of people evacuated, and number of damaged facilities identified.

G.1.4. Conclusion

G.1.4.1. DRM Analysis and future recommendations

This DRM has been developed to support answering three questions:

- Can a UAS support a HADR mission?
- What role will a UAS play in supporting a HADR mission?
- How effective will the UAS be in its support of a HADR mission?

Further analysis based on this DRM is recommended to simulate the mission narrative as written, and determine any emergent behaviors of the system.

G.1.4.2. Summary

This DRM has established:

- An operational context, description of the environment and situations in which a system of interest is expected to operate

- An operational narrative containing enough detail to generate multiple operational scenarios
- A sequence of operational activities and interactions between the key system elements and other systems in the environment
- Measures for establishing goals for mission success

G.2. Extracting MP Behavior Models from Mission Narratives

Monterey Phoenix (MP) is a framework for software system architecture and business process (workflow) specification based on behavior models. MP is intended for the use of lightweight formal methods in software and system architecture design and maintenance. It provides an ecosystem for consistency checking tools, reusable architecture patterns, reusable assertions, queries, and tools for extracting architecture views.

In this section, we show an example of how to formalize a natural language behavior narrative consistent with Step 2 in Appendix D. First, the Ingress Phase is selected to scope the demonstration. Actors are next extracted from the narrative, followed by the action each performs. Finally, the MP grammar rules are composed for the Ingress mission narrative.

G.2.1. Scope the Modeling Task

The HADR mission narrative presented in section G.1 consists of eight stages or phases. Not all source documents are as well prepared, however, and if the narrative is lengthy, it is advisable to partition it into segments like mission phases that can be modeled independently, at least initially. Segmenting behavior narratives for modeling enables the developer to focus attention on one part of the problem at a time, developing, verifying and validating (V&V) small models before composing them into larger models for further V&V. For the purposes of this demonstration, the Ingress Phase (reprinted here for convenience) is used to illustrate each step.

Ingress Phase

Pre-Conditions: Ingress Phase begins when the GCS Operator receives a launch clearance from the JTF C2

- GCS Operator receives launch command from JTF C2
- GCS Operator checks launch parameters for safety
- GCS Operator receives launch clearance from host platform
- GCS Operator commands Ground Crew to launch UAV
- Ground Crew commands UAV to launch
- UAV launches
- UAV maneuvers to clear obstacles then maneuvers to reach ingress altitude
- UAV levels off at ingress altitude
- UAV transmits status (payload and systems) and position messages to GCS Operator (on-going throughout each phase of mission)
- If status and position are acceptable, GCS Operator commands UAV to proceed on ingress flight path, otherwise the GCS Operator commands UAV to return to base. Proceed to egress phase.
- UAV follows ingress flight path to reach on-station area and altitude

- GCS Operator monitors UAV status and position during flight path to on-station area and altitude
- The UAV reaches initial on-station waypoint and reports position to GCS Operator

Post-Conditions: The Ingress Phase ends when the UAV reaches the initial on-station waypoint

G.2.2. IDENTIFY THE ACTORS

The narrative contains clear identifiers for the required actors in their Ingress Phase steps. If inconsistent abbreviations, different spellings, or different names altogether had been used to describe these actors, it would have been more difficult for the modeler to discern the actors in play. We highlight relevant actors in the reprinted narrative in green, to help with the extraction process. Figure G.3 shows the actors that were extracted from the narrative, plus one more – Environment – which was not explicitly referenced in the narrative, but it is needed to provide a navigation reference (e.g., from the Global Positioning System). The actors GCS Operator, JTF C2, Ground Crew, UAV, and Environment are all needed to originate or receive commands, information, and other items described in the narrative. We omit the ship actor for simplicity, since the ship only indirectly interacts with the UAV.

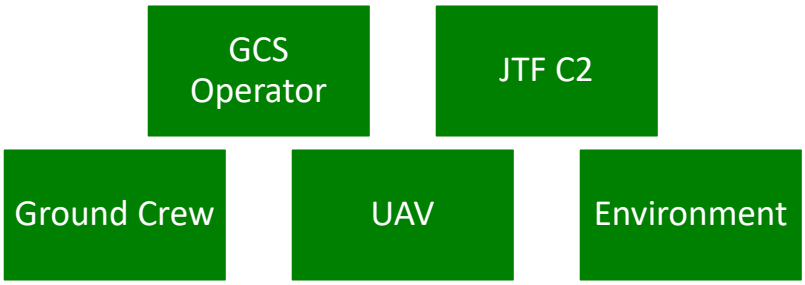


Figure G.3. Actors identified in the HADR Mission Ingress Phase

Since we are starting a new MP model, we write some introductory comments:

```
/*****************************************************************
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)

*****************************************************************/
```

We can now name the schema (MP model), and establish a root event for each actor. The top to bottom order of the roots in the model corresponds with the left to right order in which they will be graphed in the output.

```
SCHEMA UAV_Ingress

ROOT JTF_C2:     ;
ROOT GCS_Operator: ;
```

```
ROOT Ground_Crew: ;
ROOT UAV:        ;
ROOT Environment: ;
```

G.2.2. IDENTIFY THE ACTIONS

Returning now to the mission narrative, we can find the actions being performed by each of the actors. They are highlighted in blue in the narrative reprint below.

Ingress Phase

Pre-Conditions: Ingress Phase begins when the GCS Operator receives a launch clearance from the JTF C2

- GCS Operator receives launch command from JTF C2
- GCS Operator checks launch parameters for safety
- GCS Operator receives launch clearance from host platform
- GCS Operator commands Ground Crew to launch UAV
- Ground Crew commands UAV to launch
- UAV launches
- UAV maneuvers to clear obstacles then maneuvers to reach ingress altitude
- UAV levels off at ingress altitude
- UAV transmits status (payload and systems) and position messages to GCS Operator (on-going throughout each phase of mission)
- If status and position are acceptable, GCS Operator commands UAV to proceed on ingress flight path, otherwise the GCS Operator commands UAV to return to base. Proceed to egress phase.
- UAV follows ingress flight path to reach on-station area and altitude
- GCS Operator monitors UAV status and position during flight path to on-station area and altitude
- The UAV reaches initial on-station waypoint and reports position to GCS Operator

Post-Conditions: The Ingress Phase ends when the UAV reaches the initial on-station waypoint

In order for the actors and actions to stand out so clearly in a written mission narrative, several revisions to the narrative are sometimes necessary. In the best case, the model developer has a good working relationship with the narrative writer and both have the resources to refine the narrative. In the worst case, the model developer does not have a direct line of communication with the mission narrative author or the narrative cannot be revised from its current form. In this case, the best course of action for the model developer is to perform the extraction as best as able and document all assumptions.

Figure G.4 illustrates a sample comment cycle provided by a model developer (Cody Reese, PD21-171 student) on the mission narrative, early enough in its development that a narrative author (Chris Krukowski, PD21-171 student) was able to respond and refine the narrative. The comment bubbles have examples of actual clarifying questions the modeler asked the mission narrative author at the very beginning of a modeling effort in their SI4022 System Architecture course. The comments not only include questions about the narrative structure, but the modeler is also eliciting possible alternative scenarios, which is key to modeling a comprehensive set of use cases scenarios. These are the types of questions a model developer should try to elicit during modeling, especially if the intent is to predict emergent behaviors.

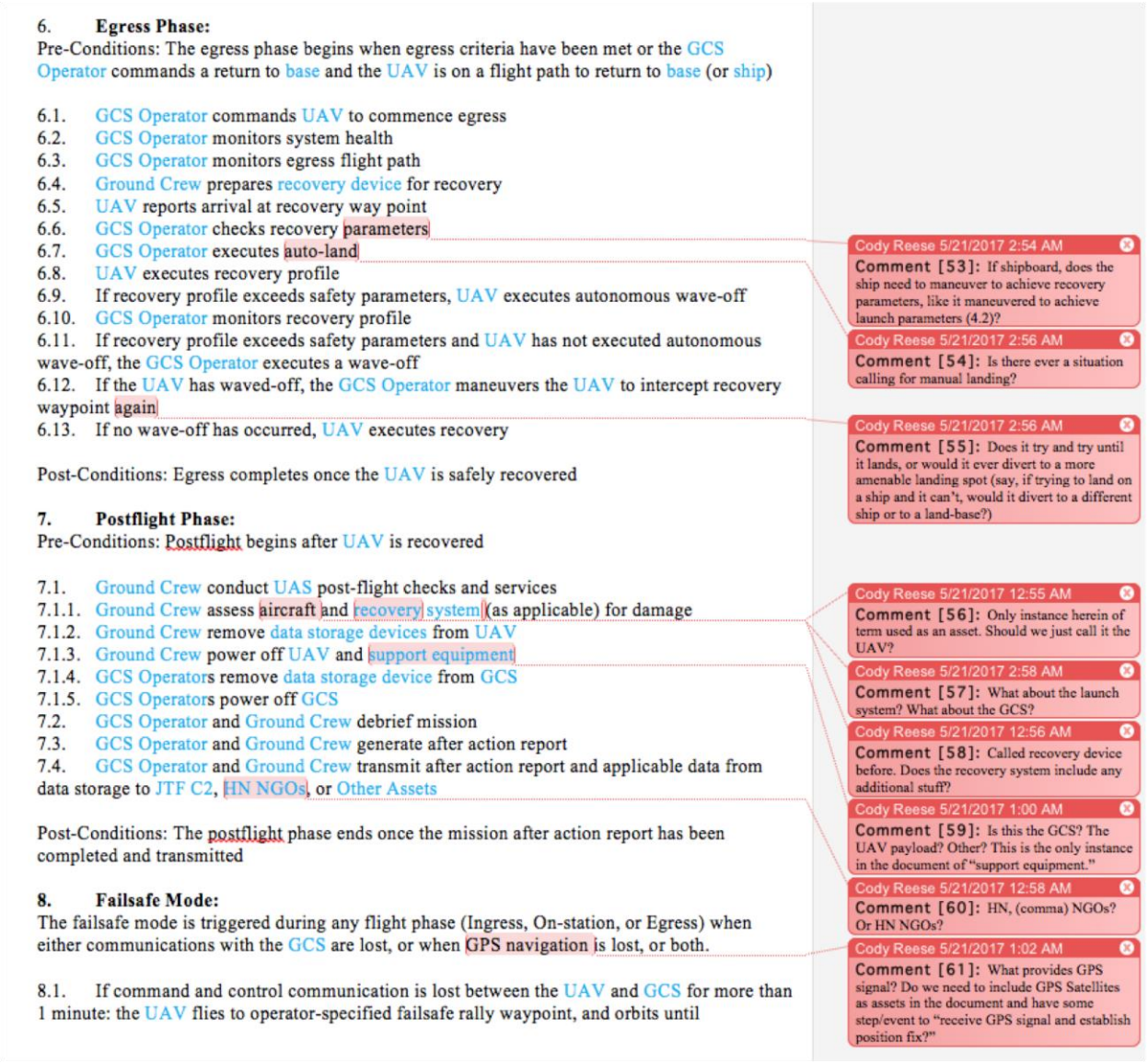


Figure G.4. The model developer marks an early draft of the mission narrative with questions for the narrative author to clarify.

Besides the list of actions evident from the narrative, it is up to the modeler to ascertain the specific actions or states that take place. In doing so, the modeler identifies any actions that may result in the

exchange of additional energy, material, money, or information (EMMI) not explicitly modeled, such as 'JTF Provide Launch Command' that initiates the Ingress Phase, or 'Abort Mission' that is one way in which the Ingress Phase may end. The model developer also identifies actions that may follow (but not directly involve) the exchange of EMMI such as 'Status Acceptable' in which the GCS Operator makes a decision concerning information previously supplied by the UAV. Table G.1 shows how the actions identified in the Ingress Phase narrative may appear as events in MP. The underscore is used to separate words in the event name. A note on event naming conventions: normally a root event name does not need to be repeated in its composite or atomic event names; in this case they are because there are multiple instances of commands being passed between the roots.

Table G.1. Actions and States Identified in Ingress Phase Modeled as MP Events

JTF_Provide_launch_command	UAV_Receive_launch_command
GCS_Receive_launch_command	Launch
Check_launch_parameters_for_safety	Execute_climb
Receive_launch_clearence_from_host_ship	Maneuver_to_clear_obstacles
GCS_Provide_launch_command	Maneuver_to_ingress_altitude
Status_acceptable	Level_off_at_ingress_altitude
Command_UAV_to_proceed_on_ingress	Transmit_status_and_position
Status_unacceptable	Receive_command_to_proceed
Command_UAV_abort	Follow_flight_path_to_reach_ onstation_area_and_altitude
Crew_Receive_launch_command	Reach_onstation_waypoint
Crew_Provide_launch_command	Receive_command_to_abort
UAV_Receive_navigation_reference	Abort_mission

The actions may be directly transcribed into the MP grammar rules as shown in the code on the next page. As a best practice, as the model is modified, the comments are updated with the modifier name, date, and change notes. Commenting throughout the model using /* … */ brackets is also a good practice for documenting assumptions or temporarily removing events as one debugs the model.

The Ingress phase model is deliberately very simple, containing just one alternate path in the GCS Operator and the UAV. Alternate events in each root are stacked on each side of the | symbol and enclosed in parenthesis to denote the range events to alternate.

```
/*****************************************************************
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 C.Reese on 06-10-2017 (added branch in the GCS operator model - status acceptable/unacceptable
 modified by C.Reese on 06-11-2017 Added Environment provide / UAV receive navigation reference
 modified by M.Auguston on 09/05/17
*****************************************************************/
SCHEMA UAV_Ingress

ROOT JTF_C2:         JTF_Provide_launch_command;
/* Assumption: this model described a single UAV launch */

ROOT GCS_Operator:       GCS_Receive_launch_command
                                 Check_launch_parameters_for_safety
                                 Receive_launch_clearence_from_host_ship
                                 GCS_Provide_launch_command
                                 /*Receive_UAV_status_and_position */
                                 (   Status_acceptable
                 Command_UAV_to_proceed_on_ingress |

                                    Status_unacceptable
                 Command_UAV_abort    );

ROOT Ground_Crew:        Crew_Receive_launch_command
                                 Crew_Provide_launch_command;

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 );

ROOT Environment:      Provide_navigation_reference ;
```

G.3. IDENTIFYING EVENT COORDINATION

Now that each actor's event grammar rule has been modeled, we establish the inter-root coordination. Before we do that, however, we can run the model and inspect it for ideas for emergent behaviors that might take place without the coordination constraints. Emergent behaviors are further discussed in the next section. The coordination statements for the Ingress phase are presented below. For example, the first COORDINATE block states that the first launch command comes from the JTF C2 actor, and is received by the GCS Operator, and that in all scenarios this launch command must be provided prior to it being received.

```
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;
```

Note that one of the COORDINATE statements contains alternate events – this is necessary to denote an event in one root that precedes alternate events in another root.

