



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2022-09

**ASSESSING INTEROPERABILITY BETWEEN
BEHAVIOR DIAGRAMS CONSTRUCTED WITH
SYSTEMS MODELING LANGUAGE (SYSML) AND
MONTEREY PHOENIX (MP)**

Hall, Joseph III; Le, Kevin; Patel, Krunal V.; Savacool,
Michael A.

Monterey, CA; Naval Postgraduate School

<http://hdl.handle.net/10945/71134>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

**SYSTEMS ENGINEERING
CAPSTONE REPORT**

**ASSESSING INTEROPERABILITY BETWEEN BEHAVIOR
DIAGRAMS CONSTRUCTED WITH SYSTEMS MODELING
LANGUAGE (SYSML) AND MONTEREY PHOENIX (MP)**

by

Joseph Hall III, Kevin Le, Krunal V. Patel,
and Michael A. Savacool

September 2022

Advisor:
Second Reader:

Kristin M. Giammarco
Scot A. Miller

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 2022	3. REPORT TYPE AND DATES COVERED Systems Engineering Capstone Report	
4. TITLE AND SUBTITLE ASSESSING INTEROPERABILITY BETWEEN BEHAVIOR DIAGRAMS CONSTRUCTED WITH SYSTEMS MODELING LANGUAGE (SYSML) AND MONTEREY PHOENIX (MP)		5. FUNDING NUMBERS	
6. AUTHOR(S) Joseph Hall III, Kevin Le, Krunal V. Patel, and Michael A. Savacool			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.		12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Systems engineers have long struggled to identify and understand system behaviors in the operational environment. System Modeling Language (SysML) is a graphical language used among systems engineers to relay details of the system's design to various stakeholders. Monterey Phoenix (MP) is a behavioral modeling approach and tool utilizing a lightweight formal method and language to generate diagrams and display expected and unexpected emergent system behaviors. Through systematic analysis of SysML and MP behavior models, this research presents recommendations for improving MP in future releases to accommodate SysML compliance. The ability to merge MP's scope complete event trace generation into a SysML compliant format would provide great insights and benefits into the DOD acquisition process. Findings from this research include several simple additions to MP diagrams that will better align them with SysML standards while preserving MP's capability to enable identification of emergent behavior early in the design process, when the risks can be addressed before system design features are ever manufactured or tested.			
14. SUBJECT TERMS Monterey Phoenix, MP, System Modeling Language, SysML, diagram, use case, behavior, modeling, language, model-based systems engineering, MBSE, modeling, simulation, magic, systems of systems, model, wreckers, sequence, state machine, activity, scope, event trace, graphical, systems engineering, firebird		15. NUMBER OF PAGES 151	16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**ASSESSING INTEROPERABILITY BETWEEN BEHAVIOR DIAGRAMS
CONSTRUCTED WITH SYSTEMS MODELING LANGUAGE (SYSML)
AND MONTEREY PHOENIX (MP)**

Joseph Hall III, Kevin Le, Krunal V. Patel, and Michael A. Savacool

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 2022**

Lead Editor: Kevin Le

Reviewed by:

Kristin M. Giammarco
Advisor

Scot A. Miller
Second Reader

Accepted by:

Oleg A. Yakimenko
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Systems engineers have long struggled to identify and understand system behaviors in the operational environment. System Modeling Language (SysML) is a graphical language used among systems engineers to relay details of the system's design to various stakeholders. Monterey Phoenix (MP) is a behavioral modeling approach and tool utilizing a lightweight formal method and language to generate diagrams and display expected and unexpected emergent system behaviors. Through systematic analysis of SysML and MP behavior models, this research presents recommendations for improving MP in future releases to accommodate SysML compliance. The ability to merge MP's scope complete event trace generation into a SysML compliant format would provide great insights and benefits into the DOD acquisition process. Findings from this research include several simple additions to MP diagrams that will better align them with SysML standards while preserving MP's capability to enable identification of emergent behavior early in the design process, when the risks can be addressed before system design features are ever manufactured or tested.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	INTRODUCTION.....	1
B.	PROBLEM OVERVIEW.....	1
C.	BACKGROUND	2
D.	PROJECT OBJECTIVES.....	6
E.	PROJECT SCOPE.....	6
F.	RESEARCH METHODOLOGY	7
G.	DOCUMENT ORGANIZATION.....	8
H.	CHAPTER SUMMARY.....	9
II.	BEHAVIOR LANGUAGE FOUNDATION	11
A.	SYSTEM MODEL LANGUAGE (SYSML)	11
1.	Use Case Diagram	12
2.	Activity Diagram.....	15
3.	Sequence Diagram	19
4.	State Machine Diagram.....	23
B.	MONTEREY PHOENIX (MP)	25
1.	Language, Approach, and Tool	28
2.	Extraction of Behavior Diagrams.....	30
C.	CHAPTER SUMMARY.....	33
III.	CASE STUDY MODELS DEVELOPMENT.....	35
A.	AUTO GCAS MODEL.....	35
B.	FIREARM SAFETY MODEL.....	39
C.	CHAPTER SUMMARY.....	43
IV.	MODEL ANALYSIS	45
A.	SIMILARITIES AND DIFFERENCES	45
1.	MP Event Trace vs. SysML Sequence Diagram	45
2.	MP Activity Diagram vs. SysML Activity Diagram	54
3.	MP State Machine vs. SysML State Machine Diagram	66
B.	OBSERVATIONS ON OPTIMIZING MODEL STRUCTURE.....	70
1.	Compact Event Trace Structure Method	71
2.	MP State Machine Diagram Comparisons	72
C.	CHAPTER SUMMARY.....	79
V.	CONCLUSIONS AND RECOMMENDATIONS.....	81

A.	CONCLUSION	81
1.	MP Event Trace vs. SysML Sequence Diagrams.....	82
2.	MP vs. SysML Activity Diagram.....	84
3.	MP vs. SysML State Machine Diagram.....	86
4.	Comparison between MP-Firebird and SysML MSOSA Tools	87
5.	Monterey Phoenix State Machine Observations.....	88
B.	RECOMMENDATIONS.....	90
1.	MP Event Trace Diagram	90
2.	MP Activity Diagram.....	92
3.	MP State Machine Diagram.....	94
4.	Recommendation for MP Use Case Diagram Development	96
5.	Enhancement Recommendation for SysML Behavior Diagrams.....	99
C.	FUTURE WORK.....	100
1.	Event Trace Diagram Table Tool.....	101
2.	Use Case Diagrams in Monterey Phoenix.....	101
3.	Monterey Phoenix User Interface Capability.....	102
APPENDIX A. MP MODEL SOURCE CODE		105
1.	AutoGCAS_Normal_Condition_V1_4.....	105
2.	AutoGCAS_Impact_Condition_V1_4.....	108
3.	AutoGCAS_State_Machine_Diagram_V1_3	111
4.	Firearm_Safety_Top_Level_V1_4.....	115
5.	Firearm_Safety_Operational_V1_4	119
APPENDIX B. FIREARM SAFETY MODEL ACTIVITY DIAGRAMS.....		123
LIST OF REFERENCES.....		125
INITIAL DISTRIBUTION LIST		127

LIST OF FIGURES

Figure 1.	ATM system MP-generated behavior diagrams. Adapted from Auguston (n.d.).	4
Figure 2.	SysML activity diagram for the ATM system. Adapted from Auguston (n.d.).	5
Figure 3.	Capstone project system engineering plan.	7
Figure 4.	SysML diagram types. Adapted from OMG SysML (n.d.).	12
Figure 5.	Gas Pump use case diagram. Adapted from Zdanis and Cloutier (2007).	14
Figure 6.	Fuel Pump activity diagram. Adapted from Zdanis and Cloutier (2007).	18
Figure 7.	Fuel Pump sequence diagram. Adapted from Zdanis and Cloutier (2007).	22
Figure 8.	Fuel Pump state machine diagram. Adapted from Zdanis and Cloutier (2007).	25
Figure 9.	Auguston’s concept of an abstract event. Source: Giammarco (2022).	26
Figure 10.	Event trace visualizations of events and relationships in MP. Source: Giammarco (2022).	27
Figure 11.	Example grammar rules with corresponding event traces. Source: Giammarco (2022).	29
Figure 12.	Event trace examples. Adapted from Auguston (n.d.).	31
Figure 13.	Example activity diagrams. Adapted from Auguston (n.d.).	32
Figure 14.	Example of an MP state machine diagram. Adapted from Auguston (n.d.).	33
Figure 15.	What is Auto GCAS. Source: Lockheed Martin (n.d.).	36
Figure 16.	SysML sequence diagram, illustrated using the Auto GCAS model.	38
Figure 17.	Firearm Safety state machine diagram.	42

Figure 18.	MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with “Built-in Test No Go”	46
Figure 19.	MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with “Auto GCAS Enabled”	47
Figure 20.	MP event trace (left) vs. SysML sequence diagram (right), illustrated using the Auto GCAS impact flight condition when the pilot takes corrective action.....	48
Figure 21.	MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS impact flight condition with fly-up executed	49
Figure 22.	Differences in parallel events notational concepts between MP event trace (left) and SysML sequence (right) diagrams, illustrating using the Auto GCAS normal condition with Auto GCAS enabled	52
Figure 23.	Differences in types of arrow notational concept between MP event trace (left) and SysML sequence (right) diagrams, illustrating using the Auto GCAS normal condition with Auto GCAS enabled	53
Figure 24.	SysML activity diagram, illustrated using the Auto GCAS normal flight condition model.....	55
Figure 25.	SysML activity diagram, illustrated using the Auto GCAS impact flight conditions model	57
Figure 26.	Notational concepts in common between SysML (left) and MP (right), illustrated using the Auto GCAS normal flight conditions model.....	58
Figure 27.	Differences in notational concepts between SysML (left) and MP (right), illustrated using the Auto GCAS normal flight conditions model.....	60
Figure 28.	Differences between SysML merge node and decision node on SysML activity diagram.....	61
Figure 29.	Differences in fork horizontal and join horizontal notational concepts between SysML (left) and MP (right) activity diagram.....	61
Figure 30.	Differences in send signal action and accept event action notational concepts between SysML (left) and MP (right) activity diagram, illustrated using the Auto GCAS with impact condition model	62

Figure 31.	Differences in precedence relationship notational concepts between SysML (left) and MP (right) activity diagram, illustrated using the Auto GCAS normal flight conditions model	63
Figure 32.	Differences in guard condition notational concepts between SysML (left) and MP (right) activity diagram.....	64
Figure 33.	Differences in conditional loop notational concepts between SysML (left) and MP (right) activity diagram.....	65
Figure 34.	Proposed simplified conditional loop between MP original (left) and simplified (right) activity diagram, illustrated using the Auto GCAS impact flight conditions model	66
Figure 35.	SysML state machine diagram, illustrated using the Auto GCAS model.....	68
Figure 36.	MP state machine diagram, illustrated using the Auto GCAS model.....	68
Figure 37.	Differences in notational concepts between SysML (left) vs. MP (right) state machine diagram	69
Figure 38.	SysML state machine diagram with composite state notational concept, illustrated using the Auto GCAS model.....	70
Figure 39.	MP event trace diagram number 2 run at scope 1, illustrated using the Auto GCAS state machine model	71
Figure 40.	Monterey Phoenix event trace diagram number 1 run at scope 1, illustrated using the Firearm Safety Top-level model.....	72
Figure 41.	Firearm Safety Top-level model, illustrated using Microsoft Visio	73
Figure 42.	Drill down of the Firearm Safety operational state diagram from the Firearm Safety Top-level model, illustrated using Microsoft Visio	74
Figure 43.	Monterey Phoenix state machine diagram, illustrated using the Firearm Safety Top-level model	76
Figure 44.	MP event trace diagram number 3 run at scope 1, illustrated using the Firearm Safety Top-level model	77
Figure 45.	Monterey Phoenix state machine diagram, illustrated using the Firearm Safety Operational model.....	78

Figure 46.	Recommendation for translating MP event trace (left) into SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with Auto GCAS enabled	92
Figure 47.	Comparison of Auto GCAS state machine models between MP Gryphon (left) and MP Firebird (right).....	96
Figure 48.	Firearm Safety model use case diagram concept, illustrated using Microsoft Visio	97
Figure 49.	Firearm Safety model use case diagram concept, illustrated using MSOSA.....	99
Figure 50.	Original SysML sequence diagram (left) vs. modified SysML sequence diagram (right), illustrated using the Auto GCAS normal condition model	100
Figure 51.	Firearm Safety Top-level model activity diagram, illustrated by Monterey Phoenix.....	123
Figure 52.	Firearm Safety Operational model activity diagram, illustrated by Monterey Phoenix.....	124

LIST OF TABLES

Table 1.	Use case diagram notations. Adapted from MSOSA (n.d.).....	15
Table 2.	Activity diagram notations. Adapted from MSOSA (n.d.).....	19
Table 3.	Sequence diagram notations. Adapted from MSOSA (n.d.).....	23
Table 4.	State machine diagram notations. Adapted from MSOSA (n.d.).	25
Table 5.	MP main state machine event traces.....	75
Table 6.	MP operational state machine event traces.....	78
Table 7.	Differences between MP event trace and SysML sequence diagrams	82
Table 8.	Similarities between MP and SysML activity diagrams.....	84
Table 9.	Differences between MP and SysML activity diagrams.....	85
Table 10.	Similarities between MP and SysML state machine diagrams.....	86
Table 11.	Differences between MP and SysML state machine diagrams.....	86
Table 12.	Differences between MP-Firebird and SysML MSOSA tools	88

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ATM	Automated Teller Machine
CPI	Conversational Programming Interface
DA	Department of the Army
DOD	Department of Defense
DOS	Disk Operating System
GCAS	Ground Collision Avoidance System
GUI	Graphical User Interface
ID	Identity
MSOSA	Magic Systems of Systems Architect
MP	Monterey Phoenix
NASA	National Aeronautics and Space Administration
NPS	Naval Postgraduate School
OMG	Object Management Group
SE	Systems Engineering
SoS	System of Systems
SysML	Systems Modeling Language
UML	Unified Modeling Language
U.S.	United States

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Systems engineers (SE) have long struggled to identify and understand system behaviors in the operational environment. A primary tool set in a SE's toolbox has been the use of System Modeling Language (SysML) behavior diagrams. These include activity, sequence, state machine, and use case diagrams. They allow the SE to draw out expected interactions of the system's subsystems (or components), the operating environment (including any external systems), and the operator of the system.