Figure G.5 illustrates horizontal precedence relations added between two roots as a result of adding the first two coordination statements. A best practice from Appendix D is to add one constraint at a time and inspect the results after each addition to ensure the constraint has the intended effect.

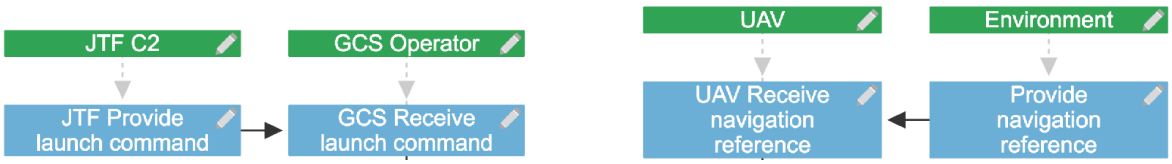


Figure G.5. The solid horizontal arrow is a precedence relation between roots (in green), a result of the first two COORDINATE statements.

G.4. Performing V&V with MP Behavior Models

Having demonstrated the process for extracting MP models from mission narratives, this section focuses on the scenario generation-inspection cycle that constitutes behavior model V&V once an MP model exists. In particular, we discuss and demonstrate example instances of verification and validation issues that can be found in MP models. First, we present a general overview of the types of V&V issues that develop in behavior models. Next, we present sample V&V issues in the context of the UAV Ingress Phase model. Finally, we present some MP modeling heuristics and some typical errors to avoid.

G.4.1. Typical Model Verification and Validation Issues

Table G.2 summarizes verification and validation activities that can be conducted using Monterey Phoenix models. *System verification* is "the confirmation, through the provision of objective evidence, that specified requirements have been fulfilled" (SEBoK authors, 2016). In preparation for verification and validation, the model undergoes a review for syntax errors, typos, transcription errors, and notational errors. For MP in particular, we must also verify we are running the model at a suitable scope for exposing verification errors. The verification team inspects the event traces at each given scope (starting at scope 1) to ensure that the required behaviors are present and that the prohibited behaviors are absent. *System validation* is the procedure used to ensure compliance of any system element with its intended purpose (SEBoK authors, 2016). Using MP, the validation team can inspect the event traces for the presence of extra behaviors that were not specified in the requirements, yet are permitted by the design nonetheless. Some of these extra behaviors may be valid and acceptable, in which case these behaviors are noted and possibly incorporated into the requirements specification. Other extra behaviors may be invalid and unacceptable, in which case new constraints must be written into the model, tested for their effectiveness at removing the unwanted behavior(s), and translated into the requirements specification.

Table G.2. Behavior model V&V activities.

Look for and address:	e.g.,
Syntax errors	missing semicolons, misplaced parentheses
Typographical and transcription errors	misspelled event names, forgotten_underscores, wrong event names used
Notational errors	deviations from required notation and style guides adopted by the organization
Scope errors	running only at scope 1 when errors need scope 2 or 3 to manifest
Required behaviors that are missing	none of the event traces show a certain behavior that was required
Prohibited behaviors that are present	some event traces show a certain behavior that was prohibited
Unspecified valid behaviors	some event traces show wanted behaviors that were not explicitly required
Unspecified invalid behaviors	some event traces show unwanted behaviors that were not explicitly prohibited

G.4.2. APPLICATION OF V&V TO BEHAVIOR MODELS

The MP-Firebird tool, accessed at https://firebird.nps.edu/, is used to perform verification and validation (V&V) of behavior models. The entry page provides an example model and access to MP's documentation (see Appendix D for an overview of the user interface). Figure G.6 shows a snapshot of the MP-Firebird interface with the UAV Ingress model loaded, the MP code typed on the left, the graphs generated from that code on the right, and the Run button and Scope slider bar on the top left. In the case of the UAV Ingress model, only two event traces are generated and they are the only possible outcomes of this model. These traces (after undergoing verification and validation) are shown in following Figures G.7 and G.8. The subsections that follow illustrate types of V&V errors using the UAV Ingress model as an example.

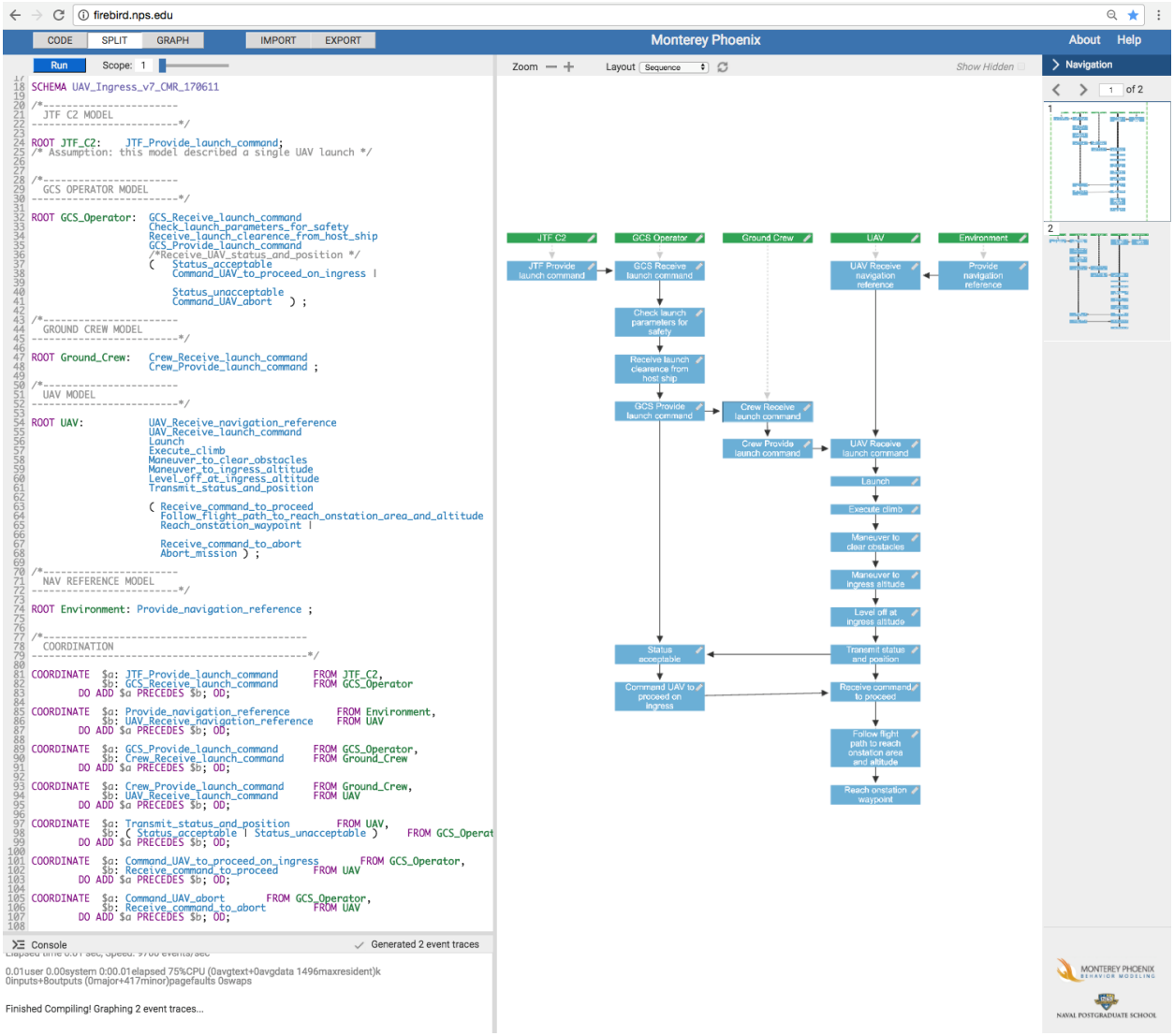


Figure G.6. The UAV Ingress model loaded and executed at scope 1 in MP-Firebird. Code is typed or loaded on the left, and graphs are generated from that code on the right. Since there are no loops in this model, running at a higher scope will have no effect, since there are no iterations to repeat.

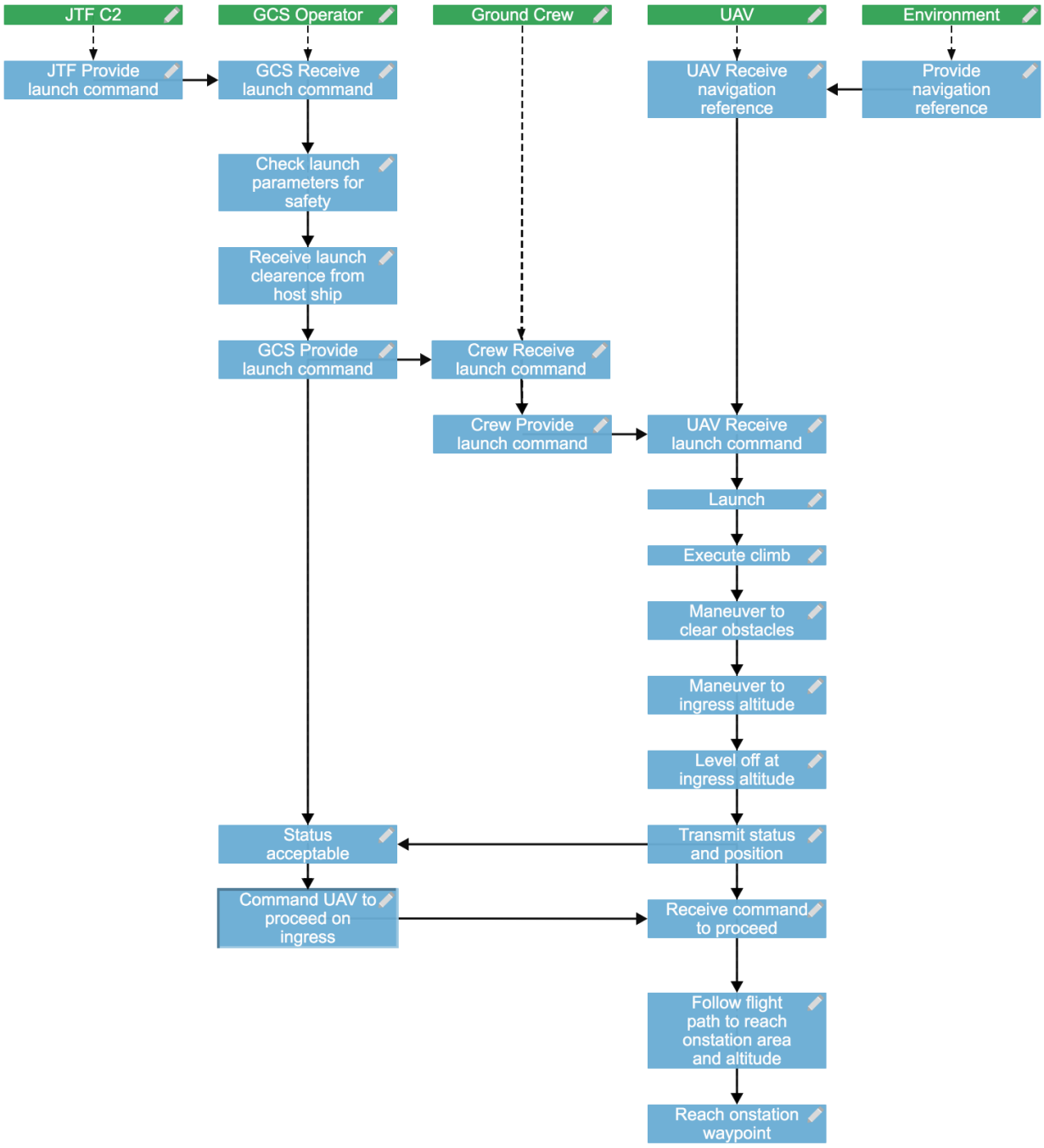


Figure G.7. The first event trace shows the Ingress phase ending with the UAV's arrival at the on station waypoint, the precondition for the On Station phase.

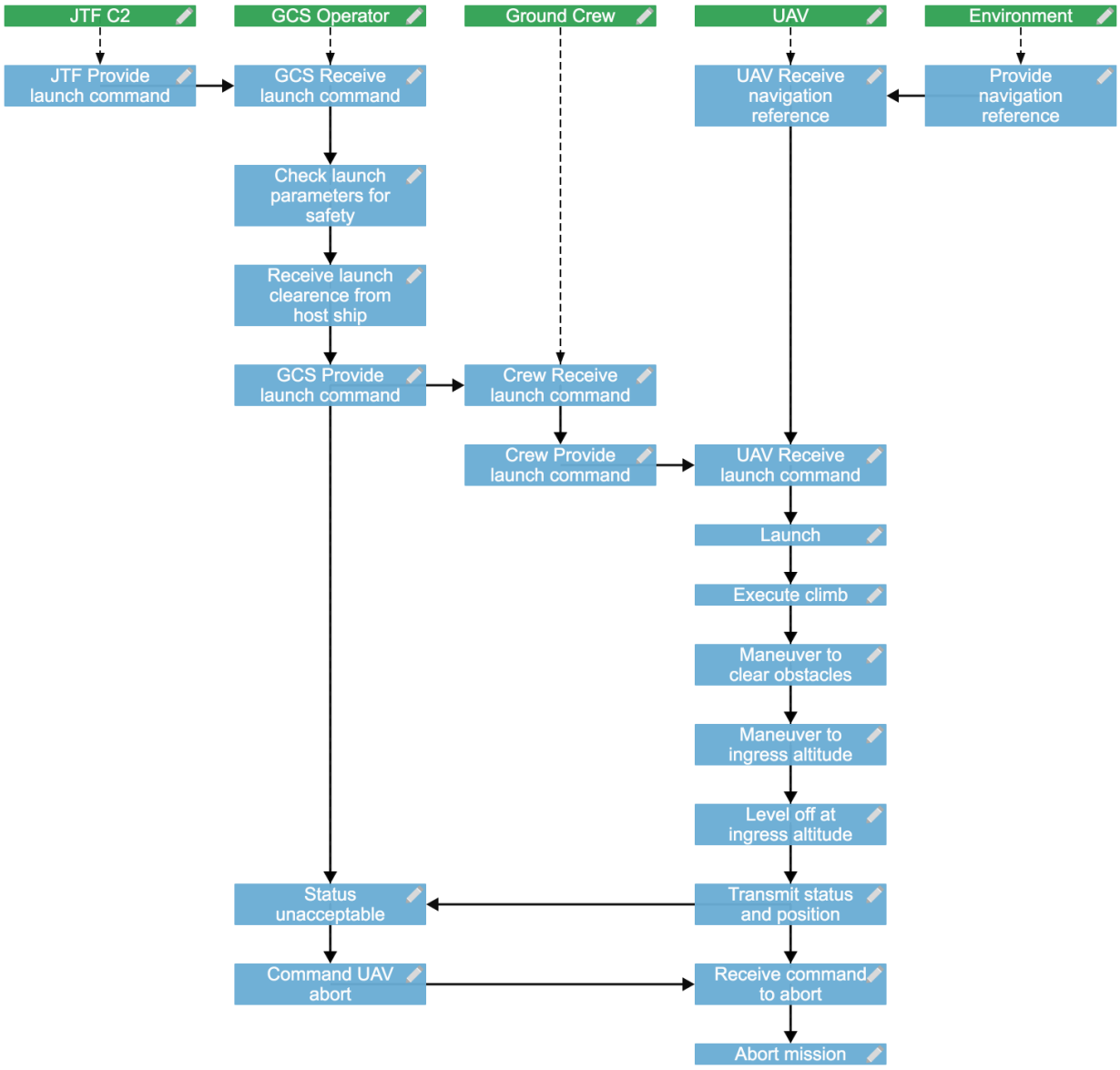


Figure G.8. The second event trace shows the Ingress phase ending with an alternate scenario (mission abort).

For this Ingress model, there are only two possible outcomes. The model will run until the UAS reaches the on station waypoint, or it is commanded to abort.

If we were to expand this model with additional exception cases, we would have a significantly more complex model that may demonstrate many more endpoints or even undesirable behavior resulting from a deficient behavior specification. From the size of the simple models depicted above, it should be clear that breaking the model up into phases is a helpful practice for keeping this expansion of detail manageable for V&V.

G.4.2.1. Syntax errors

The modeler looks for, finds, and corrects syntax errors and typos in real time, after either direct inspection of the MP model, or running the model and spotting the impact of a typo in the generated traces. Common example syntax errors are a missing semicolon (Figure G.9) or a misplaced parenthesis in the MP code.

Figure G.9. A snippet of MP code illustrates an example of real-time syntax checking. In this case, line 32 is flagged because we start declaring a new root before closing the previous grammar rule with a semicolon. The error is corrected by adding a semicolon after event `JTF_Provide_launch_command` in line 24.

G.4.2.2. Typographical and transcription errors

Common typos include misspelled event names, and forgotten underscores in event names resulting in two separate events rather than a single, multi-word event. The real-time syntax checker will flag events that are referenced in coordination constraints but were never defined in any grammar rule.

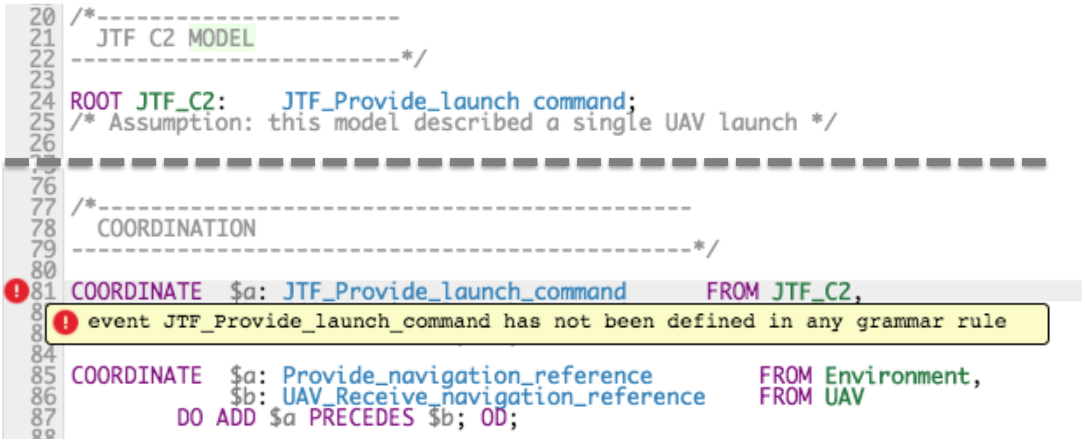


Figure G.10. A typo in the event name on line 24 (a missing underscore) shows up when we try to coordinate the correctly spelled event name on lines 81-83. If the same error is made in an event that does not have coordination, the error will show up in each event trace as two separate events.

G.4.2.3. Notational errors

Another pre-verification activity ensures the absence of deviations from notation or style guidance, such as adhering to a particular language or naming convention. Suggested conventions for modeling with MP are provided in Appendix D, but may vary based on architect or organization preference. Whatever notational conventions are used, be they text-oriented or graphical in nature, they should be established and checked prior to verification so that grammatically ambiguous names or stray lines or labels do not mask the underlying issues with the correctness of the model according to the requirements.

G.4.2.4. Scope errors

An appropriate scope size needs to be used for verification, as well as for validation. A model that looks error-free at scope 1 may present unintentional or unwanted behavior after iterating a few times at scope 2 or 3. We leverage Jackson's (2006) Small Scope Hypothesis to expose most errors on small examples (or small number of loop iterations, in the case of MP).

In the upper left corner of the MP environment, to the right of the Run button (Figure G.6), we are able to specify the scope limit prior to running the model. This allows the modeler obtain all valid event traces within the MP schema up to the specified scope – the upper limit on the number of iterations in grammar rules. Scope must be used with care, as the number of event traces may grow rapidly with increasing scope. In the relatively simple Ingress model, however, we can raise the scope up to two, three, four, or even five and receive the same results, since there is no iteration.

G.4.2.5. Required behaviors that are missing

Identification of missing required behaviors is partly cycling on requirement specification clarity, and partly checking that all required behaviors in the specification are in fact represented in the model. The mission narrative provided in the DRM in section G.1 served as the requirement specification for the Ingress model in MP. Example questions that came up during the Ingress phase modeling were the following:

- Does GCS operator confirm receipt?
- Can anyone else give the launch command?
- Are there situations where [the UAV] wouldn't climb out?
- Does there need to be a follow-on command to the ship to prepare/maneuver to receive the UAV (similar to ship [maneuver] to achieve launch parameters?)

In this project, the students had the opportunity to revise the mission narrative opportunity to reflect the answers to these and other questions (like those shown in Figure G.4) to ensure the most complete description of required behaviors.

The MP model was constructed following the process described in section G.2 and G.3, and then run to verify the presence of all required behaviors. Early on, a student noticed that one of the required behaviors was missing; in fact, an entire scenario outcome was missing, despite the presence of all required behaviors in the model. The cause was a coordination constraint that was forcing the selection of one of the outcomes in all scenarios. In particular, an event that always occurs in one root was coordinated with an event that has alternative events in another root, causing the coordinated alternative to be selected in each and every scenario (Figure G.11). This common error is corrected by specifying alternative events in the COORDINATE statement (Figure G.12).

Figure G.11. An event that always occurs in one root (GCS_Operator) is coordinated with an event that has alternative events in another root (UAV). This is a common anti-pattern that results in the suppression of the alternate events (mission abort), since the coordination requires these events to occur as a pair in every scenario.

Figure G.12. To correct the issue in the model in Figure G.14, corresponding branches of coordinated behavior are established in both the GCS_Operator and the UAV by employing alternative events in the COORDINATE statement.

G.4.2.6. Prohibited behaviors that are present

Requirement specifications predominantly describe valid, desired behaviors or characteristics that the system under design is to have. Until now, it has been extremely difficult to conduct early V&V for so-called "negative requirements," or those that are phrased like "The system shall NOT…". Browsing dozens or hundreds of event traces may be time consuming and error prone. The larger a behavior model becomes, the more automated inspection tools become necessary for verifying the absence of known unwanted behaviors. With MP, we can check a model for the presence of behaviors using *assertion checking*, where one or more behaviors of interest are formally posed as a statement and checked against the set of generated scenarios. The CHECK construct makes it possible to use automated trace monitoring. If the property (a Boolean expression in the CHECK) is not satisfied, the trace will be marked and available for further inspection.

For our UAV Ingress model example, imagine that our customer wants us to verify that there are no scenarios where the UAV is launched without permission from the JTF C2. With only two event traces, it is easy enough to determine that there are no scenarios in which the UAV is launched without the JTF C2 having issued a launch command. If this model were larger, with more UAVs and more scenario variants, it may make sense to use a CHECK construct to automate the search for any such scenario that violates the customer's expectation.

```
CHECK   #Launch                                FROM UAV    <=
                  #JTF_Provide_launch_command     FROM JTF_C2
  ONFAIL SAY("Unauthorized Launch");
```

The above CHECK construct looks at every event trace in the generated set to check that the number of launch events does not exceed the number of JTF C2 commands to launch. Each event trace that fails this check is automatically stamped with an annotation "Unauthorized Launch," which draws the inspection team to look closely at the annotated subset for errors. After the CHECK command is used to find and fix the errors behavior model, the statement can be converted into an ENSURE statement, which is a formally specified requirement.

```
ENSURE #Launch                              FROM UAV    <=
                    #JTF_Provide_launch_command    FROM JTF_C2;
```

Using these statements to detect unauthorized launch makes the assumption that the JTF issues a distinct launch command each time a UAV is launched. It is possible that one launch command may authorize multiple UAVs to launch. In this case, these constraints would have to be made more precise to detect truly unauthorized launches, which will require the modeling of more events involved in the launch authorization sequence. These examples are just to illustrate some simple use cases for CHECK and ENSURE.

Assertion checking with the CHECK clause may be the simplest and most common tool for finding counterexamples of traces that violate some property. Traces violating the CHECK condition may be marked and annotated with a message that provides some hint about the cause. Using assertion checking on much larger models can significantly speed up the verification and validation process.

G.4.2.7. Unspecified valid behaviors

Upon browsing the resulting event traces, the model developer may notice the absence of necessary behaviors that are missing from the specification such as in Nelson's ISS example in (Giammarco & Giles, 2017) or notice the presence of a number of valid behaviors that were not explicitly specified, but are present nonetheless. The latter is often the case for models run at scopes 2 and 3, when the effects of sets and iteration manifest as different possible permutations of events. Since the UAV Ingress model does not have iteration, the reader is referred to the Car Race model (in the menu of preloaded examples on firebird.nps.edu), which shows hundreds of valid behaviors for this system at scope 3. The number of cars participating in the race (one or more), the number of laps driven by each car, and whether or not a car finishes the race are the main variables for which all combinations are tried up to the specified scope. Figure G.13 shows two out of 236 examples of valid scenarios for the Car Race model at scope 3.

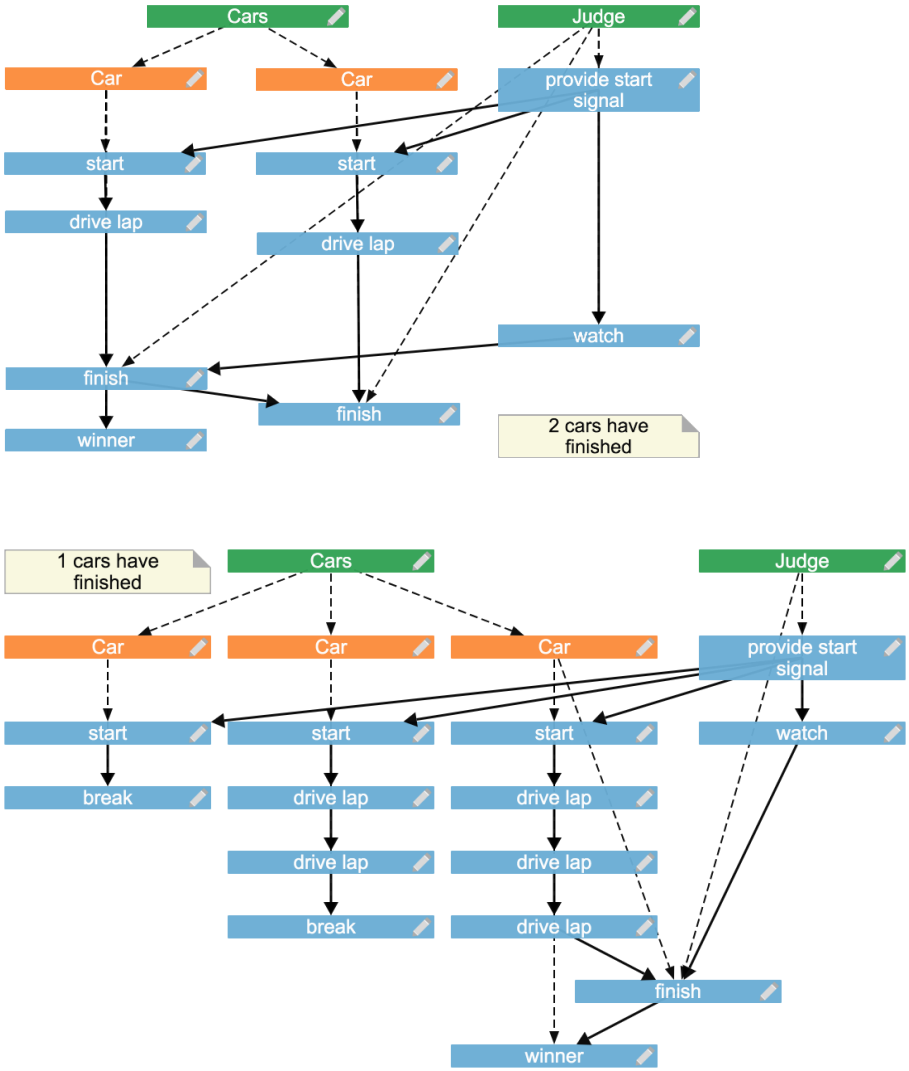


Figure G.13. Two valid scenarios for a car race model. Top: Two cars race, each drive one lap, and then the first one to finish wins. Bottom: Three cars race, two break, and the third one finishes and wins. Many more valid combinations exist; MP is used to generate them all up to the specified scope.