SysML is a graphical language that is used among systems engineering practitioners to relay details of the system's design to various stakeholders. This language has a vocabulary and grammatical structure for all its diagram types. It requires the modeler to have an intimate understanding of how the system is expected to function, how it will be used, the operational environment, and how the operator is expected to engage the system, to map the behaviors correctly. It is generally accepted that the behavior diagrams are only as good as the SE's understanding of the system and their inherent biases of how the system is going to behave.

Monterey Phoenix (MP) is a behavioral modeling approach and tool that utilizes a lightweight formal method and language composed of a combination of precise logical and mathematical notations to generate diagrams and display system behaviors (NPS Wiki n.d.) Modeling system behaviors in MP requires the modeler to define the relationships between events contained within the system, its environment, and the end user. These relationships are then automatically shown in a set of graphical models called event traces. MP generates all possible relationship combinations in separate event traces, which will help the SE to identify emergent behavior within the operational environment. MP is also capable of generating activity and state machine diagrams of the system being modeled. However, the graphical outputs from the present tools are not SysML compliant.

The primary objective of this capstone project is to advance the systems engineering community's understanding of the SysML and MP behavior models by identifying the overlaps and gaps between these two graphical languages that facilitate or

impede an automated conversion between one notation and the other. MP's ability to generate scope complete event trace diagrams (which are like the SysML sequence diagrams) based on modeler-defined relationships is a capability that is presently missing from tools that produce SysML behavior diagrams.

The approach taken to bridge this gap in behavior modeling languages involved developing SysML and MP behavior models for two case study systems; the Automatic Ground Collision Avoidance System (Auto GCAS) and the Firearm Safety Model (FSM). The Auto GCAS is a safety feature in fighter aircraft intended to prevent ground collisions. The FSM models the interactions between a shooter and a firearm. Each was chosen because of its applicability to modeling system behaviors and would allow comparisons of both modeling languages on a one-to-one comparison among the different types of behavior diagrams. Activity, sequence, and state machine diagrams were generated for both the Auto GCAS and the FSM in a SysML-compliant format. The equivalent diagrams generated from the MP model were then compared against the SysML diagrams. While the MP graphical language lacks the visual vocabulary of SysML, the information contained in the MP model is capable of being formatted for SysML compliance.

The team systematically identified the notational differences and similarities between the two graphical languages and provides recommendations for improving MP in future releases to accommodate SysML-compliant diagram generation. Comparing the MP event traces and SysML sequence diagrams, a total of four differences are identified between them and four recommendations for improvement of the MP event trace generator are provided. Next, the MP and SysML activity diagrams are compared, revealing a total of five differences and four similarities, with five recommendations provided for improvement. Finally, the comparison between the state machine diagrams from MP and SysML revealed three strong similarities in conceptual employment, three minor differences in graphical notations, and six recommendations provided for improving MP's state machine generation. Throughout the research activities, the Model Wreckers also identified three recommendations for future work to improve the capabilities and overall user experience of modeling system behaviors with MP.

The ability to merge MP's scope complete event trace generation into a SysML compliant format enables great insights and benefits for the DOD acquisition process. By being able to identify potentially emergent behavior much earlier in the design process, the risks can be addressed before system design features are ever manufactured or tested. The cost of correcting for, or mitigating, an emergent behavior is significantly higher after prototyping and testing a physical design solution. This research advances the SE community one step closer towards leveraging the emergent behaviors generated from MP in familiar SysML notation.

References

NPS Wiki. n.d. "About - Monterey Phoenix." Accessed March 1, 2022.
<https://wiki.nps.edu/display/MP/About>

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The Model Wreckers would like to express our deepest gratitude to all the Naval Postgraduate School's Systems Engineering faculty and staff who have made this research/thesis possible. The professors guided our endeavor for lifelong learning with guidance and patience in preparation for our execution of this capstone research project.

Many thanks to our advisor, Dr. Giammarco, and second reader, Mr. Scot Miller, for all of your guidance and patience through the stresses of completing this research. It would not have been possible without you both. Thanks also go out to the Graduate Writing Center staff and coaches for the valuable feedback, suggestions, and assistance to meet the NPS standard for thesis publication.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. INTRODUCTION

This capstone report describes the purpose, methodology, analysis, and findings when comparing SysML v1.6 and Monterey Phoenix 4.0 behavior diagrams by the Model Wreckers team. The analysis in this report is not only used to improve the Monterey Phoenix software to better serve the SysML community, but also to show the SysML community where the next generation of SysML (SysML 2.0) can be improved upon.

This report contains an introduction and background chapter, a behavior language review chapter, a case study model development chapter, a model analysis chapter, and a conclusion chapter. In the introduction and background chapter, the problem overview, background, project objectives, scope, and research methodology are discussed. The behavior language review chapter discusses relevant background information about Monterey Phoenix 4.0 and SysML 1.6. The case studies chapter introduces both models used to perform diagram comparisons, and the following model analysis chapter discusses the analysis results. The conclusion and recommendation chapter summarizes the analysis findings and includes information on how MP can be improved in reference to the SysML community.

B. PROBLEM OVERVIEW

Monterey Phoenix (MP) is a behavioral modeling language, approach, and tool that helps its users unveil errors in system interactions by visually displaying all possible permutations of these interactions modeled in the context of the operational environment with and without known constraints.

MP has supported its users in understanding system behaviors, realizing the impacts of assumptions, and discovering unwanted behaviors of the system being modeled early in the design phase (NPS Wiki, n.d.) In “Verification and Validation (V&V) of System Behavior Specifications,” Giammarco et al. (2018) discuss multiple examples of unexpected system behaviors that may have avoided discovery without MP’s exhaustive trace generation capability. These unexpected system behaviors are usually permitted by

system design, but violated the mission requirements in certain operational conditions. As an example, in the DOD, a flaw in the software design relating to the international date line caused the U.S Air Force F-22 Raptors to cancel their first deployment mission. The international date line is an imaginary borderline in the Pacific Ocean that divides the other side by two consecutive calendar days. The F-22s were developed and tested in the United States using the U.S. time zone. As a result, the international date line crossing test case was not considered during the F-22 system development. When the F-22 crossed the boundary during a mission from Hawaii to Japan, the time in the aircraft's mission computers reversed in such a manner that caused the F-22's mission computers to malfunction and shut down multiple aircraft systems. Luckily, the F-22s were able to return to base with support from the tankers (Nambi 2020). This unexpected system behavior proves that other catastrophic and unwanted emergent behaviors may occur in certain circumstances without considering all possible and impossible scenarios that may occur during operation.

With v4.0, MP supports generating scenario variants as event traces and extracting global architecture views to include activity diagrams, sequence diagrams, state diagrams, and other views. However, because MP diagrams have some differences from the widely used Systems Modeling Language (SysML), systems engineers who use SysML but also want to leverage MP's scope-complete trace generation capability may encounter some notational discrepancies between MP and SysML, impeding both user communities from efficiently leveraging the work done by the other. As a result, there is a need to identify the gaps and overlaps between the current MP v4.0 and SysML v1.6 behavior diagrams while characterizing their differences for MP's behavior diagrams to have consistency with SysML standards. This consistency should make MP's exhaustive trace generation capability more easily used by the SysML community.

C. BACKGROUND

SysML is a graphical language that is used among systems engineering practitioners to relay details of the system's design to various stakeholders. This language has a vocabulary and grammatical structure for all its diagram types. The shape of every

object in the diagram, the connecting lines, arrows, and the relative position of the connections to various objects all relay very specific details about the system to those fluent in the language (Delligatti 2014). Since SysML v1.0a was officially accepted as an extension of Unified Modeling Language (UML) in May 2006, systems engineering (SE) practitioners have employed it to help them understand the nuances and details of system behavior using information relayed via SysML behavior diagrams. Non-technical system stakeholders often use simpler notations to capture and convey their understanding of the nuances with how the system is intended to behave and interact with the intended users of the system.

MP is a behavioral modeling approach and tool that utilizes a lightweight formal method and language composed of a combination of precise logical and mathematical notations to generate diagrams and display system behaviors. Unlike other modeling tools, MP generates a scope-complete set of behavior examples from a single formal specification that uses a simple event grammar. Currently, MP can generate behavior diagrams in the forms of an event trace, analogous to a SysML sequence diagram; an activity diagram for each individual actor represented in the MP model, and state machine diagrams that present the transitions and states for each actor within the MP model. Figure 1 illustrates the different behavior diagrams generated by MP for an ATM system. The event trace and activity diagrams show the order flow of the following events: the customer inserts their card into the ATM system; the ATM system reads the card and validates the ID with the database; once the validation is successful, the ATM system allows the customer to request a withdrawal. The state machine diagram illustrates the same flow based on the states and transitions of the ATM system.

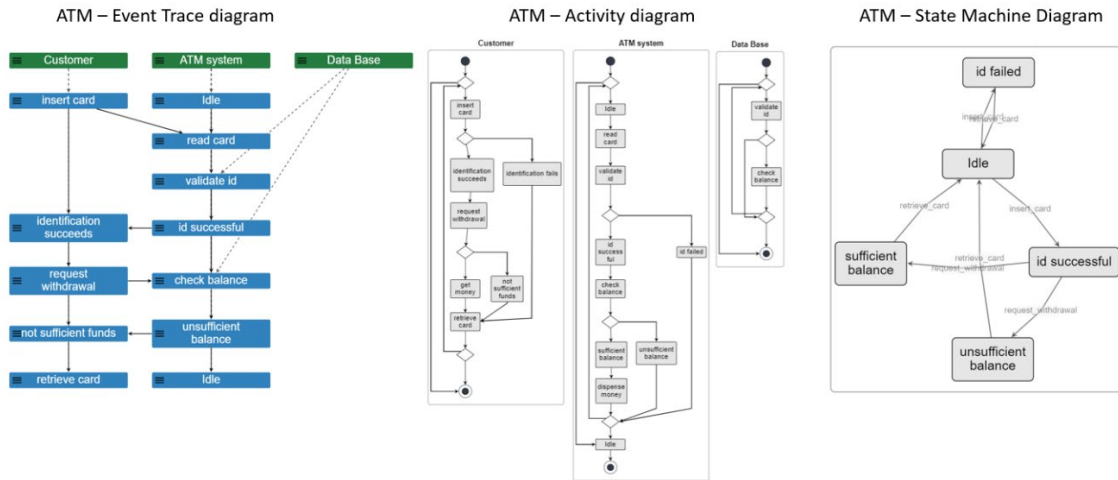


Figure 1. ATM system MP-generated behavior diagrams. Adapted from Auguston (n.d.).

MP’s event grammar utilizes two object types to show behaviors: events and relations. When the user finishes an initial description of behavior logic for each separate system, the user can add constraints to coordinate the related behaviors of the overall modeled system of systems. The relaxation and restriction of constraints on events and relations allow the MP coding language to express system behaviors on a high level and to probe for undesired emergent behavior (Giammarco 2022). Behaviors of different systems are captured separately from each other and assumed to be independent unless a constraint is separately added to coordinate the behaviors in the separate systems.

By contrast, SysML v1.6 supports the construction of graphical views containing identified logic flows through activities in different systems using a swim lanes-style diagram. Figure 2 is an example of an activity diagram in SysML v1.6 format, created using Magic Systems of Systems Architect (MSOSA) software. This activity diagram shows all the planned logic routes in the customer’s interaction with the ATM system from start to finish. However, when creating integrated graphical views, it is easy for the system designers to overlook unwanted but operationally plausible paths through the activity being depicted (e.g., the user making a withdrawal, despite the system denying the user card access).

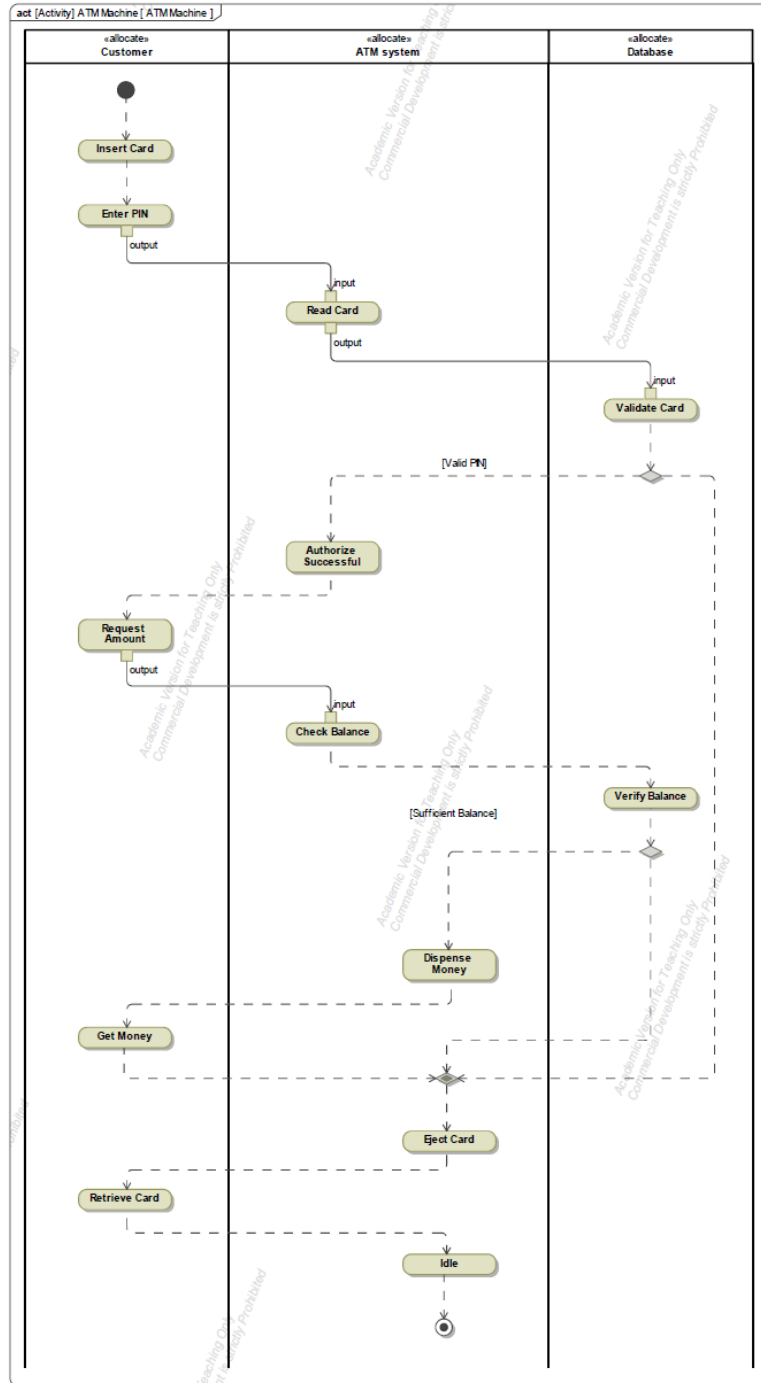


Figure 2. SysML activity diagram for the ATM system. Adapted from Auguston (n.d.).

As of this writing, there is no formal interface or relationship map between MP and SysML diagrams. However, a cursory examination showed an opportunity to document the

similarities and differences between MP v4.0 and SysML v1.6 diagrams. This lack of formal interface is the motivation for our capstone team, the “Model Wreckers,” to identify and characterize the overlaps and gaps that need to be rectified, to support MP’s interoperability with the SysML diagram standards.

D. PROJECT OBJECTIVES

The primary objective of this capstone project is to advance the systems engineering community’s understanding of SysML and MP behavior models by identifying the overlaps and gaps between these two notations that facilitate or impede an automated conversion between one notation and the other. Specifically, this project seeks answers to the following questions:

1. What are the differences that impede the consistent interpretation of behaviors expressed in MP v4.0 and SysML v1.6 behavior diagrams? This question specifically pertains to:
 - Sequence diagrams
 - Activity diagrams
 - State machine diagrams
2. What changes does MP v4.0 require in order for MP to produce diagrams consistent with SysML v1.6 standards?
3. What changes, if any, does SysML v1.6 require in order to support MP behavior concepts?

Finally, the team provides feedback and suggestions for refining and improving the MP users’ experience.

E. PROJECT SCOPE

For this project, the Model Wreckers team models several systems at high levels using both MP v4.0 and SysML v1.6 modeling tools. From the operational concepts for the systems of interest, one group models and generate different SysML v1.6 behavior diagram views, including the activity, sequence, state machine, and use case using the

MSOSA software. Concurrently, the other group models and generate the sequence, activity, and state machine diagrams utilizing MP-Firebird.

Since MP v4.0 does not currently support use case diagrams, the team does not use MP to automate the generation of the use case diagram for the sake of this project. Instead, the team uses SysML v1.6 modeling tools to create use case diagrams, which are then used to provide recommendations for future MP support of use case diagrams.

F. RESEARCH METHODOLOGY

The Model Wreckers tailored the System Engineering process and executed the project through three main phases, as illustrated in Figure 3. In Phase 1, our team focused on understanding the background, problem motivation, and the benefits of the study. Then, by identifying stakeholders’ needs, conducting literature reviews, and drafting the operational scenarios for our case study models, the team developed the capstone project proposal. In Phase 2, the team studied Monterey Phoenix, Magic SoS Architect, and SysML. Next, we executed the main part of the project, which included building system models utilizing two different model-based systems engineering tools, generating, and creating behavior diagrams from those tools. Finally, in Phase 3, the team recorded our findings, and provided feedback and recommendations for future research on improvements to both MP and SysML, and to enable MP diagram generation consistent with SysML v1.6 standards.

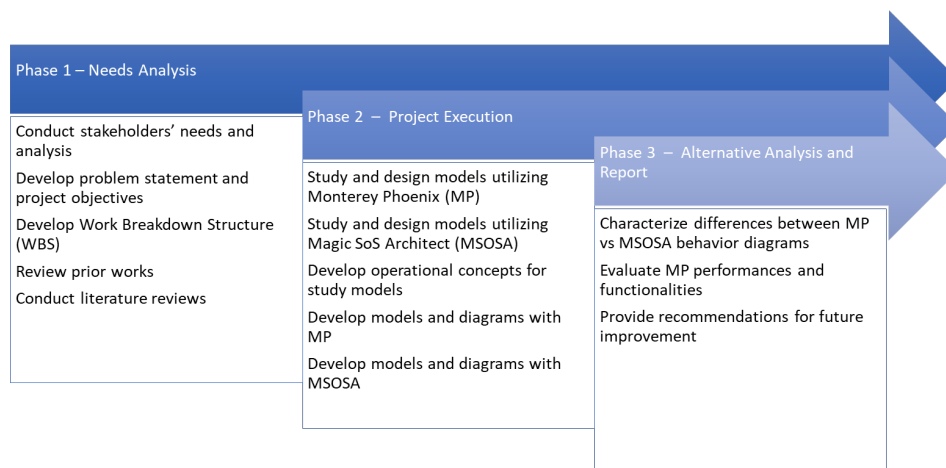


Figure 3. Capstone project system engineering plan

The team utilized the following Model-based System Engineering and Project Management tools and software:

- Model-based System Engineering
 - Magic Systems of Systems Architect, by 3DS
 - MSOSA provides various modeling tools to work with different types of diagrams. The team utilizes MSOSA to develop models and SysML diagrams during the capstone project.
 - Monterey Phoenix, developed by the U.S. Navy
 - MP is a language, approach, and open-source tool for modeling and simulating behaviors of processes and systems. The team utilizes MP to develop models and generate diagrams currently supported by MP.
- Project Management Tools
 - Microsoft Teams—Team Communication
 - Microsoft Project—Develop and Manage Project Schedule
 - Microsoft Office 365 Suite—Calculation, Reporting, and Presenting
 - Microsoft OneDrive—File Sharing

G. DOCUMENT ORGANIZATION

The Capstone report is composed of five chapters, which are Introduction, Behavior Language Foundation, Case Study Models Development, Model Analysis, and Conclusion and Recommendation chapters.

Chapter I—Introduction, introduces the problem overview, project background, project objectives, scope, and research methodology. Next, Chapter II—Behavior Language Foundation, establishes a baseline understanding of SysML, Monterey Phoenix, and the different behavior diagrams that are presented in this project including the use case, activity, sequence, and state machine diagrams.

In Chapter III—Case Study Models Development, the team discusses the two case study models: 1) Auto Ground Collision Avoidance System and 2) Firearm Safety Model. From these two models, the team develops various operational scenarios to generate

different behavior diagrams for the study. With the behavior diagrams developed, Chapter IV—Model Analysis, discusses the similarities and differences between MP and SysML behavior diagrams. In the same chapter, the team compares the similarities and differences between MP event trace vs. SysML sequence diagram, MP vs. SysML activity diagram, and MP vs. SysML state machine diagram. Especially with the Firearm Safety model, we provide observations on optimizing the model and a detailed comparison between MP vs. SysML state machine diagram.

Chapter V—Conclusions and Recommendations discusses the findings which include similarities and differences between MP and SysML behavior diagrams, the feedback from using MP, and the recommendation for future research on improvements to both MP and SysML.

H. CHAPTER SUMMARY

The introduction and background chapter presented the introduction, problem, overview, background, project objectives, project scope, research methodology, document organization, and chapter summary.

The next chapter introduces modeling concepts and applicable syntax for SysML and Monterey Phoenix. focusing on the behavior diagrams used for our case study system models: sequence diagrams, activity diagrams, and state machine diagrams.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BEHAVIOR LANGUAGE FOUNDATION

This chapter presents the baseline information needed to understand the research being presented. First, a brief background on the Systems Modeling Language (SysML) will be presented. Second, the appropriate syntax of the graphical language regarding the four behavior models of interest will be presented. These system behavior models of interest are use case diagrams, activity diagrams, sequence diagrams, and state machine diagrams.

Following the SysML information will be a brief background on Monterey Phoenix (MP). The behavior language foundation will include a section explaining the MP language and approach required to build a system's behavior model in MP. Finally, a discussion of the extracted behavior diagrams from the MP model with graphic examples will conclude this section.

A. SYSTEM MODEL LANGUAGE (SYSML)

According to Delligatti (2014), SysML is one of the graphical modeling languages that extends from Unified Modeling Language (UML) and Unified Modeling Language 2 (UML 2). The Object Management Group (OMG) publishes a version 1.6 standards specification that defines SysML syntax and vocabulary using graphical notations with predetermined meanings. SysML language is used by engineers to communicate their ideas about their system designs during the development of system simulations (OMG SysML n.d.).

Figure 4 shows the nine types of SysML diagrams used to represent different aspects of system design when building models in a graphical representation. Our team used just four of these diagram types for behavior: use case diagram, activity diagram, sequence diagram, and state machine diagram (shown in the green boxes of Figure 4). Each of these diagram types is discussed in this section using a fuel pump model (Zdanis and Cloutier 2007). This model introduces each diagram, explains its process, and highlights the elements of its syntax, employed to communicate the behavioral aspects of the system using SysML notation.

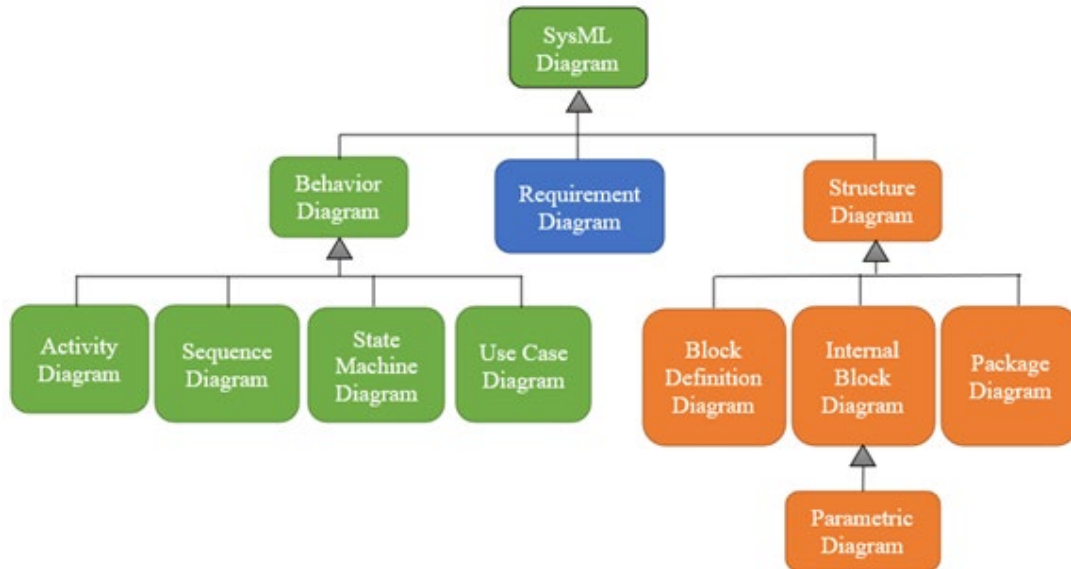


Figure 4. SysML diagram types. Adapted from OMG SysML (n.d.).