The event traces from the car race model illustrate examples of emergent behaviors (defined in Appendix D). We can detect, classify, predict and control emergent behaviors with MP as discussed in research product [12] in Appendix A. Table G.3 provides an example emergent behavior analysis related to the car race model.

Table G.3. Summary of Emergent Behavior Analysis for Car Race Examples

Example	Figure	Detection	Classification	Prediction	Control
Car Race	G.16 top	Automatic and scope-complete with MP	Weak positive emergence	Two cars race, each drive one lap, and the first one to finish wins. A typical and expected case.	-
	G.16 bottom		Weak positive emergence	The only car to finish wins the race. Not a typical case, but it is permissible.	-

G.4.2.8. Unspecified invalid behaviors

Browsing the event traces may result in finding examples of behavior that are unwanted. For example, the car race model showed some unwanted behaviors, such as cars winning before having driven as many laps as other cars in the race, and multiple cars winning. Unwanted scenarios like these are not always explicit in the requirements; that is why scenario generation with MP is useful for explicitly and formally identifying them. The following constraints prevent the emergence of such unwanted behaviors.

```
…
/* everybody who finishes drives the same number of laps */
ENSURE FOREACH DISJ $c1: Car, $c2: Car
                        (#finish FROM $c1 == 1 AND #finish FROM $c2 == 1 ->
                         #drive_lap FROM $c1 == #drive_lap FROM $c2)
…
/* there always will be at most one winner */
ENSURE #winner <= 1;
…
```

After fixing the issue in the MP code, we run the trace generation again, until our expectations are satisfied and all behaviors that emerge are valid.

There did emerge an interesting unwanted behavior in the UAV Ingress model. The following description is an excerpt from (Reese, 2017), a PD21 student term paper analysis of the UAV Ingress model for SI4022 System Architecture. PD21 Student Anthony Constable also contributed to the UAV Ingress model.

An early run of the model showed a scenario where an unanticipated “abort” command was issued. Early model simulations [generated] a scenario during which the system reported acceptable status and the operator, without provocation, commanded the system to abort the mission. This was flagged as a possible emergent behavior and inspired reflection on the conditions which would cause such an incident. We rationalized several possible scenarios which could result in this behavior.

The first such scenario [that could make this a valid scenario] is one in which the GCS operator receives an indication from a third party that the system is not in acceptable status, despite internal reports to the contrary. It is feasible that an external observer detects something wrong with the system, which the system itself has not detected either due to limited sensor capability (it can't detect the type of failure), timing (the observer detected the problem before it developed to a level where the system could detect it), or other (e.g. sensor failure).

The second scenario is one in which the system has detected an issue, but needs to alert the GCS operator in [an alternate] way. The UAV could execute a maneuver such as a wing-wave or other pre-determined flight pattern which would alert the GCS operator of an issue...

The third possible scenario is pure operator error. It is feasible that the GCS operator could trigger the abort command by accident. Perhaps the exhausted and delirious pilot set his coffee cup down on the big red "ABORT" button. This scenario [causes the model developer to have an idea] for a multi-step abort sequence in which the system requires secondary validation and confirmation of the abort command; "are you sure you want to abort the mission?"

On inspection of the model, it was determined that the unanticipated abort command was a result of a modeling error. The script essentially forced the scenario to occur because the abort command was coded as a process step as opposed to a conditional action based on the system status. The author re-coded the model to make the abort command conditional on a "status unacceptable" notification. Further simulations revealed that the system behaved as expected with the revised code, however the mission narrative and system model should be revised to include other alert pathways for the GCS operator and/or ground crew.

[Figure G.14] displays model outputs before and after the updates described above. The early model output is shown on the left, and the updated model output is shown on the right. Note the highlighted areas at top right of the model where the system is now interacting with the environment for a navigation reference [an error of omission], and at the bottom left of the model where the abort command is now contingent on an unacceptable status condition.

The error detected by the student was a verification error that led to a model correction, but the commission of the error in this case also exposed a model validation issue that generated ideas for requirements that may be necessary to control the situation, should it occur. What if the UAV is experiencing a malfunction that prevents it from transmitting its status through the normal channels? Experience has shown that it is far better to consider such circumstances prior to implementation and deployment than to be forced to consider ways to deal with such events as they are unfolding.

Table G.4 provides an example emergent behavior analysis related to the UAV ingress model analyzed by Reese (2017).

Table G.4. Summary of Emergent Behavior Analysis for UAV Ingress Examples

Example	Figure	Detection	Classification	Prediction	Control
UAV Ingress	G.17 left	Automatic and scope-complete with MP	Strong positive emergence	The UAV gives the appearance of an acceptable status, but the operator commands the UAV to abort the mission.	-
			Strong negative emergence	The UAV's status is acceptable but the operator erroneously commands the UAV to abort the mission.	Provide adequate rest and training for operators to reduce the likelihood of occurrence of this scenario

Many other examples of unspecified invalid behaviors detected with MP are provided in (Giammarco & Giles, 2017) and in research products [12] [13] [14] in Appendix A, along with elaboration on emergent behavior analysis using MP.

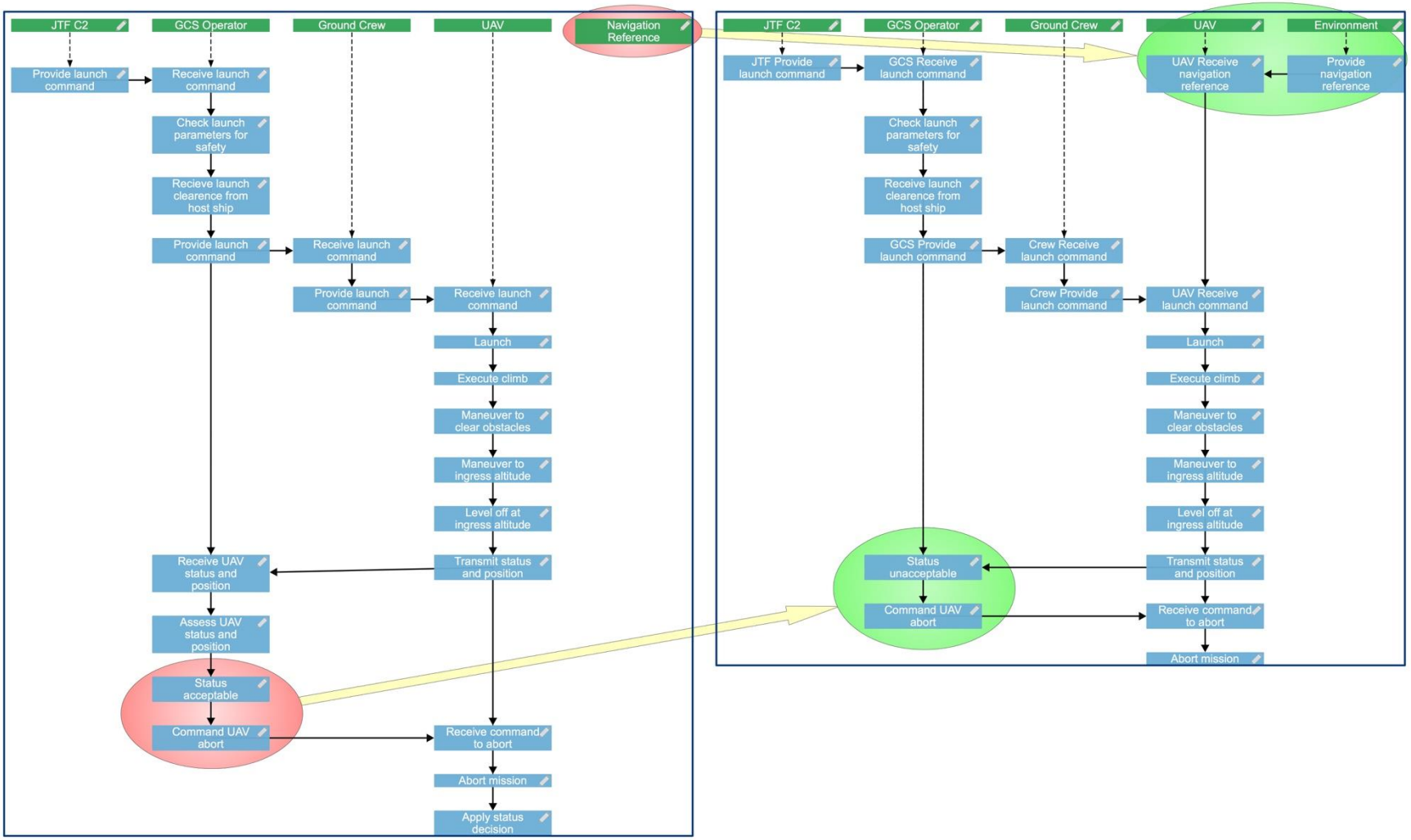


Figure G.14. Side-by-side comparison of ingress model before (left) and after (right) updates. From (Reese, 2017).

G.4.3. MP MODELING HEURISTICS

As model developers gain experience with the basic concepts of MP and start to use it for behavior model verification and validation, they should refer to these MP modeling heuristics to improve the efficiency of trace generation and avoid some typical MP modeling errors.

Event trace generation time may be one of the main concerns when using MP tools. With scope increase it may grow rapidly. Generation time depends on several factors.

- The number and complexity of composite and root events.
- The number and size of traces generated for each root/composite event (this information is available in Firebird's Console window).
- The number of composition operations and their place within the MP code.
- The structure of composition operations (in particular, the nesting within COORDINATE).
- The ordering of root events in the schema's code.

As described in Appendix D, the derivation process is based on the search tree (selection of trace segments for assembly top-down and left-to-right). Pruning the search tree as early as possible is the main principle for speed-up. Here are some heuristics that may help to reduce the generation time.

G.4.3.1. Use scope with caution

The number of traces (and trace generation time) may increase dramatically with the scope. Start MP model's testing/debugging with scope 1 to detect the initial obvious mistakes, and proceed to the higher scopes with caution. In many cases small scope (up to 3) is sufficient to detect/fix most of bugs and to verify all event traces within a scope with the CHECK constructs (Small Scope Hypothesis).

G.4.3.2. Use composition operations as early as possible

COORDINATE and SHARE ALL require coordinated threads of events to have the same size. If this condition does not hold, the derivation process backtracks one step back and picks up another root/composite event segment. When performed early in the derivation, it can prune a significant part of the search tree. Each composition operation works as a powerful search tree's pruning tool. ENSURE also acts as a filter, rejecting an assembled trace segment when it violates a context condition (assertion).

For the schema's trace assembly, root event segments are picked up in the order of root event rule appearance in the schema. Place a composition operation in the MP schema code immediately after all participating root event rules.

G.4.3.3. Use BUILD blocks

Root and composite event segments are derived before the schema's trace assembly begins. Composition operations placed within BUILD blocks may reduce the number of segments derived for composite/root event, and as a result reduce the total search. This is yet another application of the general principle: "prune the search as early as possible".

G.4.3.4. Optimization of COORDINATE operations

Event thread selection for coordination in COORDINATE or SHARE ALL is time-consuming. If the same event thread participates in several coordination operations, it makes sense to merge them. For example:

```
COORDINATE      $x: a FROM A,
                        $y: b FROM B
                DO ADD $x PRECEDES $y; OD;

COORDINATE      $x: a FROM A,
                        $z: c FROM C
                DO ADD $x PRECEDES $z; OD;
```

These can be merged as:

```
COORDINATE      $x: a FROM A,
                        $y: b FROM B,
                        $z: c FROM C
                DO      ADD     $x PRECEDES $y,
                                $x PRECEDES $z; OD;
```

The merge eliminates the repeated processing of the coordination thread for A. Notice that ADD may process several relations.

G.4.3.5. Beware of asynchronous coordination

Use asynchronous coordination (event reshuffling with <!> or <!CHAIN>) only when it is necessary for capturing the requirements. An example of Publish/Subscribe architecture model in the MP Manual section 2.8 uses <!> event reshuffling to try all possible permutations of Register/Unsubscribe pairs. Such coordination may cause dramatic increase in the number of generated traces (and the generation time), since MP will generate all possible permutations of events to coordinate with other sources. If the selected event set has N events, then there are N! possible permutations. Since we expect traces to be generated for a reasonably small scope, the size of the selected event set will be also small and, correspondingly, the number of permutations also is expected to be modest.

G.4.3.6. Coordination and iteration

Coordination of several event threads requires that the numbers of selected events in each thread are equal. If selected threads have different number of events, the coordination fails, the trace under derivation is rejected, and the derivation backups and proceeds with the next step.

The inconsistence between the numbers of events in the coordinated threads may be a typical MP coding mistake. For example, in the following snippet, a valid trace will be produced only once, when the number of B is precisely 1.

```
ROOT R1: A;
ROOT R2: (* B *);
COORDINATE $x: A, $y: B
  DO ADD $x PRECEDES $y; OD;
```

G.4.3.7. Coordination and alternatives

Another typical mistake with coordination may appear when the event selected for a coordination thread appears as an alternative. Then only traces containing that event will be accepted for coordination, but traces that select other alternatives will be rejected. Here is an example.

```
ROOT R1: ( A | B | C);
ROOT R2: D;
COORDINATE $x: B, $y: D
  DO ADD $x PRECEDES $y; OD;
```

Only traces for root R1 containing B will be selected, traces for R1 containing A or C will be rejected because root R2 will have only traces with D. The solution for this issue (if we want to coordinate only B and D) is to make the coordination conditional on the presence of B.

```
ROOT R1: (A | B | C);
ROOT R2: D;
IF #B > 0 THEN
  COORDINATE $x: B, $y: D
    DO ADD $x PRECEDES $y; OD;
FI;
```

G.4.3.8. Synchronizing iteration cycles and coordination

Coordinated events may appear inside iteration as alternatives. There are at least two options for coordination.