1. Use Case Diagram

The use case diagram is a behavior type of diagram. The use case diagram is employed to model complex systems, system of systems, and the traceability of systems requirements. The purpose of a use case diagram is to deliver a top-level view of system requirements and convey those top-level requirements in a simple model language. The use case diagram's top-level requirement allows all engineers, customers, stakeholders, architects, and project managers to comprehend the information. The data in a use case diagram communicates functional requirements (use cases) and external users (actors). These functional requirements are captured within a system boundary (subject box). A use case characterizes a stakeholder's high-level functional requirements to accomplish their goals, allows different flows of system design scenarios, and builds multi-level functionality within its framework. An actor can serve as an external system user, an organization, or a human. The subject box shows the system boundary within the use case diagram.

The elements available in use case diagrams (Delligatti 2014) are as follows:

- Actor—Actors are people, users, or any systems that interact with the system.

- Include—An include relationship connects one use case to another use case to perform a task.
- Extend—An extend relationship provides alternative options relative to a specific case.
- Generalization—A generalization relationship is used between two actors/ two use cases. (Note: This definition is not illustrated in Figure 5.)
- Association—An association relationship represents communication between actors and use cases.

Figure 5 shows an example of the use case diagram for a gas pump system, which is surrounded by the subject box (rectangular box). The subject box includes a “Get Fuel” (oval) use case, use case relationships (arrows), and a subset (oval) of use cases. The actors (stick figures) are presented outside of the subject box. The actors are “Operator,” “Financial Institution,” “Clerk,” and “Maintainer.” These actors are interacting within the system boundary surrounded by the subject box and are connected via association relationship to the use case and a subset of the use cases. The main “Get Fuel” use case has a subset of three use cases connected via an “include” relationship. The subset of use cases is “Obtain Authorization,” “Pump Fuel,” and “Pay for Fuel.” The “Initialize Pump” is connected via an “extend” relationship that also links to a use case. This relationship contains optional additional activity within the main use case “Get Fuel.” The “Maintain” use case is separate from the main use case and only interacts with “Maintainer.”

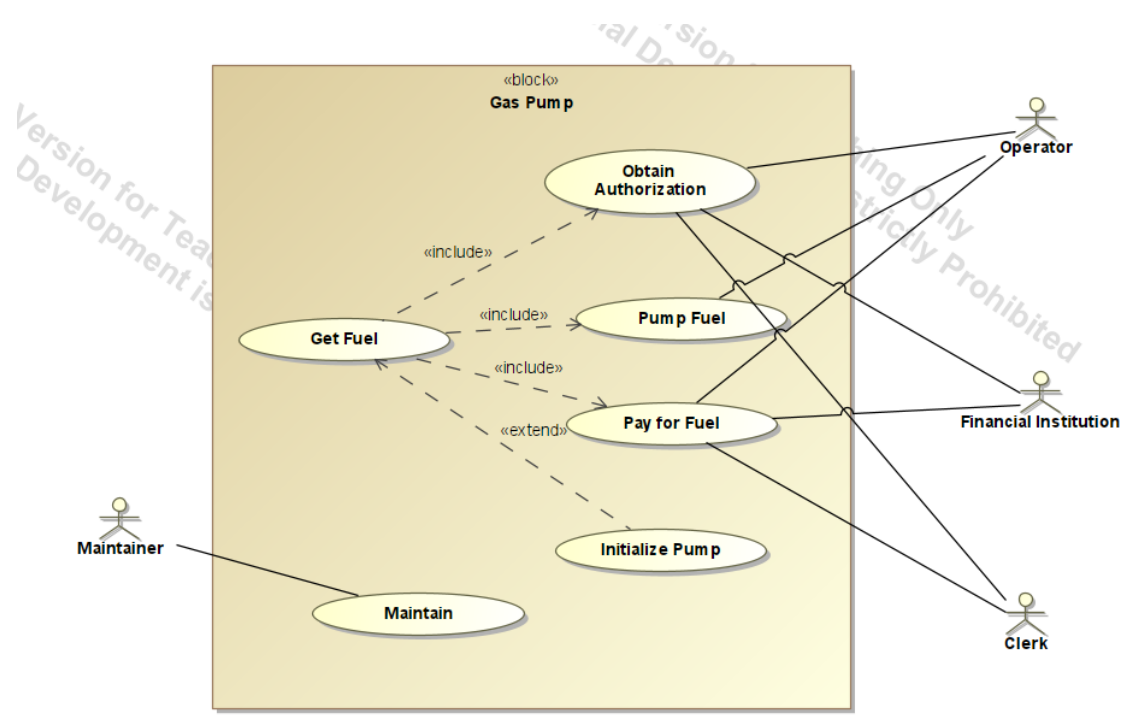


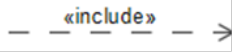





Figure 5. Gas Pump use case diagram. Adapted from Zdanis and Cloutier (2007).

Table 1 highlights the subset of use case diagram notations our team employed to develop the diagrams for the case studies serving as the focus of this capstone analysis (Auto GCAS and Firearm Safety Model). The full set of use case diagram notations can be found in the SysML standard OMG SysML 2019, v1.6 user guideline and is available in the MSOSA tool.

Table 1. Use case diagram notations. Adapted from MSOSA (n.d.).

Name	Syntax
Actor	
Use Case	
Include	
Extend	
Generalization	
Association	

2. Activity Diagram

The second type of behavior diagram in SysML is the activity diagram. The activity flow constantly changes in this model. The activity diagram is used to model complex control logic, exhibit the continuous behavior of the system, indicate the order in which actions can be performed, and present the structure of who performs each action. An activity diagram specifies active system behaviors to satisfy functional requirements using both object and control flows. The functional requirements represented in a flow include activities (rounded rectangle), decision (diamond), and merge (two diamonds) nodes. It incorporates inputs, outputs, and control among other activities. Auto-triggering of control also exists in an activity diagram.

In contrast to the use case diagram, the activity diagram illustrates only one possible activity at a time whereas the use case diagram provides a top-level view of the system behavior. Furthermore, the activity diagram provides auto-triggering of control completing activities, while the use case diagram provides only scenario-based use cases to explain the system models. These properties can be seen in the Get Fuel example, Figure 5 use case diagram, and Figure 6 activity diagram.

In line with Delligatti (2014), the activity diagram graphical notations are described below.

- Initial node—An initial node is used to start the flow of an activity.
- Activity—An activity represents the flow of functional behaviors in sequential or parallel activities depending on the conditions.
- Control flow—A control flow illustrates the transitions from one action to another action.
- Object flow—An object flow represents the creation and modification of objects by activities. In addition, an object's flow arrow from an activity influences the object. An object flow arrow from an object to an activity indicates that the activity utilizes the object.
- Decision node—A decision node represents a decision with alternative paths. The alternative paths are noted before moving to the next activity. The decision node requires a condition/guard's expression which connects to activities. A guard needs to be a true response before it moves to the next activity.
- Merge node—A merge node combines multiple alternate flows into a single outgoing flow.
- Fork horizontal—A fork horizontal depicts a single incoming flow of activity coming in and splitting into multiple concurrent flows of activities.

- Join horizontal—A join horizontal joins concurrent flows of activities into a single outgoing flow of activity.
- Swimlane—A swimlane is a group of activities that categorizes a single row/column to keep related activities together.
- Accept event action—An accept event action cannot change until a response is received from a send signal action.
- Final node—A final node illustrates the final action of an activity diagram.
- Send signal action—A send single action is modified from outside of the system. It is paired with another action of accept event action.

Figure 6 displays an activity diagram example containing the flow of activities for the Get Fuel system. The activity starts with an initial node. Once the activity initiates, the customer obtains authorization from the financial institution to pay for the fuel. The financial institution approves/disapproves the request for the customer to purchase fuel at the gas station. If the financial institution approves the request, then the customer may purchase the fuel at the gas station; otherwise, the customer is forbidden to purchase the fuel. Once approved, the customer has two ways to purchase fuel. They may use a credit card or cash. In this example, the customer pays for the fuel with cash. The gas station cashier turned on the pump station for the customer to begin pumping the fuel. The customer returns to the pump station and selects a grade and begins pumping gas into the gas tank. Upon filling the gas tank, the customer then places the nozzle back into the nozzle slot and the pumping process is completed.

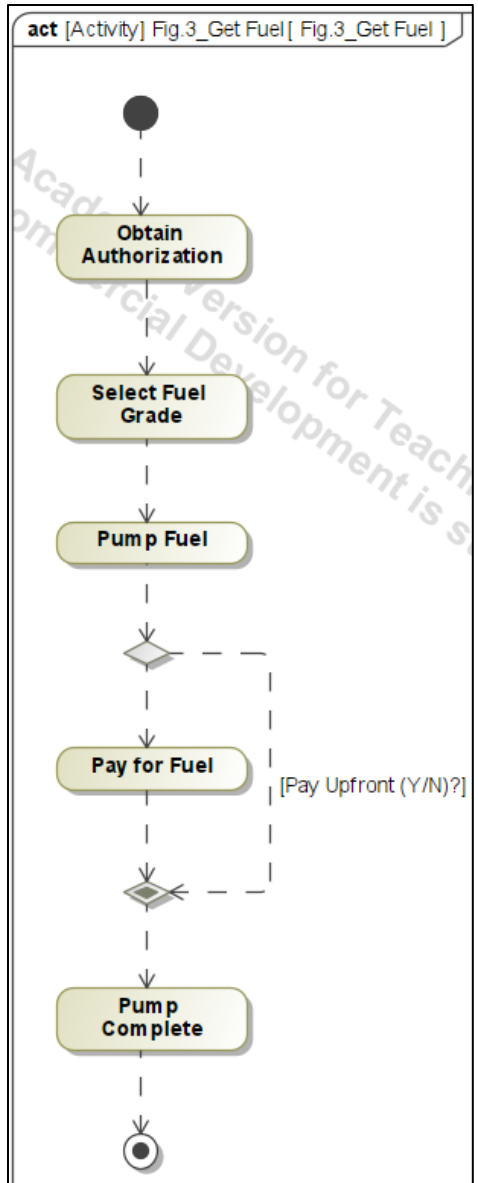







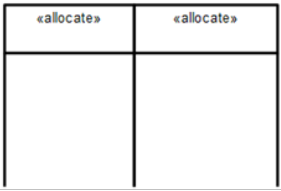

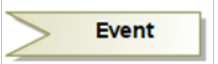




Figure 6. Fuel Pump activity diagram. Adapted from Zdanis and Cloutier (2007).

Table 2 contains the subset of activity diagram notations our team employed to develop the corresponding diagrams for the Auto GCAS and Firearm Safety case studies. The full set of activity diagram notations can be found in the SysML standard (OMG SysML Version 1.6, 2019) and is available in the MSOSA tool.

Table 2. Activity diagram notations. Adapted from MSOSA (n.d).

Name	Syntax
Action	
Control flow	
Object flow	
Decision node	
Merge node	
Fork horizontal	
Join horizontal	
Swimlanes	
Send signal action	
Accept event action	
Initial node	
Final node	

3. Sequence Diagram

The third type of diagram in a behavior diagram family is the sequence diagram. The time marches vertically down the page among lifelines and systems in the sequence diagram. The sequence of messages or operations (for example, message 1, message 2, message 3, etc.) get exchanged along horizontal lines. It uses operation calls and asynchronous signals to produce emergent behavior. Compared to activity diagrams that

provide the flow of activities, the sequence diagram provides the flow of messages between lifelines. The purpose of using a sequence diagram is to denote expected system behaviors that are passing messages between lifelines.

Figure 7 depicts an example of the sequence diagram, which includes lifelines (rectangular box), messages (arrows), and two types of messages — a synchronous message (open arrowhead), an asynchronous message (black-triangle arrowhead), and an alternative (alt) option [alt]. When compared to the activity diagram, the sequence diagram depicts events as interactions between lifelines. The activity diagram offers interactions from the perspective of activities being performed with object and control flows. These features can be seen in the Get Fuel example, Figure 6 activity diagram, and Figure 7 sequence diagram.

In line with Delligatti (2014), the sequence diagram graphical notations are described below.

- Lifeline—Represents an individual object or actor that participates in the interactions.
- Synchronous message—Represents interactions where the sender must wait for a response before continuing further interactions.
- Asynchronous message—Represents interactions where the senders can continue further interactions without waiting for a response from the receiver.
- Reply message—A message that replies to calls.
- Message to Self—Represents the message within the calling lifeline.
- Parallel option—Shows two or more sets of event occurrences that take place in parallel with each other.
- Loop option—Illustrates events and interactions that occur under specific conditions. The event occurrences in the loop option occur more than one time during an execution of a scenario.

- Alternative option—Represents a choice between two or more message sequences. This presents alternatives in a sequence diagram.

Figure 7 is the Get Fuel sequence diagram. It shows an example of a sequence diagram drawing from the Get Fuel system. For example, the “Operator,” “Fuel Pump,” “Clerk,” and “Financial Institution” are lifelines. These lifelines are interacting sequentially downward from one to another. The alternate option is used in this sequence diagram to show whether the customer’s credit card was approved to purchase fuel or not. The second alternate option is used in the sequence diagram to show whether the customer’s credit card is clerk activated or trust activated. The example continues with the customer arriving at the fuel pump station to purchase the fuel. There are two ways to purchase fuel. The first way is self-service at the fuel pump station. A customer pays with a credit/debit card. The second option is to purchase the fuel by paying the clerk inside the store. A customer may pay with the cash/credit/debit card inside the store. The first option of paying by credit/debit resumes as follows: 1) In order to purchase the fuel, the operator needs to swipe the credit or debit card at the fuel pump station. 2) The operator swipes the credit/debit card into the reader. 3) The Operator’s credit/debit card information is sent to the financial institution for approval/disapproval authorization to purchase the fuel. 4) If the operator’s credit/debit card is approved, then the 5) Operator selects the fuel grade. 6) If the operator’s credit/debit card is rejected, then 7) The operator has an inactive account and cannot select the fuel grade to pump fuel into the car gas tank. The alternate option of paying via the clerk activates these steps: 8) The operator pays at the gas station by giving the cash/credit/debit card to the clerk. 9) The clerk receives the cash/credit/debit card from the operator, and the 10) Operator alerts the clerk to activate the pump station. 11) The clerk activates the pump station for the operator to pump the fuel, and thereafter, the 12) Clerk advises the operator to select the fuel grade.

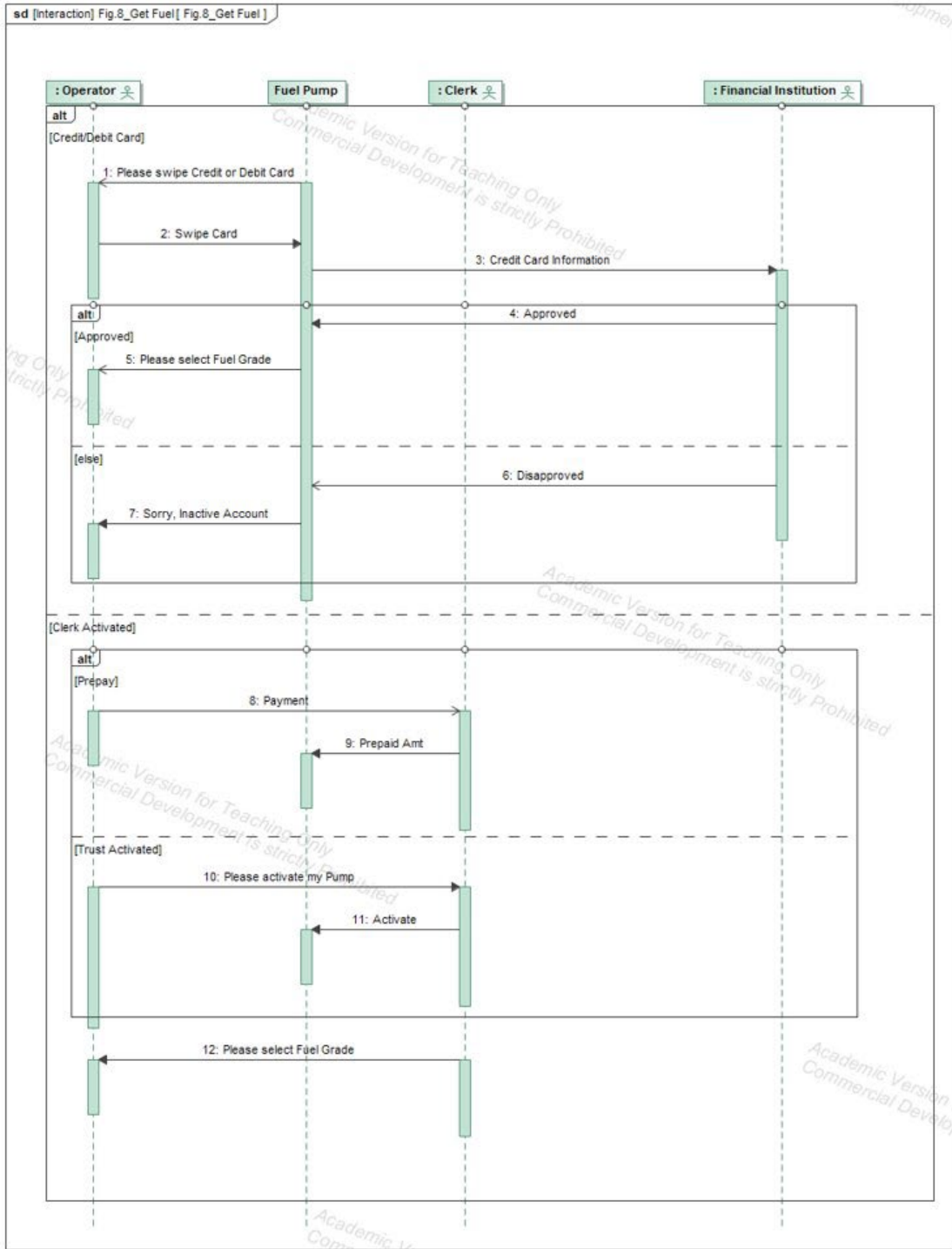










Figure 7. Fuel Pump sequence diagram. Adapted from Zdanis and Cloutier (2007).

Table 3 illustrates the subset of sequence diagram notations our team used, which were critical as they carved the pathway to perform the analysis and develop the necessary diagrams. The full set of sequence diagram notations can be found in the SysML standard (OMG SysML Version 1.6, 2019) and is available in the MSOSA tool.

Table 3. Sequence diagram notations. Adapted from MSOSA (n.d).

Name	Syntax
Lifelines	
Synchronous message	
Asynchronous message	
Reply Message	
Message to Self	
Parallel Option	
Loop Option	
Alt Option	

4. State Machine Diagram

The fourth diagram in the behavior diagram is the state machine diagram. To monitor the state progression and changes within a system, a state machine diagram is necessary. This diagram represents the changes that occur after an event (trigger) and its cascading life cycle. It delineates the life cycle of a change. The purpose of using a state machine diagram is to indicate active system behaviors for safety-critical, time-critical, and

mission-critical states in a system. The state machine diagram includes states and transitions. Figure 8 shows the example of the state machine diagram with states (rounded rectangles) and transitions (arrow with open head). A state machine diagram provides all the possible transitions from each state of the subsystem and system, whereas the sequence diagram illustrates sequential flow. These results can be seen in the Get Fuel example, Figure 7 sequence diagram, and Figure 8 state machine diagram.

In line with Delligatti (2014), the state machine diagram graphical notations are described below.

- States—Defines a current moment in time which will determine how that current moment reacts to an event occurrence.
 - Simple states—Defined as giving names to each state that gets identified in the state machine diagram.
 - Composite states—Nested substates that can be displayed in a third compartment of the state itself.
- Transition—Event or activity that causes a change between different states.

Figure 8 illustrates an example of a state diagram using the fuel pump model. The fuel pump states are *Idle*, *Authorized*, *Prime*, *Charging state*, and *Pumping*. The states are in a box and the transitions are shown via arrows. The fuel pump system initially is in the *idle* state, and it transitions to *authorized (no pressure)* once the credit/debit/clerk approved/activated is completed, then moves onto the next state. Once the state is *authorized (no pressure)* it transitions to *prime* state with the valid fuel selection and pump primed transition. Once the state is in *prime* it transitions to *pumping (flow)* state with squeeze nozzle transition. Once the state is in *pumping (flow)* then it transitions to *prime* state with release nozzle transition. Once the state is in the *prime* state then it transitions to *charging state (no pressure)* with credit or debit auth-receipt response transition. Once the state is in a *charging state (no pressure)* then it transitions to an *idle* state with an account charged & receipt response.

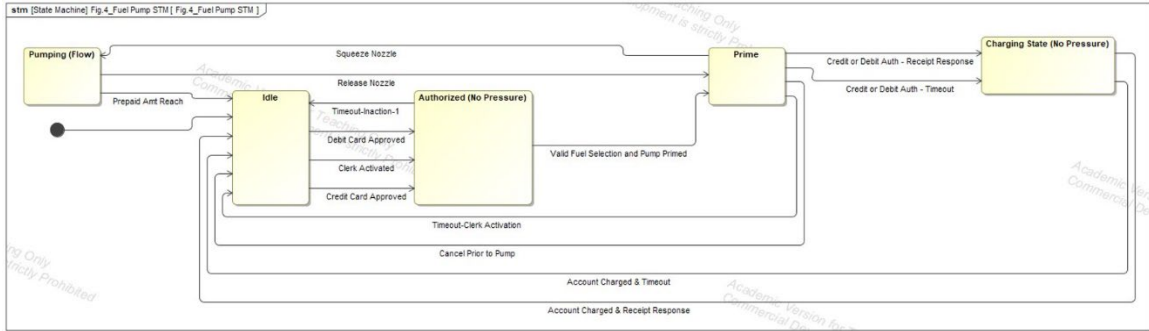


Figure 8. Fuel Pump state machine diagram. Adapted from Zdanis and Cloutier (2007).

Table 4 contains the subset of state machine diagram notations our team used to develop the diagrams for the Auto GCAS and Firearm Safety case studies. The full set of state machine diagram notations can be found in the SysML standard (OMG SysML Version 1.6, 2019) and are available in the MSOSA tool.

Table 4. State machine diagram notations. Adapted from MSOSA (n.d.).

Name	Syntax
States	
Transitions	
Initial state	
Final state	
Composite State	

B. MONTEREY PHOENIX (MP)

MP has a distinct advantage in modeling system behaviors when compared to SysML. While most engineers designing a system focus their efforts and understanding on how the system works to provide a desired utility or functionality to a customer, systems

engineers need to ensure that a holistic view of the system’s behavior is acceptable to all users in the expected operational environment. MP has demonstrated utility in enumerating possible behaviors of abstract systems such as logistic supply chains and decision-making processes for emergency responders. MP uses Events and Relationships based on logical relationships, defined by the modeler, to generate event traces, which are graphical representations of the system’s behavior logic similar to SysML sequence diagrams.

The research requires a basic understanding of what an Event is in MP. Events are the primary building blocks of the MP model. An event in MP can be used to represent any concept of a system (e.g., component, activity, state, transition, condition). There are Root, Composite, and Atomic event types. A Root event is comprised of one or more atomic and/or composite events. Composite events are comprised of one or more atomic events and are included in a composite or root event. An atomic event is a single component or concept of a system’s functionality. Figure 9 is a good visual depiction of Auguston’s concept of what constitutes an event in MP. The fact that it can represent various aspects of the system being modeled makes the concept of events flexible and abstract for ideation and behavior scenario visualization and interpretation.

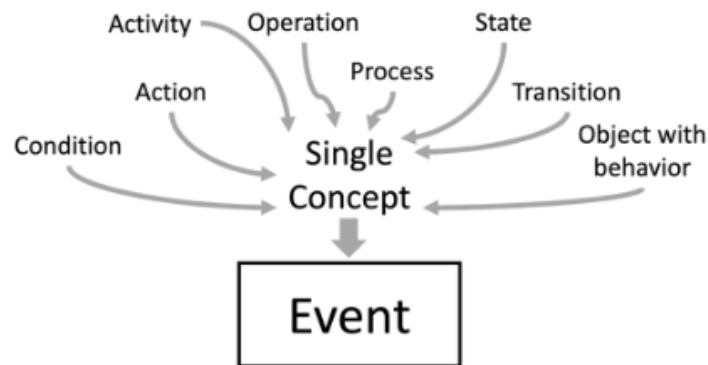


Figure 9. Auguston’s concept of an abstract event. Source: Giammarco (2022).