Option 1. Coordinate event pairs even without any concern whether they appear in the same cycle of iteration or not. The following example illustrates this. Notice that the default event sequence (the order of event appearance during the derivation) is used for event selection in coordination threads.

```
SCHEMA S1
ROOT R1: (+ (A | B) C +);
ROOT R2: (+ (D | E) F +);

COORDINATE $a: A FROM R1, $d: D FROM R2
  DO
        ADD $a PRECEDES $d;
  OD;

COORDINATE $b: B FROM R1, $e: E FROM R2
  DO
        ADD $b PRECEDES $e;
  OD;
```

It may result in a trace like the one shown on the left of Figure G.15.

Option 2. Coordinate event pairs, but only if they appear in the same iteration cycle. If selected alternatives within the same cycle cannot be coordinated, this trace is rejected.

```
SCHEMA S2
ROOT R1: (+ (A | B) C +);
ROOT R2: (+ (D | E) F +);

COORDINATE $a: (A | B) FROM R1, $d: (D | E) FROM R2
  DO
          IF $a IS A AND $d IS D OR $a IS B AND $d IS E THEN
                  /*        This pair is selected from the same cycle,
           since the default event sequence is used for
           coordination threads */
                  ADD $a PRECEDES $d;
          ELSE REJECT; /* otherwise reject the trace under derivation */
          FI;
  OD;
```

Now the coordination is synchronized with the iteration cycles, as shown on the right of Figure G.15.

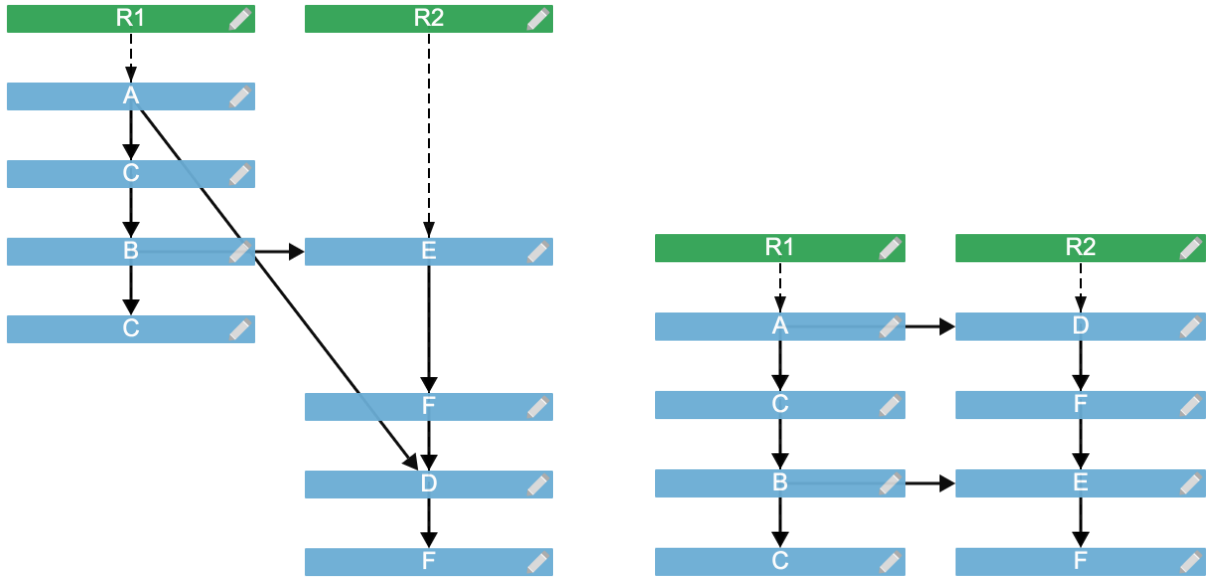


Figure G.15. (Left) Coordination of event pairs even without any concern as to whether they appear in the same cycle of iteration or not, and (right) coordination of event pairs only if they appear in the same iteration cycle.

G.5. SKYZER MISSION MODELING

The Skyzer model is a notional model based on a fictitious mission for small package delivery in a search and rescue context developed on RT-195 and NAVAIR. Skyzer is being used to test MBSE technologies' potential to reduce the lifecycle cost and time to fielding of acquisition systems. This section presents the use of Monterey Phoenix (MP) to model a specific scenario excerpted from the Skyzer IM20 Mission Model documentation, showing how to convert a SysML model into an MP model to leverage the scenario generation and V&V capability of MP. The SysML model is expanded with alternative possibilities not addressed in the original model to provide additional scenario variants. The Skyzer system models demonstrate the Navy-developed MP approach and tool as a candidate for integration into the Naval Enterprise Modeling Environment based on its ability to help modelers find overlooked requirements whose absence results in unwanted behaviors in the modeled design.

G.5.1. SOURCE DATA

The main mission in the Skyzer case study features a UAV-enabled capability for supporting search and rescue operations. The OPSIT is as follows: A family traveling from Puerto Rico to Florida in a 1984 Catalina 36 experience an emergency at sea. The 40-year-old father is diabetic and has run out of insulin. Because he is the main sail operator, the family is stuck at sea until medical attention can arrive. The usual first responder in this situation, a USCG Cutter, is on another mission four hours distant from the last known location point of the sailboat's last communication beacon. The USS Fitzgerald, however, is 75 nm from the last known location point of the beacon, has the needed medical supplies, and also has UAV capabilities to transport the supplies in a timelier manner than the USCG Cutter. Environmental variables include range, sea states, weather, precipitation, visibility, and time of day.

To support this case study, system architects developed a baseline scenario that has the UAV fly out to the sailboat and drop off the medical supplies to sustain the patient until the USCG can arrive to transport the family to safety. The successful case scenario has been modeled in SysML, in which the stranded patient received the needed supplies within a time line that avoided serious medical complications from the lack of insulin, such diabetic coma.

The scenario '**Non-Combatant Operations - Scenario 1'** was originally documented using a SysML behavior model authored by Nataki Roberts using Cameo System Modeler. The produced SysML behavior diagram is shown in Figure G.16 (two page spread).

In short, this scenario describes a civilian vessel in need of medical supplies with the US Coast Guard not able to respond in time to prevent a medical emergency, but a Navy ship equipped with a UAV and the necessary medical supplies can deliver the needed medical supplies in a timely fashion.

The model describes the behavior of the various actors, including: the Rescuee (Victim, EPIRB), Air Force, Rescue Coordinator, Mission Commander, UAV Operator, Ground Crew, INMARSAT, GPS SAT, Ship Utilities, Control Station, Air Vehicle, and Launch and Recovery Sysem (UCARS, RAST).

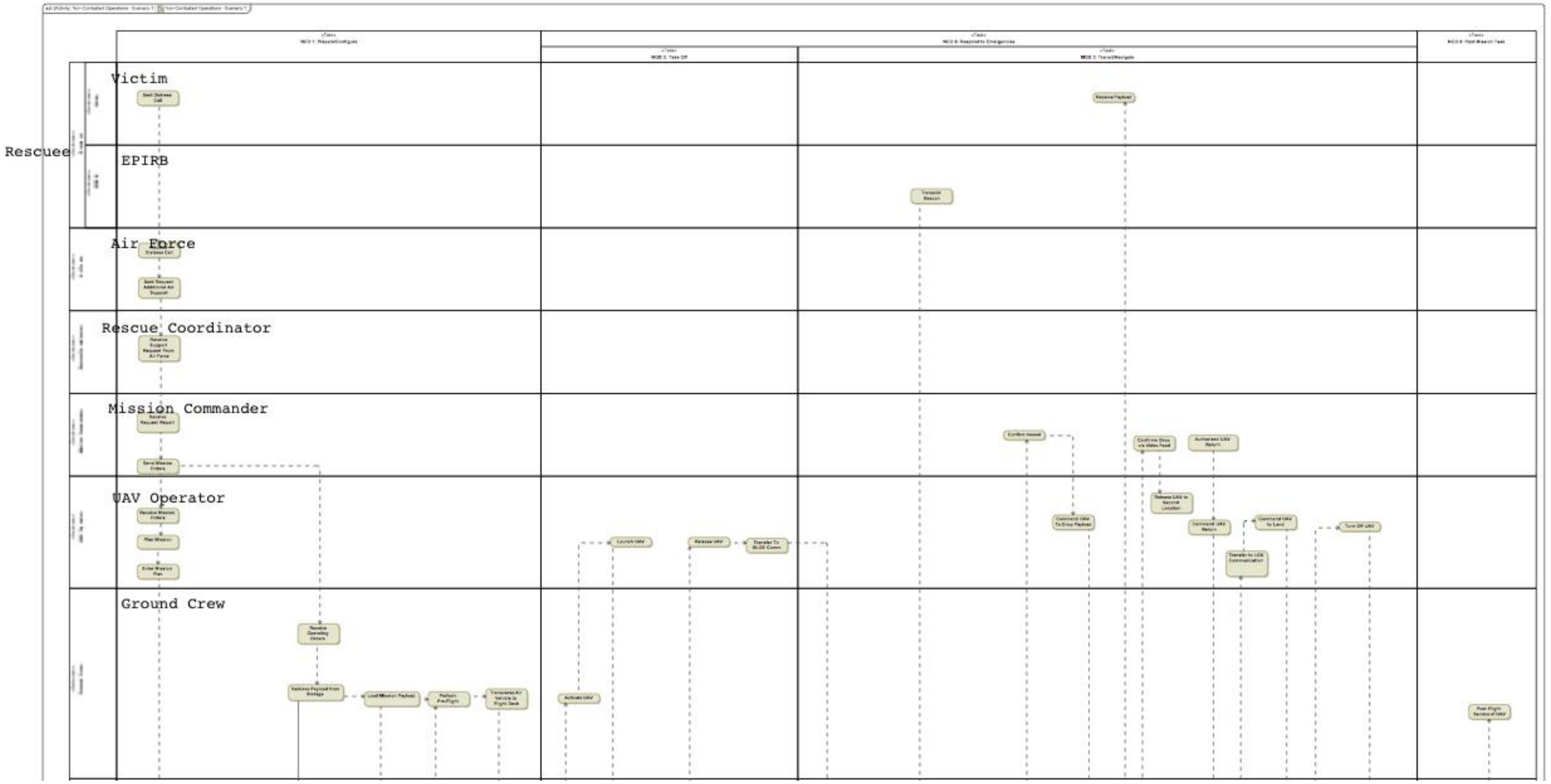


Figure G.16. SysML diagram for 'Non-Combatant Operations - Scenario 1' (upper half)

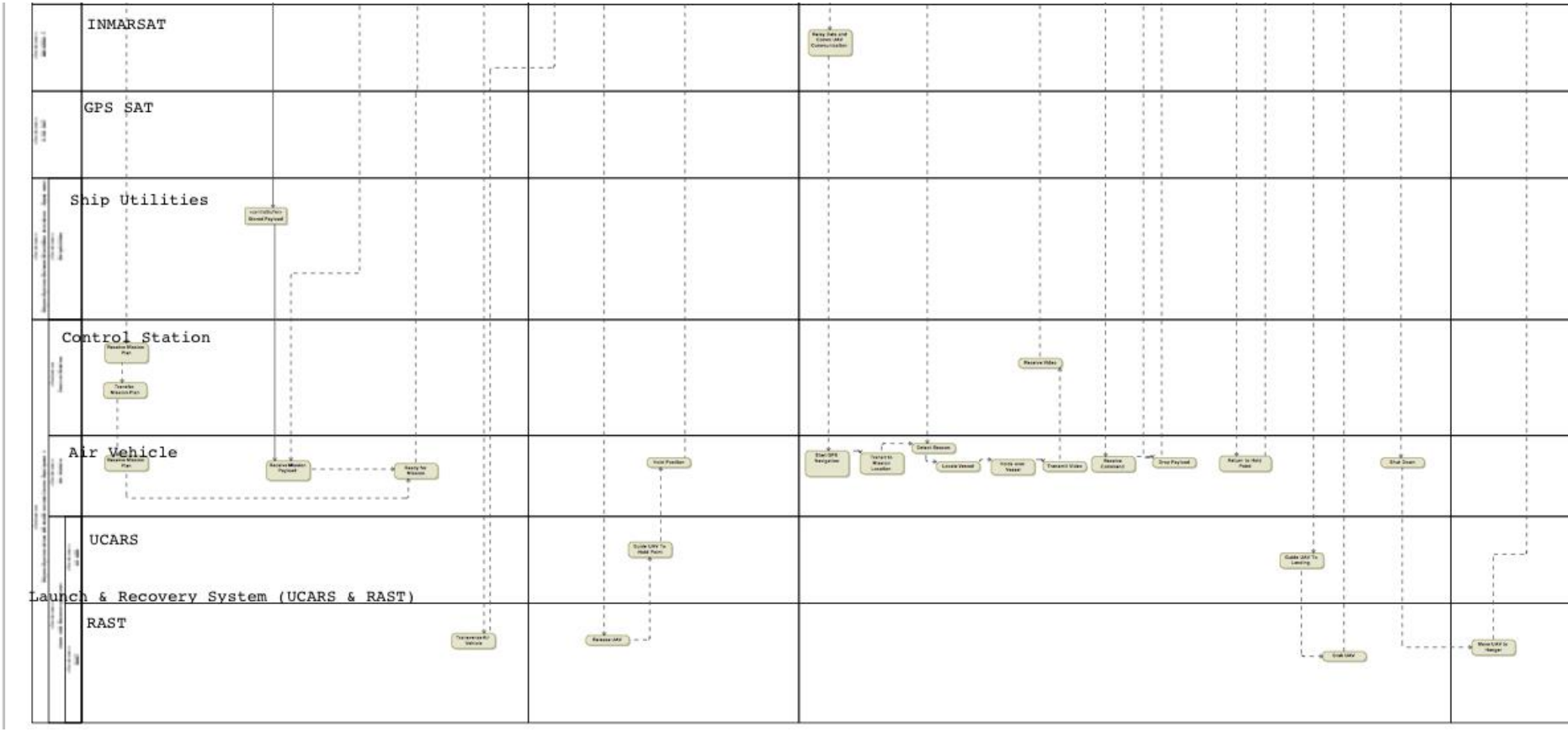


Figure G.16 (continued). SysML diagram for 'Non-Combatant Operations - Scenario 1' (lower half)

G.5.2. EQUIVALENCY DEMONSTRATION

The SysML diagram in Figure G.16 was used as the specification for an equivalent MP model. The actors, down the left side of Figure G.16, became root or composite events in MP, while the actions, shown in the shaded ovals of Figure G.16, became atomic events in MP (Figure G.17). The arrows connecting activities in the SysML diagram were used to create precedence rules for the MP events. The resulting MP model is shown enlarged in Figure G.18.

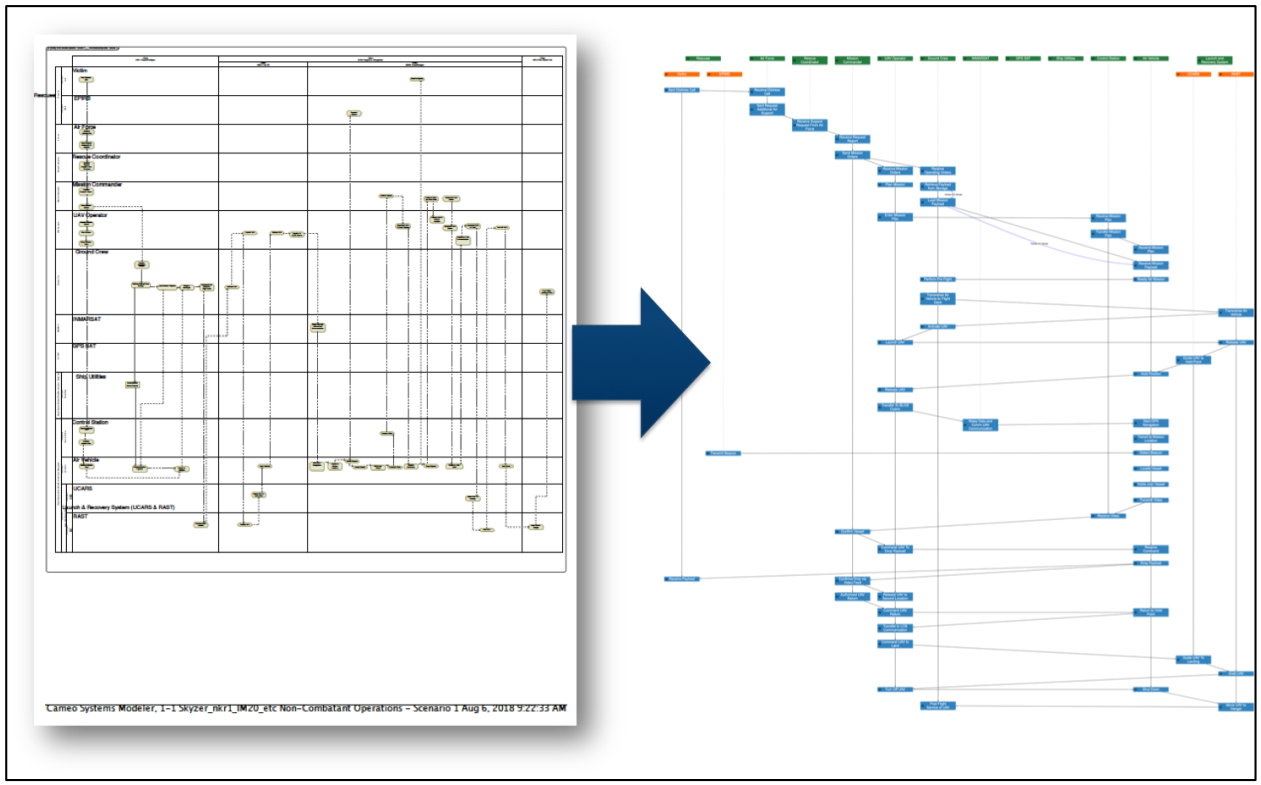


Figure G.17. The SysML activity model was first manually converted into an MP model to demonstrate logical equivalency.

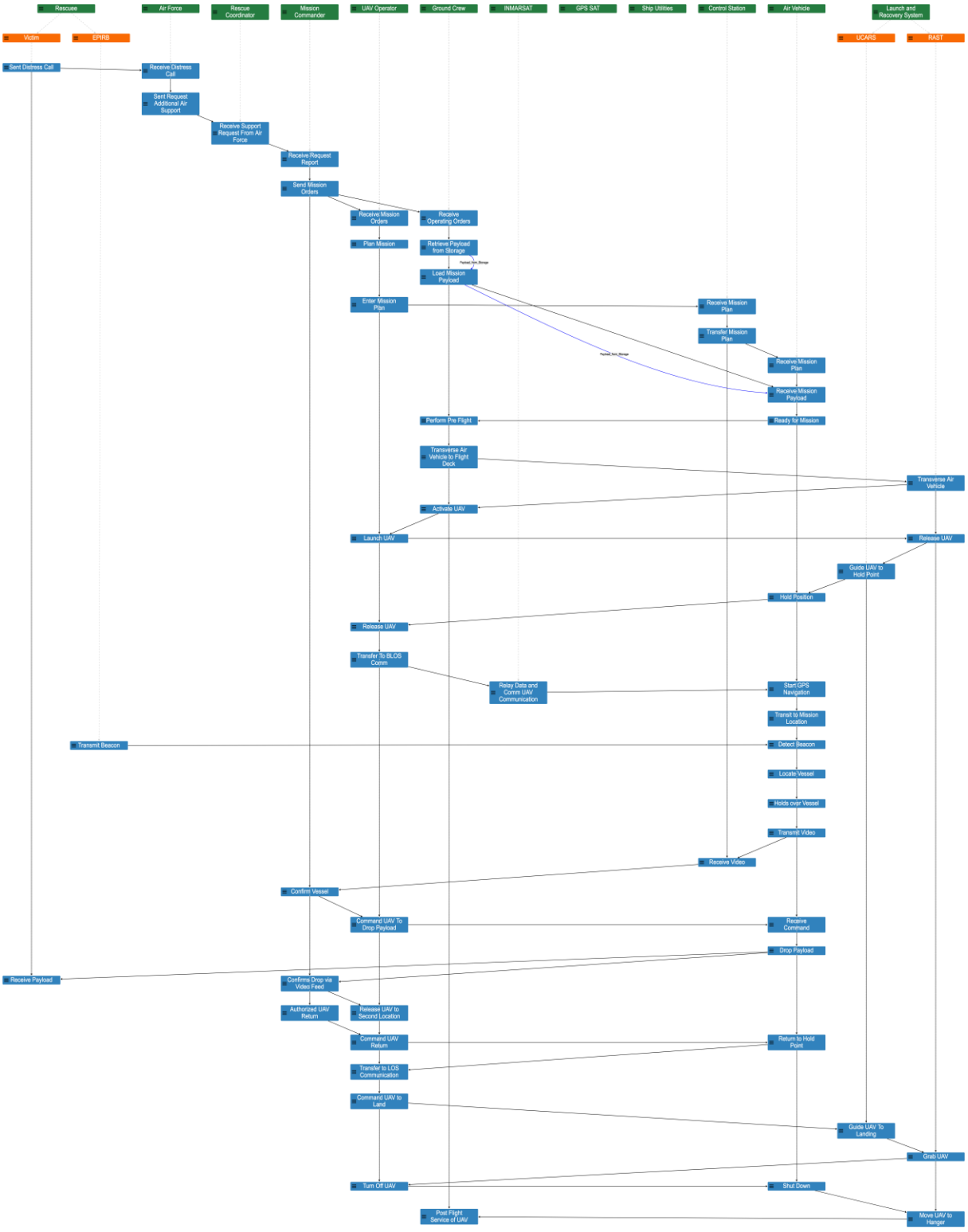


Figure G.18. The MP equivalent of the Skyzer SysML model. Root events are shown in green, composite events in orange, and atomic events in blue. Inclusion is shown with dashed arrows between events, and precedence is shown solid arrows between events.

G.5.3. MODEL SEGMENTATION

As can be seen in Figure G.17, this model is fairly large and complex. To better understand and evaluate the model, the model was segmented into 4 parts or phases (Figure G.19). Reducing the complexity of the model into smaller phases allows for more in depth analyses, as well as verification and validation (V&V). The actors that are inactive during a phase were also omitted to further simplify the phase models.

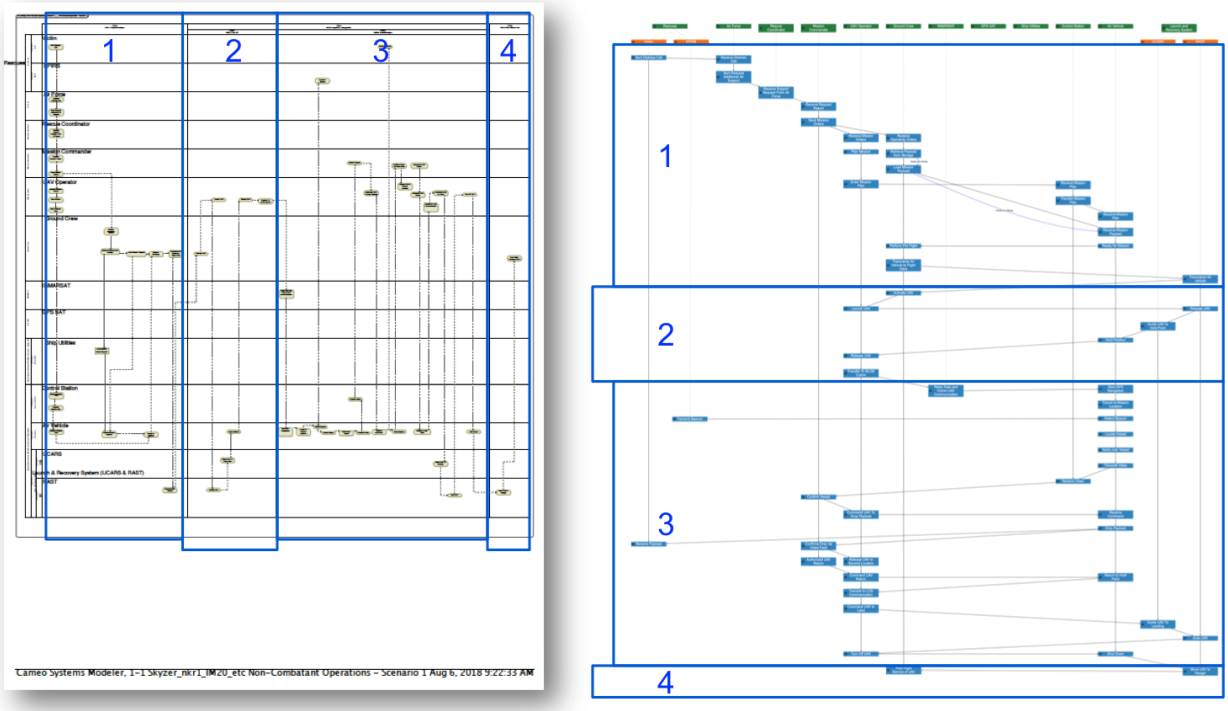


Figure G.19. Segmentation of the model into four phases enables more room for eventual elaboration on alternative behaviors.

Phase 1: NCO 1: Prepare/Configure Task

This phase includes all the actions before the launching of the UAV. The phase 1 event trace is shown in Figure G.20. Directions for downloading the code for all following models are located in Appendix F.

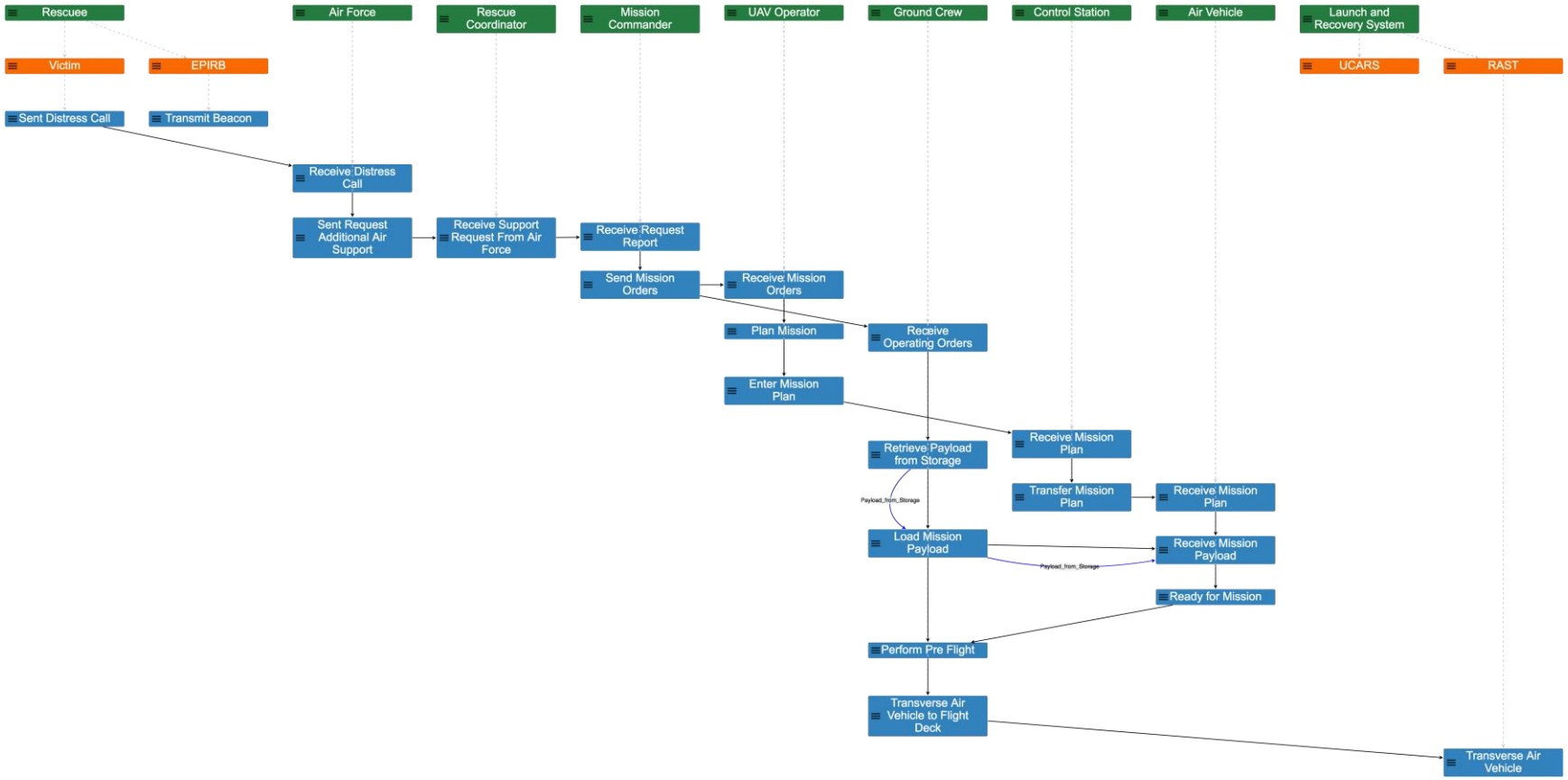


Figure G.20. Phase 1, equivalent to NCO 1: Prepare/Configure Task in Figure G.16

Phase 2: MOB 2: Take Off Task

This phase includes all the actions before the UAV transits to the mission location. The phase 2 event trace is shown in Figure G.21.