The relationships between events come in two primary types, inclusion and precedence. There is a third type, which is user-defined and represented by a solid blue

arrow but is not utilized in any of our case study models. The includes relationship simply means events can contain one or more lower-level events. It is represented by a dashed arrow between events. Figure 10 (right) shows a depiction of the event hierarchy and how the two basic relationships between events are displayed in the event traces (left). An example would be to define a root event as a “Car_A.’ Car_A includes several atomic events (or components) such as a door, a hood, or a wheel. Depending on the fidelity desired in the model, these component pieces could also be defined as composite events. For example, a wheel could be defined as containing both a tire and rim as the atomic events within the wheel.

A precedence relationship simply defines the order in which events occur. It is represented by a solid arrow between events. These relationships are not limited to events within a single root. These relationships also depict interactions across root events. A simple example would be the need for a driver to flip the turn signal switch before the turn signal bulb starts to blink. In MP, a modeler might have one root as the “Driver” and another root as “Car_A.” From this example, the root “Driver” would include an “Initiate_Turn” event that precedes Car_A’s “Indicate_Turn” event.

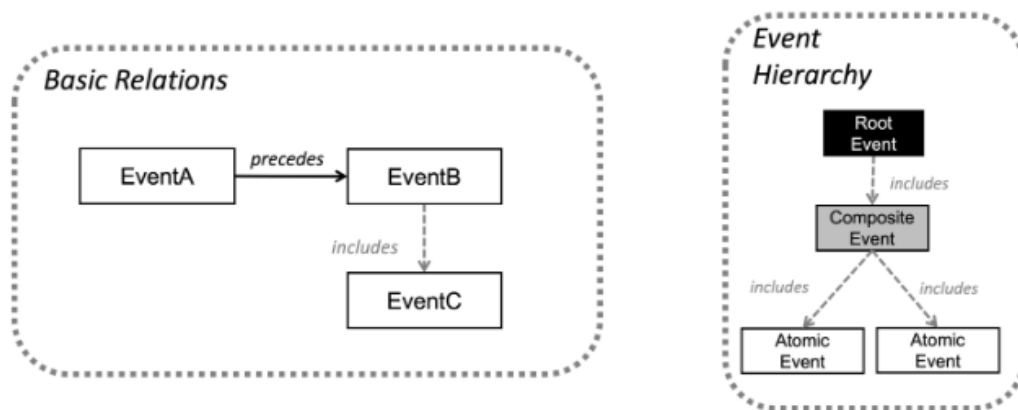


Figure 10. Event trace visualizations of events and relationships in MP.
Source: Giammarco (2022).

Complex algorithms for system behaviors can be modeled with MP. The following sections will discuss the processes involved in creating an MP model and the extraction of the various behavior diagrams it can create.

1. Language, Approach, and Tool

Monterey Phoenix is a lightweight formal method and tool for modeling system behaviors (Giammarco and Troncale 2018). The language and approach were developed by Mikhail Auguston and are based on logic statements that relate a system's *events* through relationships (or connections) discussed above (inclusion, precedence, and user-defined).

The approach to building a model in MP requires a methodical approach of “build a little, test a little.” The current user interfaces to MP, in both the MP-Firebird (web-based) and MP-Gryphon (a downloadable application) modeling environments, require an understanding of the MP coding language.

Appendix A: MP Model Source Code contains the various MP code samples used in this research for the two case studies that are presented later.

The first step in building a model in MP is to outline the *root events* within the system. These roots are the systems, subsystems, or players within the system's operational environment. For example, in a system of systems (SoS), each constituent system would be a root. The signals, interactions, functions, etc., within each root, will be separate atomic or composite events contained within that root event. Once all the roots are defined by the events necessary for the behavior to be modeled, they will be linked by an inclusion connection to the root.

As events are added to each root, the modeler needs to understand several key pieces of information. What is the starting point of the behavior to be modeled? How many starting points are there? What are the end states of the system's behavior? How many end points are there? Are there multiple paths that can reach the desired end state from a given starting point? What signals are involved in the paths?

A modeler does not necessarily need to have all the answers, but they do need to have the insight to know what does not look correct. MP will automatically generate all the possible behavior paths based on the relationships the user creates in the model. When the modeler believes all relevant events and paths are present for each root, the next step is to start connecting the precedence relationships (interactions) across the various root events.

Figure 11 shows basic behavior logic composition examples involving inclusion and precedence relationships within a root.

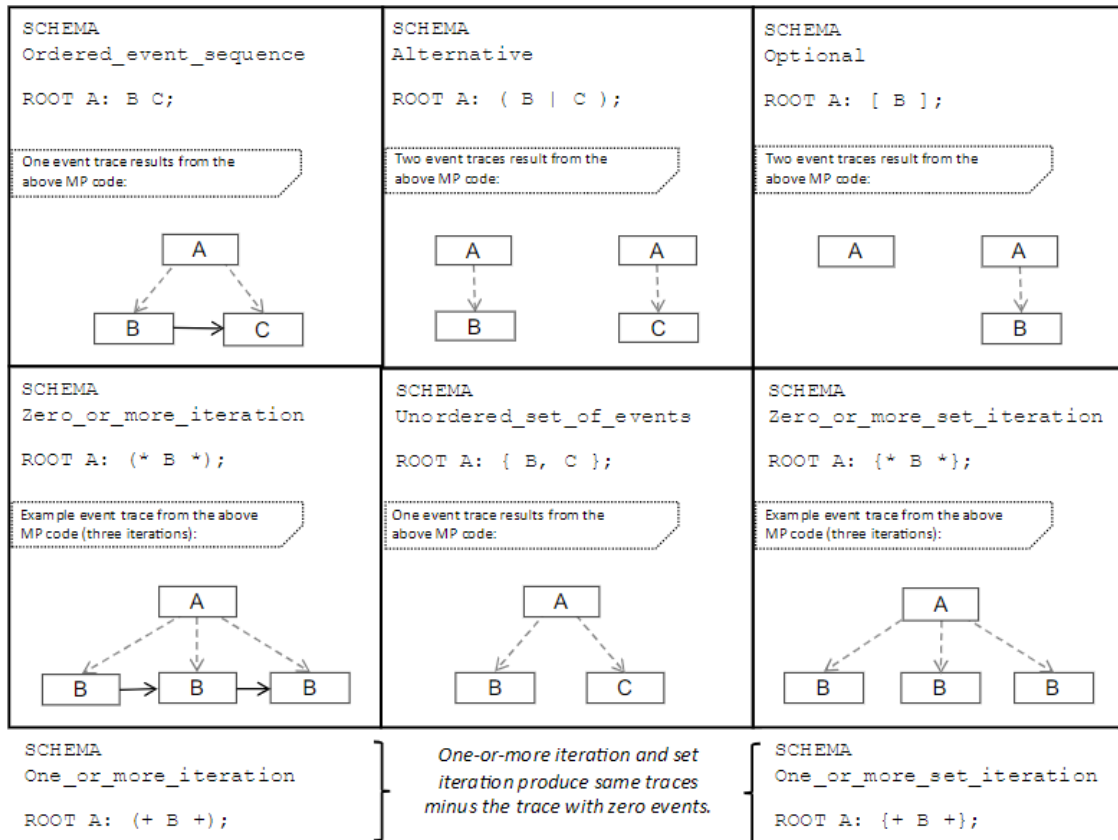


Figure 11. Example grammar rules with corresponding event traces. Source: Giammarco (2022).

Each time the modeler adds a precedence relationship to the model, best practice requires running the code to generate the event trace models. Examining the resulting event traces for the expected precedence connection is paramount. Precedence arrows between roots on an MP event trace are analogous to interaction arrows on a SysML sequence diagram. Each time a new relationship is added to the model, it is adding a constraint to the system's behavior. It is possible to over-constrain the model unintentionally when overloaded or contradictory constraints result in fewer traces than expected. Therefore, it is best practice to “build a little, test a little” to enable pinpointing of the MP code responsible for the resulting traces. It is in the event traces and this iterative model-building process that emergent behaviors can be discovered (typically resulting from the relaxation

of constraints). The modeler can also produce MP global view behavior diagrams that depict the current structure of the system being modeled. The following sections will describe how to extract the MP activity diagrams and MP state-machine diagrams.

2. Extraction of Behavior Diagrams

MP shines in its ability to create scope-complete event traces, which shows all the possible paths of the system's event relationships based on the user's model for the run scope. It is the ability to generate scope-complete event traces that makes discovering emergent behaviors in the system much easier. Scope, as it relates to modeling in MP, is a setting that allows the user to define a set number of times through a particular logic loop within the model. Scope-complete refers to MP's ability to generate individual event trace diagrams that show all the possible logic flows in separate diagrams based on user-defined event relationships (Giammarco et al. 2018). The automatic generation of scope-complete event traces is a non-existent capability in current SysML modeling tools.

Once the user has built the MP model by defining all the appropriate relationships, creates the event traces, examines the generated event traces for obvious errors and emergent behaviors, and fixes any issues, while confirming all remaining traces are valid, the modeler is able to get several other behavior diagrams from the same model.

a. Event Trace Diagram

MP event trace diagrams are similar in interpretation to simple sequence diagrams. The way to read the information being presented is from top to bottom. In MP-Firebird, the root elements are listed along the top of the diagram in green boxes. Beneath each root (connected by dashed inclusion arrows) is the atomic (blue box) and composite (orange box) events that describe a given behavior. Within the vertical area beneath the roots, read the topmost event and then follow the solid black arrows from left to right (or right to left, depending on the order of root elements) first. By following the solid black arrows from one event to the next event(s) eventually you reach the bottom of the diagram. There is a definitive start and end point for each event trace. Each event trace shows one possible path through the sequence of events to go from the start point to the end point, based on the user's model.

“Example 32: Model of ATM Withdrawal with State Chart” (Auguston n.d.), provides the following event traces in Figure 12.

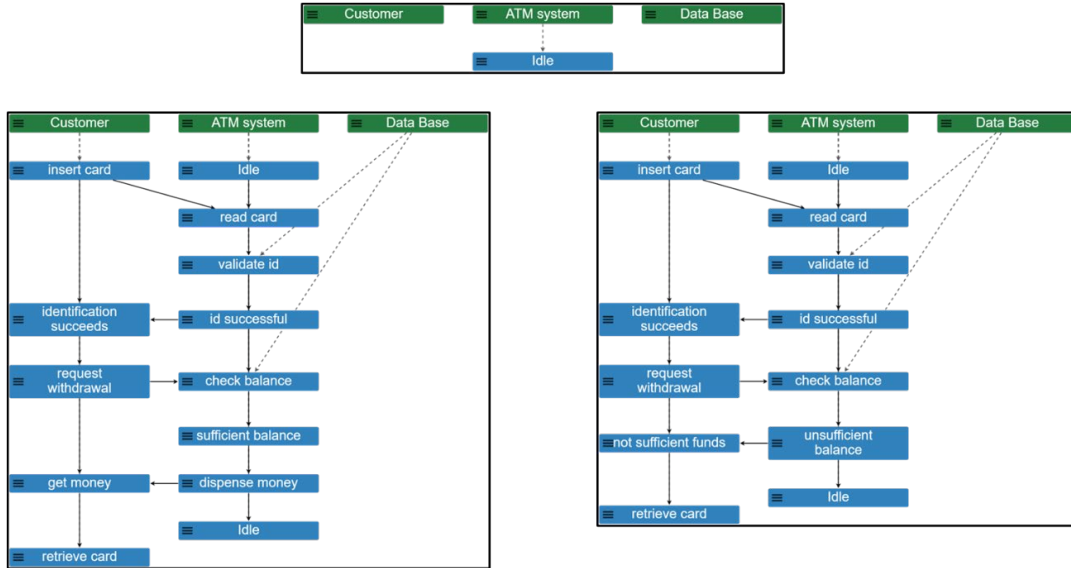


Figure 12. Event trace examples. Adapted from Auguston (n.d.).

Figure 12 shows the event traces from the same MP model that generated Figure 13 and Figure 14. To create Figure 13, it required adding the command to show the activity diagrams for each root to the existing code in Auguston’s example. Figure 14 is a product of the example code without any change necessary.

b. Activity Diagram

The MP activity diagrams are separately generated for each independent root. This is a unique feature and requires interpretation along with the event traces, due to the lack of availability in the current tools for graphed interactions across the separate activity diagrams. Figure 13 shows the three activity diagrams that correspond to the event traces in Figure 12. The grey boxes represent the actions, and the solid black arrows show the flow and direction of the object throughout the activities. MP contains symbols representing decision (hollow diamond), fork (horizontal bar), and join (horizontal bar) nodes. It does not contain a symbol for a merge node. The independence of the activity diagrams by root event gives insight into the logic within each root.

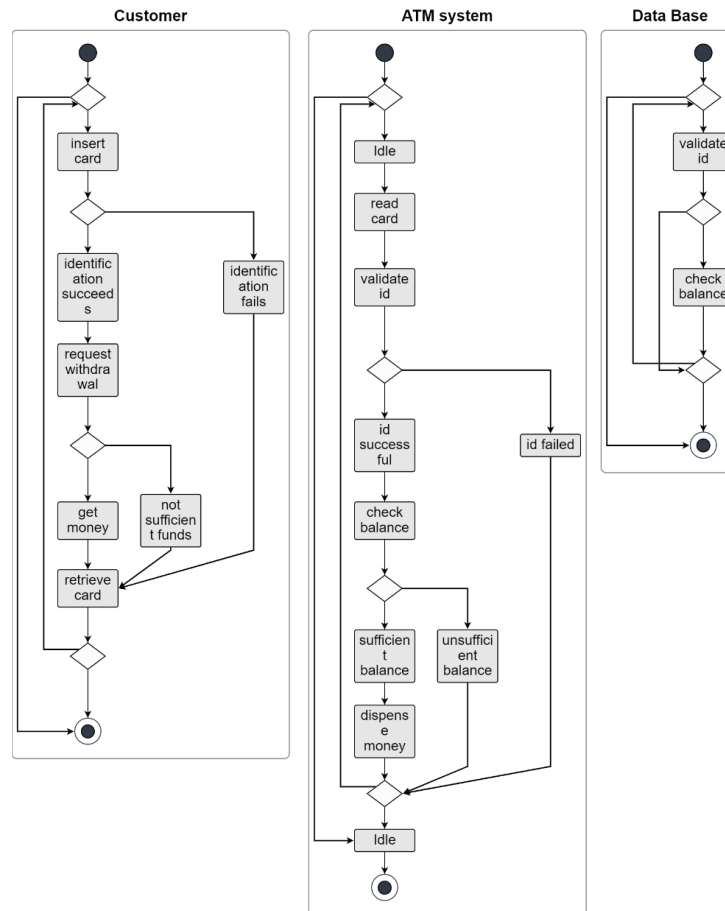


Figure 13. Example activity diagrams. Adapted from Auguston (n.d.).

c. State Machine Diagram

MP state machine diagrams utilize a simple visual vocabulary of symbols similar to SysML. In MP-Firebird, states are depicted by grey boxes and the transitions are depicted with the use of solid black arrows. The state machine diagrams within MP require a specific block of code to generate the diagram. It utilizes the events already created for the generation of the event traces. For the example state machine diagram in Figure 14, it can be observed that not all the ATM system's events are utilized, only the ones that describe the state of the system. The transitions are events that occur within other roots of the ATM System schema and become labels for the transitions within the diagram.

ATM state transition diagram

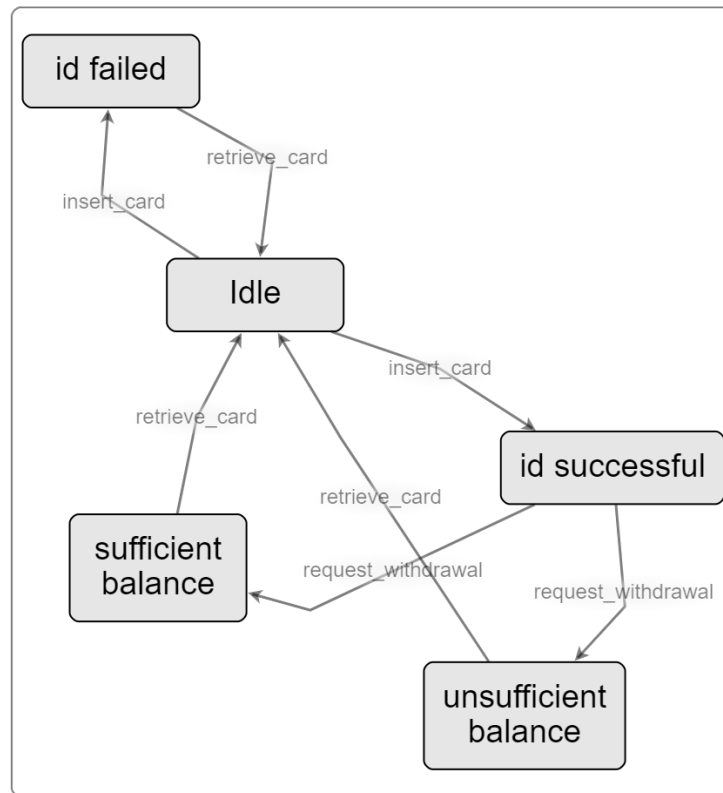


Figure 14. Example of an MP state machine diagram. Adapted from Auguston (n.d.).

C. CHAPTER SUMMARY

This chapter has presented a fundamental understanding of the System Modeling Language and Monterey Phoenix behavior diagrams. It presented the pertinent SysML vocabulary and examples applicable to the behavior models under examination, as well as the equivalent MP models and vocabulary. The MP code, while shown in rudimentary examples in Figure 11, and the user interface were intentionally left out due to irrelevance to the comparison of model outputs. The next chapter will discuss the development of the case study systems used for developing the behavior models in both languages.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CASE STUDY MODELS DEVELOPMENT

For the purposes of this research, the two systems utilized as case studies for the comparison of behavioral models are the Automatic Ground Collision Avoidance System (Auto GCAS) and the Firearm Safety Model. The Auto GCAS model is a highly technical system with many signals and reactions. The Firearm Safety Model is a procedure of interactions with a simple system (a firearm), its user (the shooter), and the operational environment (the shooting range). These two systems are well suited to behavior modeling due to the interactions of a system with its user and operational environment and the allowance for direct comparisons of the different modeling languages.

In order to standardize a naming convention in this report, messages in event trace and sequence diagrams and actions in activity diagrams are referenced throughout the text within quotation marks, such as “Command Power Up.” For state machine diagrams, the states and transitions are displayed in italics, such as *Power Up*.

A. AUTO GCAS MODEL

As a starting point, the team designs and models behavior diagrams based on the Auto GCAS, which is developed by Lockheed Martin Skunk Works, the Air Force Research Laboratory, and the National Aeronautics and Space Administration (NASA). The Auto GCAS consists of complex collision avoidance algorithms and utilizes digital terrain elevation data, aircraft performance, and precise navigation data. The Auto GCAS algorithm is enabled at start-up and executed in the background during the flight. When the program detects an imminent ground impact, it provides alerts and prompts the pilot to act. If the pilot does not take necessary corrective action, the Auto GCAS will take temporary control and divert the aircraft to a safe region. Once the aircraft returns to safe conditions, the system will return control to the pilot (Lockheed Martin n.d.). Figure 15 illustrates the top-level operation concept of the Auto GCAS.

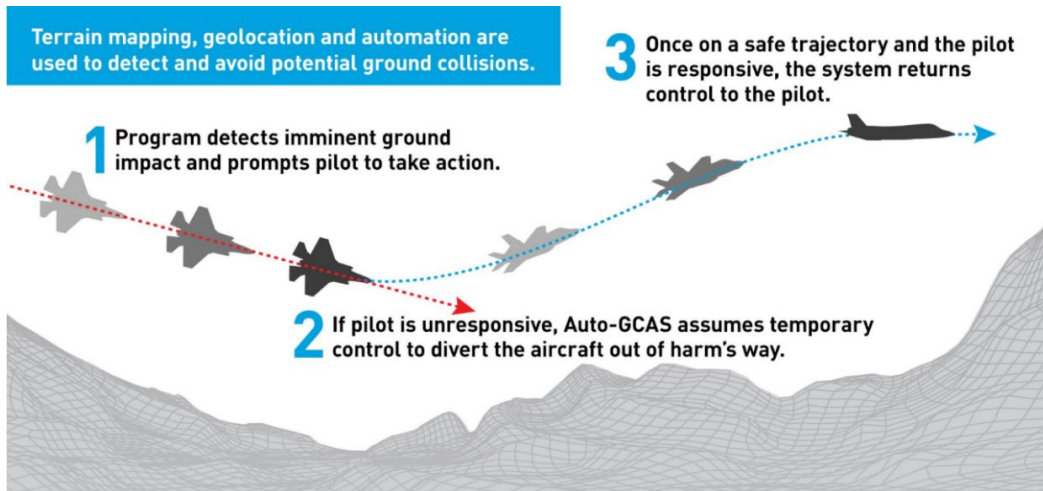


Figure 15. What is Auto GCAS. Source: Lockheed Martin (n.d.).

From the Auto GCAS operational concept, the Model Wreckers use the SysML sequence diagram to establish the event sequence. Figure 16 illustrates the SysML sequence diagram for the Auto GCAS model, which is comprised of two main actors — the pilot and the aircraft. The terrain and mission commander are actors that the aircraft communicates with.

Initially, the pilot executes “1: Command Power Up” to the aircraft. Once it receives the command, the Aircraft completes “2: Perform System Initialization” to determine the system operational status and “3: Report System Operational Status” to the pilot. In the next sequence, the Alternative loop (alt) is utilized with “Operational Status is No Go” as the Guard condition. When the operational status is not good, the Aircraft performs “4: Disable Auto GCAS” and executes “5: Display Warning Cues” to the pilot. Otherwise, the operational status is deemed as good and the Aircraft will “6: Enable Auto GCAS” and “7: Display Real-time Operational Status” to the pilot. While the pilot executes “8: Perform Mission,” the Aircraft will concurrently “9: Scan Terrain” and “10: Execute Auto GCAS Algorithm in Real-time.” In the diagram, the Parallel (par) notational concept is utilized to illustrate the parallel events where two events occur at the same.

When the Aircraft “11: Detect Imminent Ground Impact,” the Aircraft executes “12: Provide Warning Alerts” to the pilot. The Alternative loop (alt) and the Guard condition “Pilot Take Corrective Action (Y/N)?” are utilized to determine the alternative

scenarios. When the Aircraft receives “13: Perform Corrective Action” from the pilot, it executes “14: Continue Evaluate Ground Impact Condition” and will “15: Stop Warning when Safe Condition is Determined.” Otherwise, when no corrective action is determined or safe condition has not been met, the Aircraft will “16: Gain Temporary Control” and “17: Execute Fly-up Command.” During this time, the Aircraft will “18: Provide notification that the Auto GCAS is Activated” to the pilot. Once the Aircraft reaches the safe condition, it executes “20: Activate Autopilot Mode” and concurrently performs both “21: Continue Monitoring Environment” and “22: Send Aircraft Status to Mission Control.” Only when the Aircraft receives “23: Provide Control Input” from the pilot, it then “24: Deactivate Autopilot Mode” and “25: Return the Control Authority to Pilot.” The pilot performs “26: Return to Base,” and then “27: End Mission.”