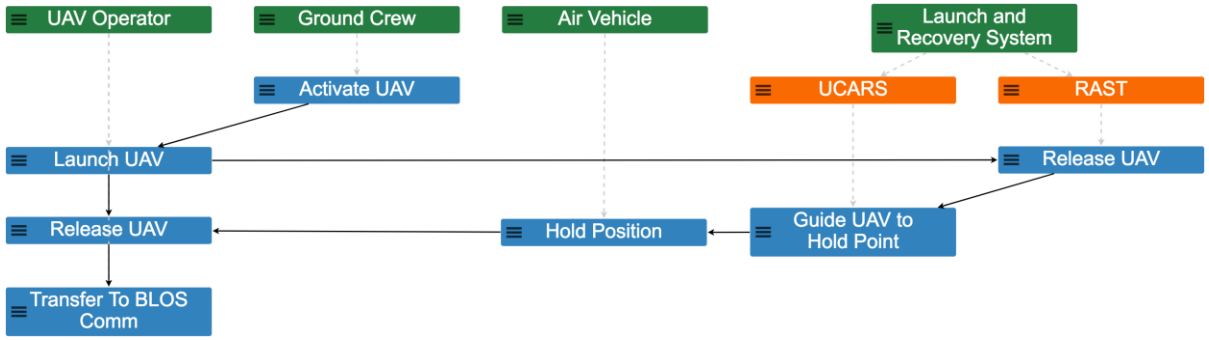


Figure G.21. Phase 2, equivalent to MOB 2: Take Off Task in Figure G.16

Phase 3: MOB 3: Transit/Navigate Task

This phase includes all the actions while the UAV is on station up until the UAV has returned to the ship. The phase 3 event trace is shown in Figure G.22.

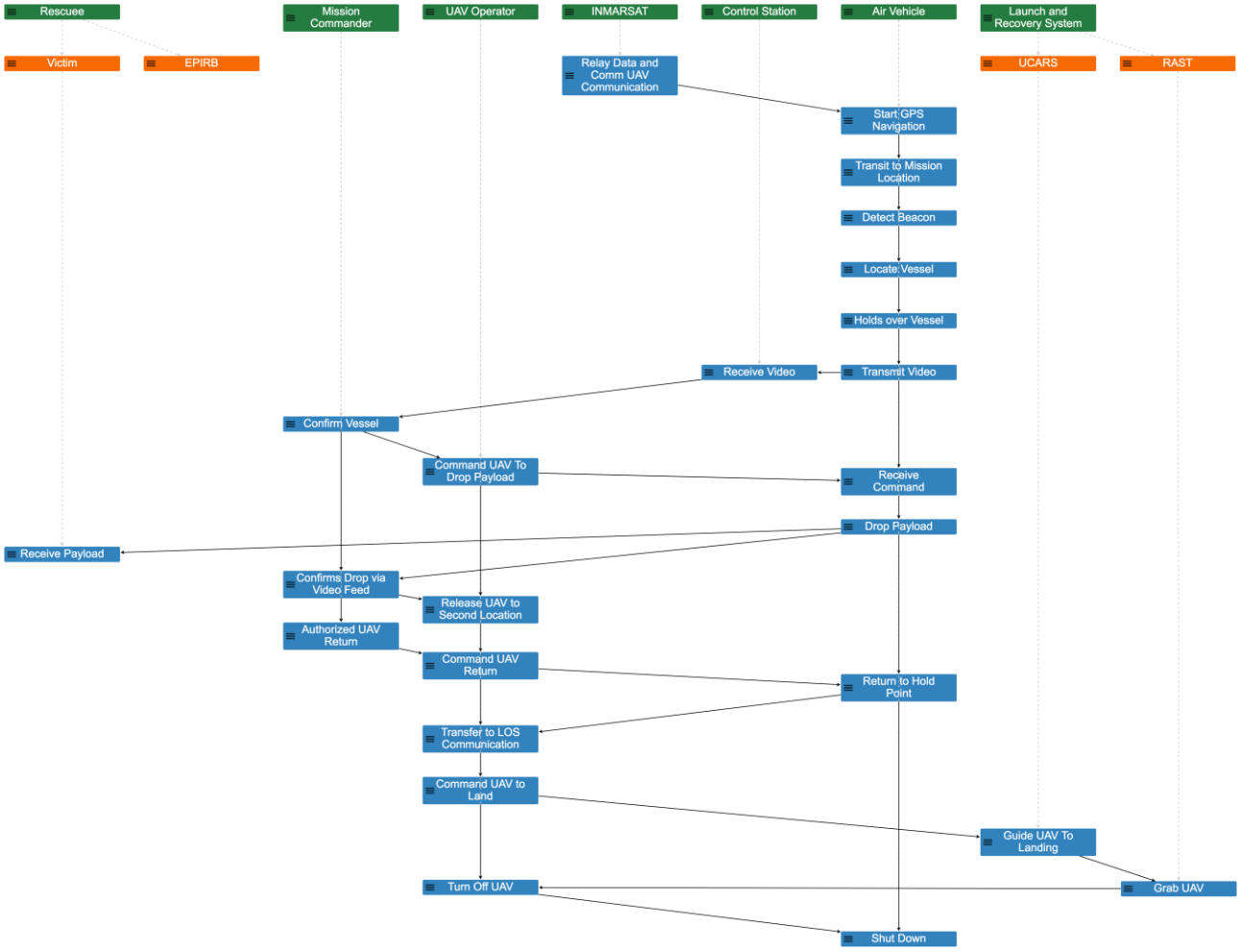


Figure G.22. Phase 3, equivalent to MOB 3: Transit/Navigate Task in Figure G.16.

Phase 4: NCO 8: Post Mission Task

This phase includes all the actions after the UAV has returned to the ship. The phase 4 event trace is shown in Figure G.23.

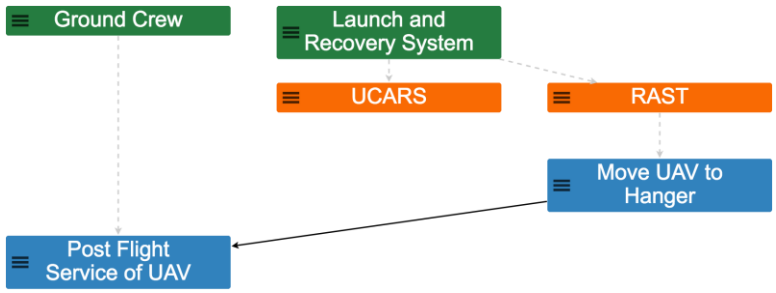


Figure G.23. Phase 4, equivalent to NCO 8: Post Mission Task in Figure G.16.

G.5.4. MODEL ELABORATION

With the SysML model represented in MP code, the next step was to select and expand one of the segmented models. The phase 3 was model selected for expansion. The expansion process involves a systematic consideration of alternatives and factors beyond the scope of the baseline scenario, in which the rescue package is delivered to the rescuee by the UAV in a timely fashion.

The first alternative that was added was the possibility that when dropping the mission payload (the medical supplies), the drop could be on target, or the drop could miss the target resulting in mission failure (the medical supplies not being delivered). The factors that could influence the outcome of this alternative include: visibility, sea conditions, wind conditions, and the size of the target vessel.

The second alternative that was added was the possibility that even with a locator beacon, once the UAV gets to the vessel's reported location, the vessel could not be found at that location. The 'not found' alternative was further split into two possibilities: 'continue searching for the vessel' and 'bingo fuel'. The factors that could influence the outcome of these alternatives are the same as above, with the added constraint of a time limit on the 'continue searching' possibility.

With the possibilities 'payload misses target', 'vessel not found' and 'bingo fuel' being added to the UAV (Air Vehicle) root, corresponding alternatives also had to be added to the Rescuee root, ('Receive Payload' vs. 'Payload Not Received'), and the Mission_Commander root, (the various different 'Confirm' events).

The MP code (available via download directions in Appendix F) produces eight traces at scope 1 and twelve traces at scope 2, with various combinations of the scenarios described above. Figures G.24-G.27 show four of the scope 1 scenarios. The scope 2 scenarios contain an extra search iteration.

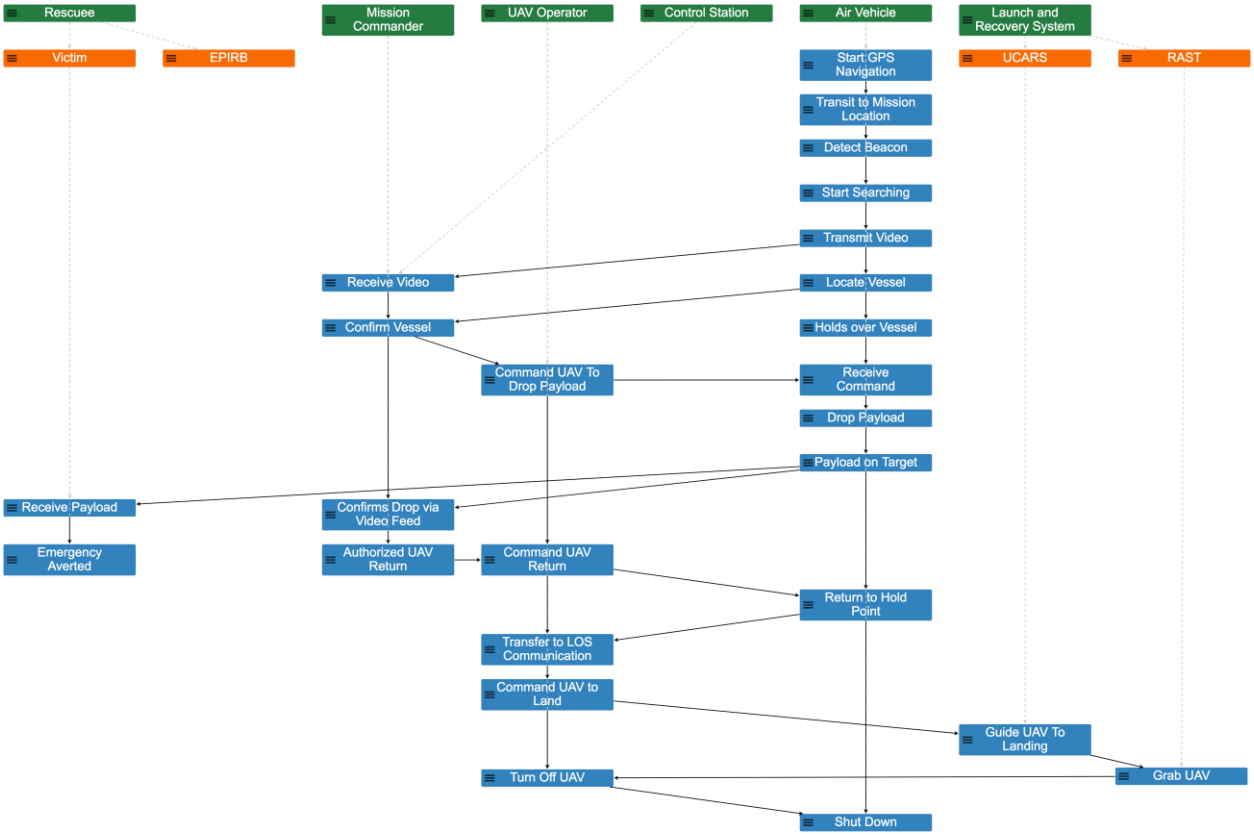


Figure G.24. Trace 1: Vessel is located and payload is on target.

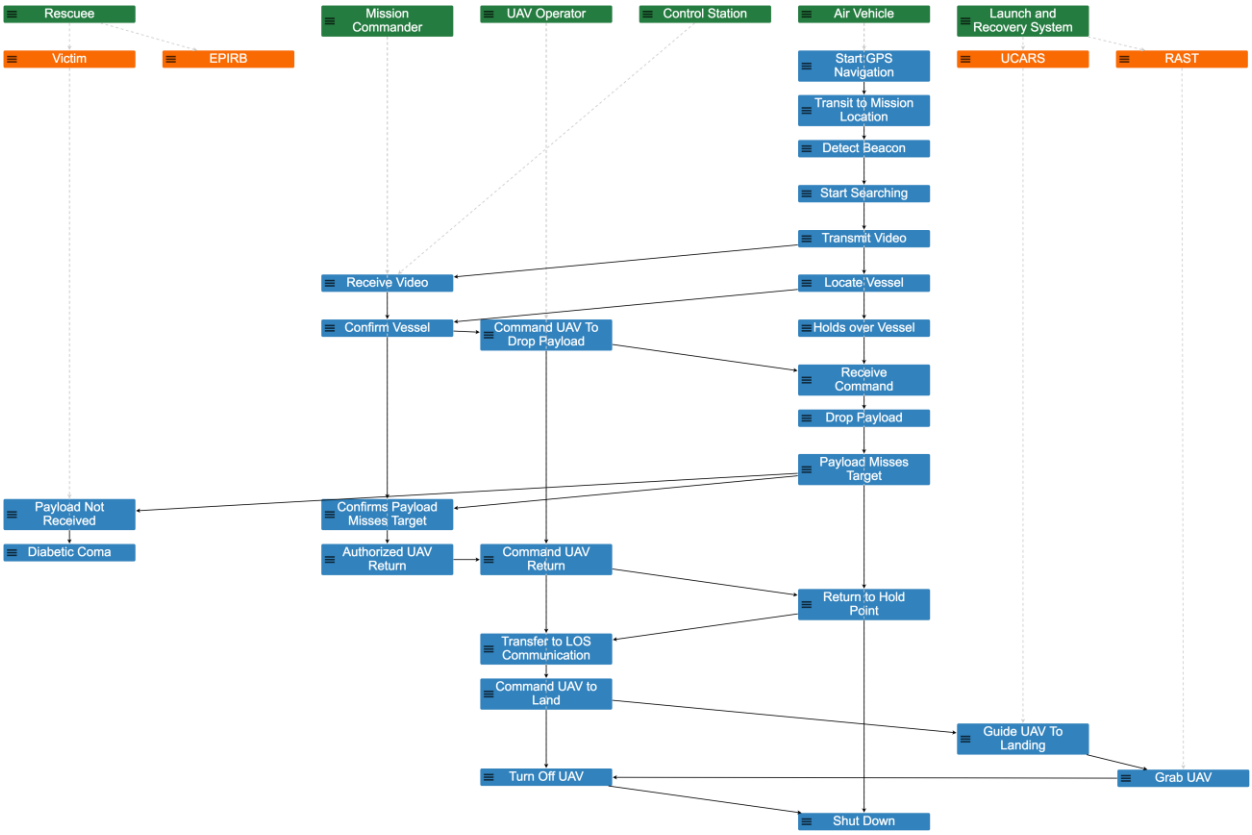


Figure G.25. Trace 3: Vessel is located and payload misses target.

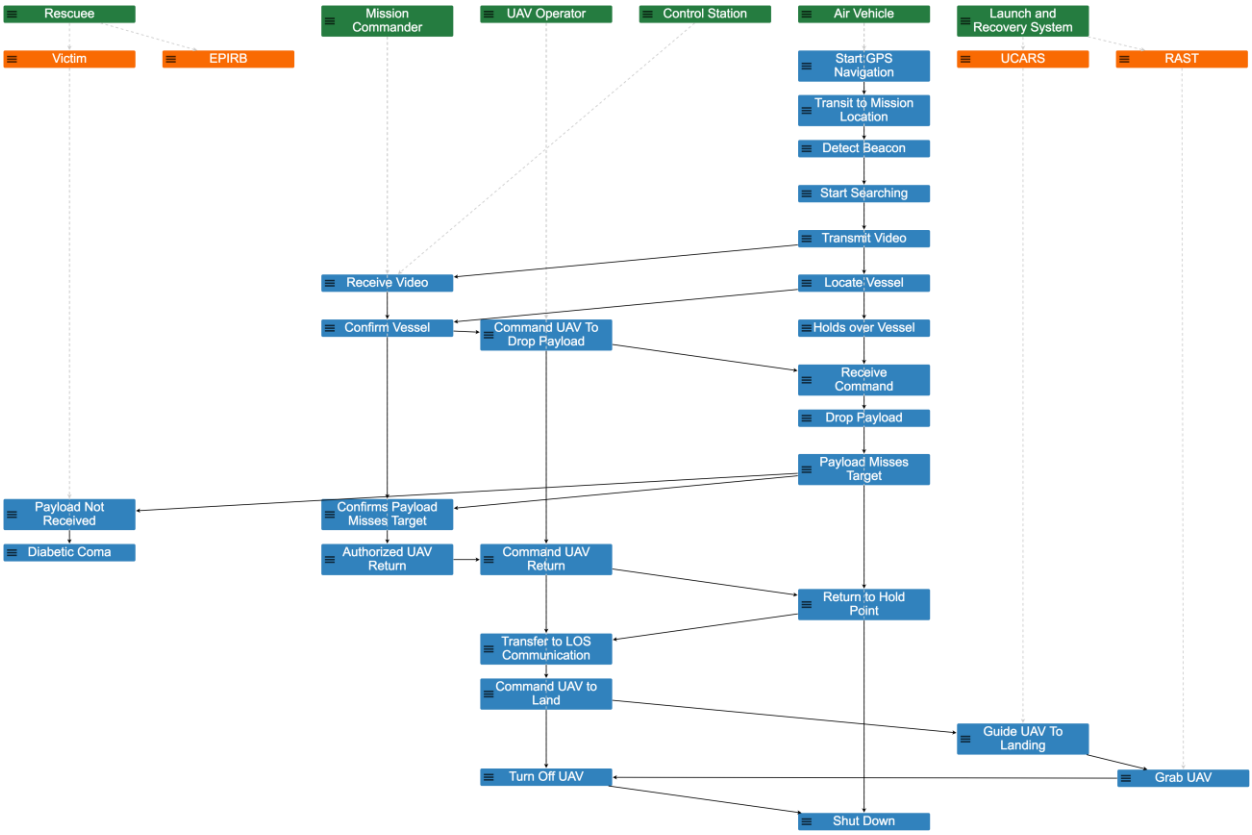


Figure G.26. Trace 7: Vessel is not located and search is stopped.

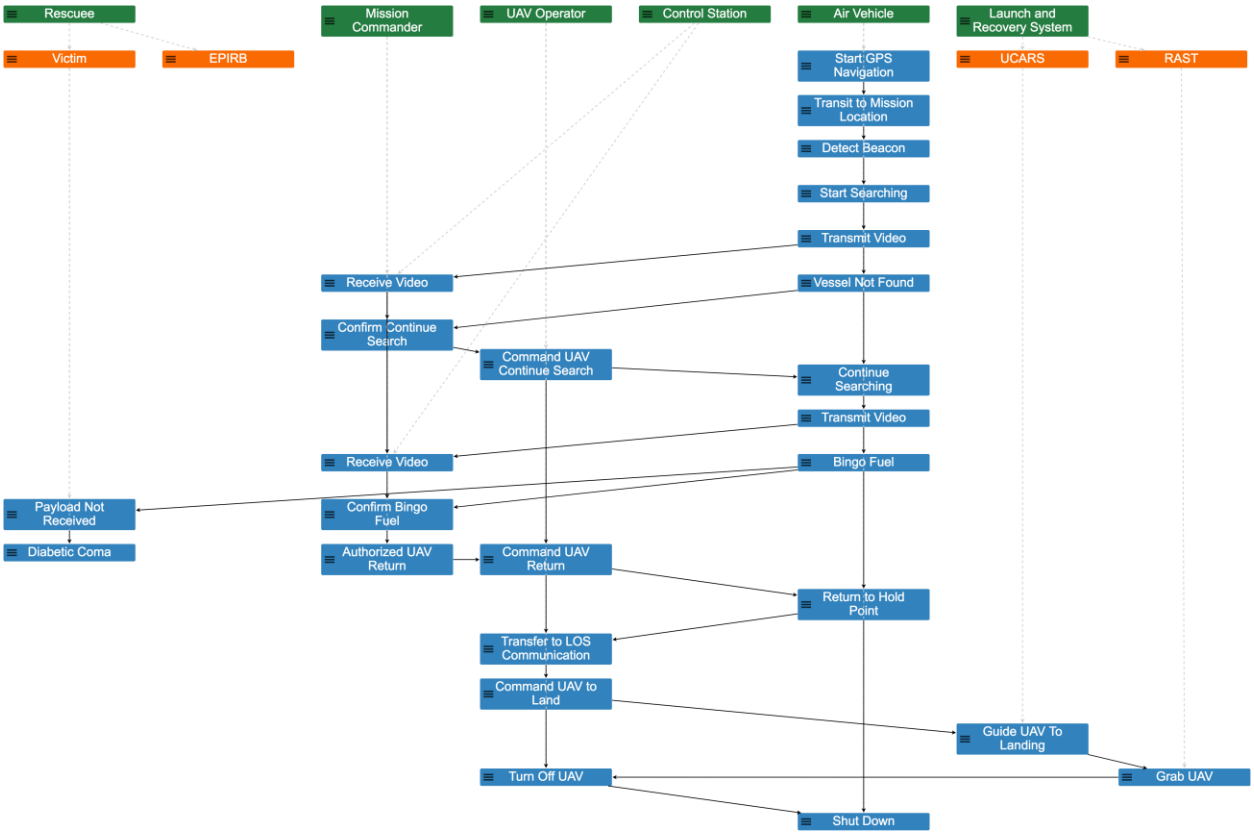


Figure G.27. Trace 8: Vessel is not located and bingo fuel results in recall of UAV.

G.5.5. Alternative Emergent Behaviors

With systems of increasing complexity, interest is increasing in the detection, classification, prediction and control of emergence in systems. *Emergence* occurs when global behaviors arise in the whole system from individual behaviors in parts of the system. While emergence in natural systems has been studied for some time, emergence in human-made systems often surprises us with unexpected and costly consequences. MP provides a new type of Petri dish for the cultivation and study of emergent behaviors in simulations of designed systems.

During the process of expanding the phase 3 model, the modeler initially came upon fewer scenarios than expected. To debug the model, the Air Vehicle root was run by itself, with no constraints or interactions with other roots. This debugging model produced six scenarios at scope 1, four of which are shown side by side in Figure G.28. The error turned out to be a verification type error but the debugging process also exposed some validation-type emergent behaviors that drew attention to the need for additional requirements in the air vehicle.

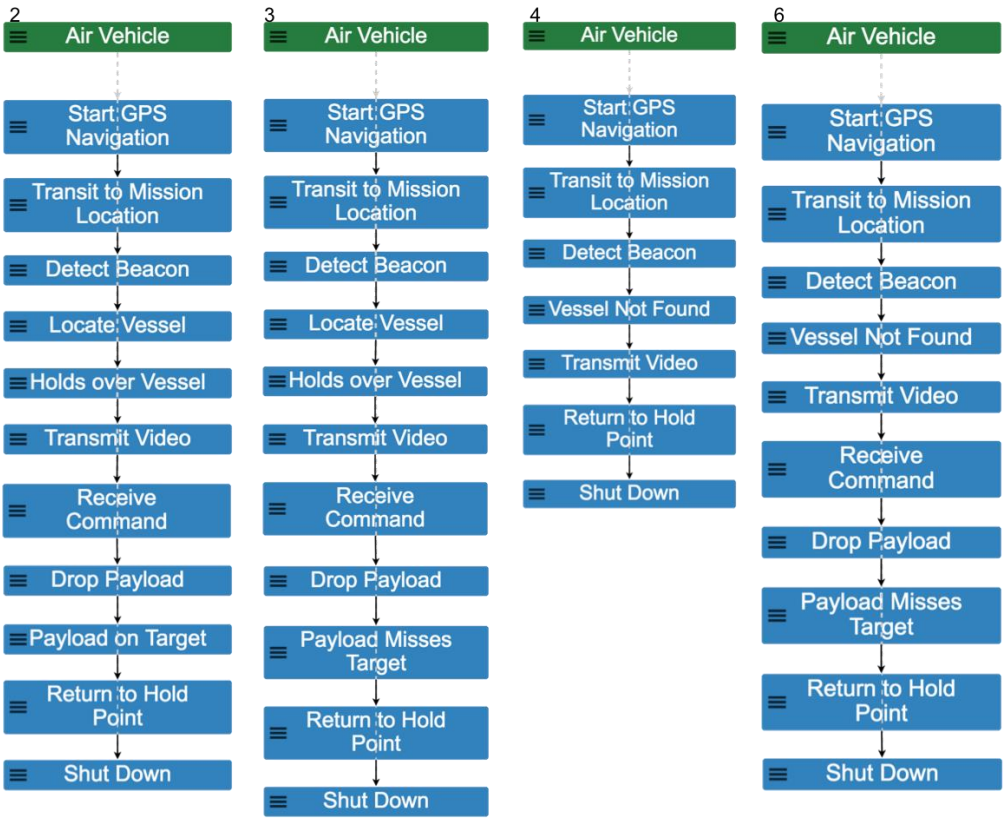


Figure G.28. Alternative emergent behaviors in the Air Vehicle. Far left: Baseline scenario; vessel located and payload on target (trace 2). Middle left: Vessel located but payload missed target (trace 3). Middle right: AV needs to return before vessel is located (trace 4). Far right: Vessel not found but AV drops payload (trace 6).

These scenarios inspired contemplation of scenarios that had not been previously considered. For example, what should happen if the payload just misses the target (trace 3)? Could the payload still be retrieved by target vessel? What requirements may help increase mission success for relatively little additional cost? What should happen if the AV has to return before locating/reaching the vessel (trace 4)? Could the payload be dropped at the air vehicle's maximum range, equipped with a means for local retrieval? And what should happen if the AV drops the payload prematurely, enroute to the vessel (trace 6)? Though unintended by the modeler, trace 6 contained an idea for handling out of range vessels or air vehicles that experience a return to base condition. Another viewpoint of scenario 6 is that it is a completely undesired scenario that fails to meet mission requirements. In this case, it is a demonstration of strong negative emergent behavior and a target for risk analysis to determine how such a scenario could come about, and how to mitigate the risk.

All of these operational "what if" questions were exposed through MP modeling of the provided baseline scenario. In summary, MP modeling of SysML behavior diagrams helped to find requirements and expose possible negative emergent behavior that may otherwise not be considered until later in the lifecycle. The model containing the original error and the corresponding debugging model are available for download following instructions provided in Appendix F.