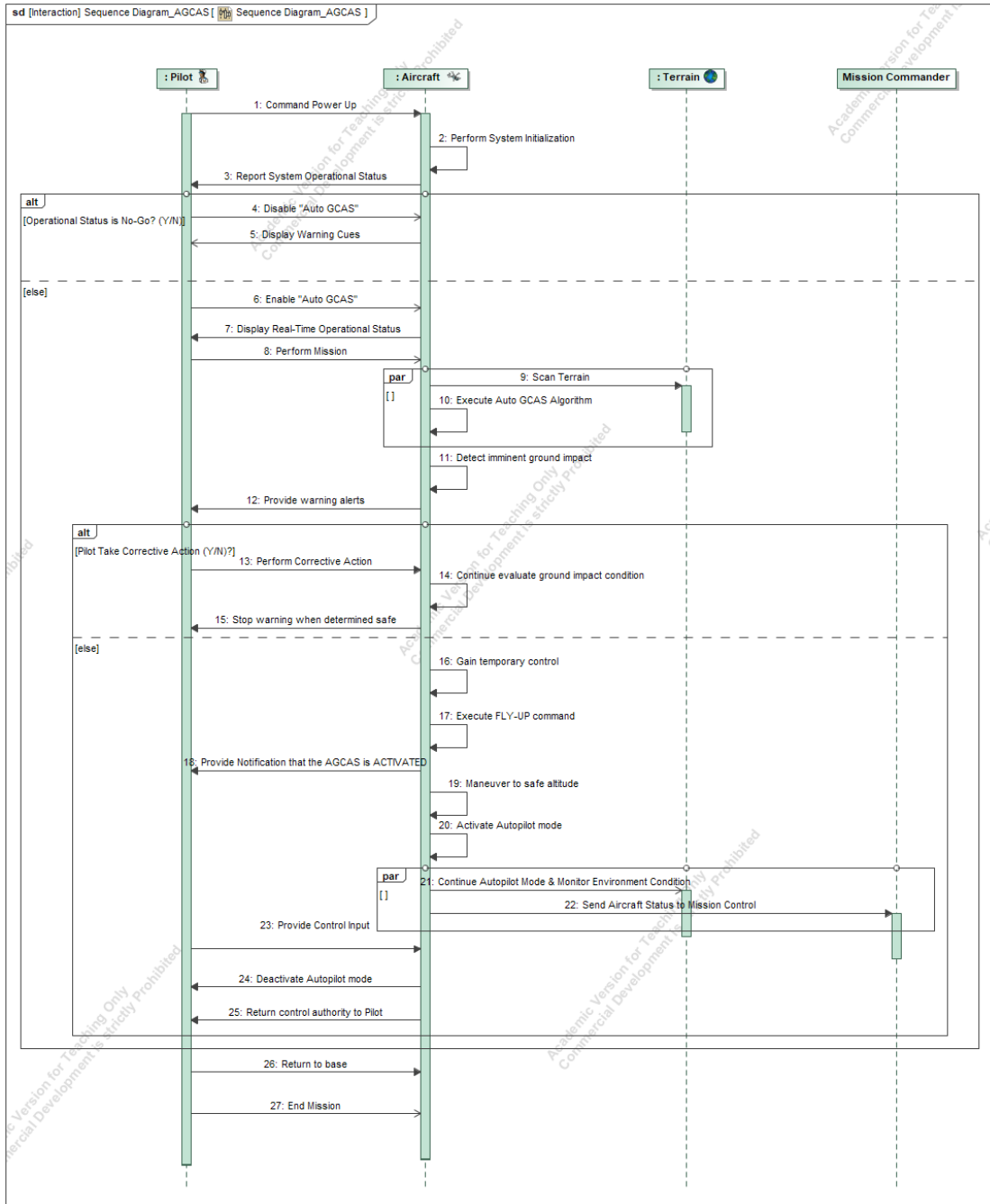


Figure 16. SysML sequence diagram, illustrated using the Auto GCAS model

B. FIREARM SAFETY MODEL

Firearms have been around for centuries and have evolved into various types and styles that serve various purposes. The core design principle common among all firearms is that they propel projectile(s) out of a barrel, using an explosive propellant. For the purposes of our research, a firearm is defined as a weapon that falls into one of the following categories: rifle, shotgun, or handgun. The firearm (or weapon, used synonymously and interchangeably) is also complete, functional, and manually fired. The exact differences between the types of firearms are not important for our research. Firearms are inanimate machines that can sit indefinitely until an outside force interacts with them in some way. Any manually operated firearm is incapable of changing its current state, without direct inputs from an operator (or shooter, used synonymously and interchangeably) or severe changes to its environment (e.g., earthquake, hurricane, explosion, etc.). For this reason, it was a good candidate for modeling in behavior diagrams.

Firearms have been used in hunting, self-defense, law enforcement, and militaries for centuries, which provides a nearly infinite number of use cases to choose from. However, there is one common use case for all firearms that translates well to a simple state machine model: the proper and safe handling of a firearm.

This case study looks to model the proper interactions and behaviors between the firearm and the operator. These interactions are the fundamental, most crucial rules that need to be observed by any operator of any firearm to ensure safe operation in a training environment.

Safety is always the top priority when handling firearms. The National Rifle Association of America's three fundamental rules when handling any firearm are:

1. "Always keep the firearm pointed in a safe direction.
 2. Always keep your finger off the trigger until you are ready to shoot.
 3. Always keep the firearm unloaded until ready to use."
- (National Rifle Association 2022)

These three simple rules are the overarching behavior constraints that everyone needs to adhere to when handling any firearm. The U.S. Army trains its soldiers to a higher standard. According to TC 3–20.40, Appendix C, the "Rules of Firearm Safety" are:

- “Rule 1: Treat every weapon as if it were loaded.”
- “Rule 2: Never point the weapon at anything you do not intend to shoot.”
- “Rule 3: Keep the finger straight and off the trigger until ready to fire.”
- “Rule 4: Ensure positive identification of the target and its surroundings.” (Department of the Army [DA] 2019)

These two sets of rules are the basis for the development of the Firearm Safety Model. There are several overlapping principles in both that guided the development for the interactions and behaviors used in the system behaviors.

The first order of business for every shooter is to verify that the chamber is empty. Operators need to understand how to operate the weapon’s action and perform a functional check of the weapon to verify if the chamber is empty. Even if handed a firearm and told it is empty, the operator in possession of the weapon is responsible to know the state of the weapon. It is this mindset that led to the development of the Firearm Safety State-Machine Behavior Model as seen in Figure 17.

The assumptions behind the development of the model are:

1. The operator is already trained and familiar with the operation of the firearm, including the proper procedure for clearing and performing a function check of the weapon.
2. The firearm is one (of several) being used during a range training exercise for a class of shooters taking specialized training for this firearm.
3. The firearm being modeled has a semi-automatic action. A semi-automatic firearm is capable of automatically reloading the chamber with a new round of ammunition each time the previous round is fired from the weapon.
4. The weapons are laid to rest on the firing line bench in a cleared state after each shooter is finished with their firing of the weapon.

In the following narrative, the weapon states are in italicized text for quick identification. The weapon is in an *Inert* state as the operator approaches the firing position.

This *Inert* state of the weapon means it is not being handled by an operator and is in a static state of rest. This distinction is important because only the operator of a firearm truly knows whether the weapon is in a *Safe*, *Unsafe*, or *Broken* state. These states cannot be truly known until the operator has inspected the weapon firsthand.

While the operator believes the weapon is in a *Safe* state (unloaded and safety on), the weapon's state upon being picked up by a shooter is unknown until they have verified the weapon's chamber is empty and performed a function check of the firearm. This initial handling of the weapon by an operator puts the weapon into a *Status Unknown* state. From this state, the operator must cycle the action to confirm the chamber is empty. If the chamber is empty, the safety switch is verified/placed in the safe position, and the action is left open, rendering the firearm *Safe*. If the ammunition is found in the chamber, it is in an *Unsafe* state. This requires the ammunition to be removed from the chamber, the safety switch is then verified/placed in the safe position, and the action left open to move the firearm into the *Safe* state.

The third possible state of the firearm from the *Status Unknown* state is *Broken*. This state will be declared if the weapon fails any function check as part of the inspection/clearing process. If the firearm is determined to be *Broken*, then it will be placed aside and sent off to be repaired.

Once the firearm is in the *Safe* state, the operator will then be ready to load the ammunition into the weapon. Firearms are considered loaded if there is any ammunition in the chamber, or if there is ammunition in a magazine (fixed or removable) that can be chambered by a simple cycle of the firearm's action. For the purposes of our state transitions in this model, to move from *Safe* to *Standby*, the type of "loaded" being referred to requires a round of ammunition to be chambered and ready to fire.

Once the operator has gotten the firearm into *Standby*, the following states usually happen in a very rapid succession. *Ready* and *Aiming* could have been shown as happening in parallel, but for the sake of our modeling effort were left to be serial states of the weapon, to emphasize the safe handling of firearms. By moving the safety switch to the off position, the weapon's trigger is now live and will fire the round as soon as the shooter squeezes the trigger. Between *Ready* and *Firing*, is *Aiming*, which requires a lot of concentration from

the shooter. As the shooter is *Aiming* the firearm to hit the intended target, he or she is also making sure there is nothing beyond or near that target that could be hit by mistake. The shooter is also maintaining an accurate sight picture as he or she squeezes the trigger, being sure not to move the sight off the target for proper follow through and to make sure the shot hits its targeted point of impact.

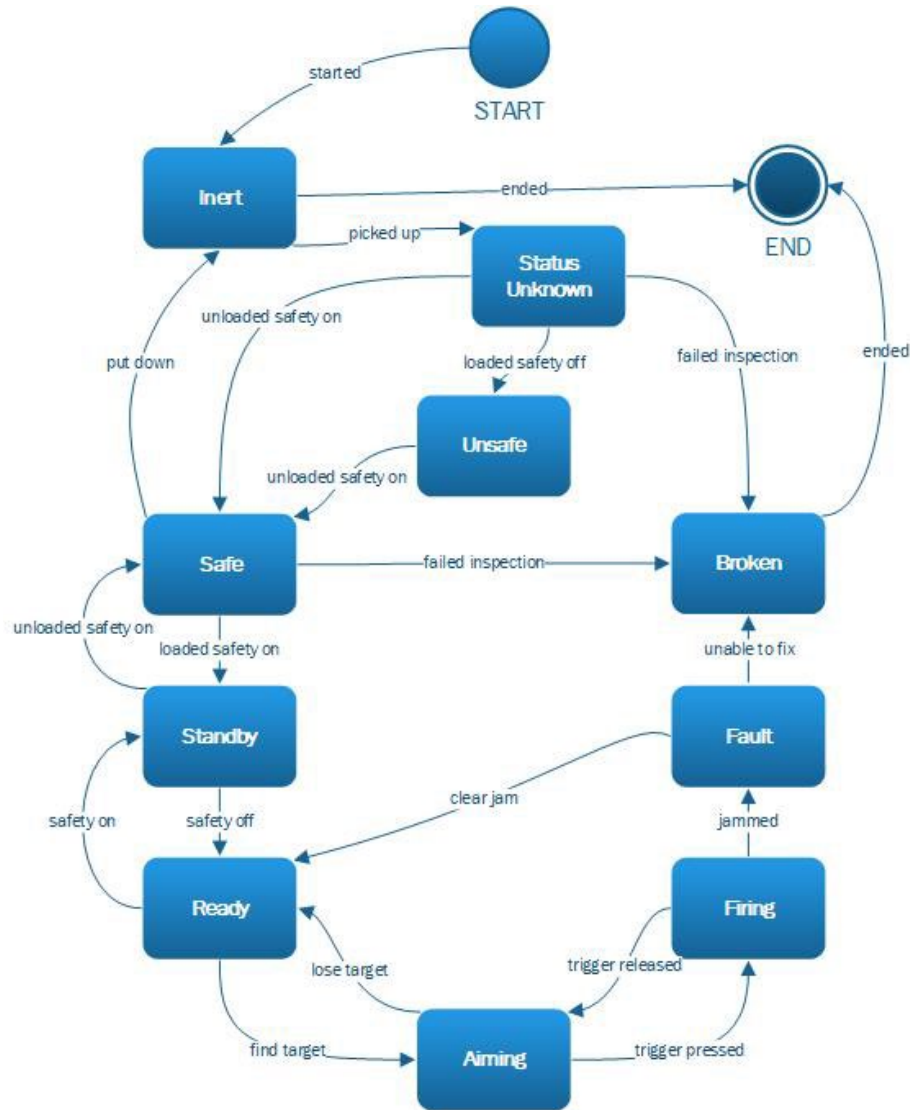


Figure 17. Firearm Safety state machine diagram

As the weapon is *Firing*, the bullet is being propelled down the barrel by the exploding propellant within the cartridge casing. The round's cartridge casing is being

extracted from the chamber, ejected from the weapon, the next round is being stripped from the magazine, and the new round is being loaded back into the chamber to be fired at the next target. All these listed actions occur in rapid succession, which explains why they are simply wrapped into the *Firing* state. Should anything go awry with any of these *Firing* functions, the likely outcome is a jamming of the firearm's action. Most jams can be cleared by the shooter and put the firearm back into a *Ready* state. If the operator is unable to fix or clear the jam, then the weapon is unloaded of any remaining ammunition and declared to be *Broken*, and put aside for further maintenance and repair by proper personnel.

The above descriptions are very generic and simplified to give the reader a basic understanding of the behavioral interactions between a firearm and operator in a controlled training environment where safety is the top priority for everyone involved.

C. CHAPTER SUMMARY

This chapter has provided a high-level description of the two system case studies that are used in the comparisons of the SysML and MP behavior models. The Auto GCAS is a safety feature in fighter aircraft intended to prevent ground collisions. The Firearm Safety Model describes the interactions between a shooter and a firearm. Each was chosen because of its applicability to modeling system behaviors and would allow comparisons of both modeling languages on a one-to-one comparison. The next chapter will walk through the analysis of each of these case studies with respect to the SysML and MP behavior models.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. MODEL ANALYSIS

In this chapter, the Model Wreckers discuss the similarities and differences between MP event trace and SysML sequence diagrams, MP and SysML activity diagrams, and MP and SysML state machine diagrams. In section A, the team compares and contrasts MP and SysML behavior diagrams developed based on the Auto GCAS case study model. In section B, the team discusses observations on optimizing the model structure and the development of MP state machine based on the Firearm Safety case study model.

A. SIMILARITIES AND DIFFERENCES

This section covers the Auto GCAS Model case study. In order to provide the same basis and perspective that are unique to each type of behavior diagram, the team has developed multiple sub-scenarios — derived from the Auto GCAS Model case study — in order to effectively compare and contrast each behavior diagram type generated by MP and MSOSA.

1. MP Event Trace vs. SysML Sequence Diagram

In this section, the Model Wreckers introduces four operational scenarios including both simple and complex conditions in order to investigate the similarities and differences between MP event trace and SysML sequence diagrams from different view angles.

The first scenario models a simple operation in normal conditions and is illustrated in Figure 18. In this scenario, following the power-up command from the pilot, the aircraft performs the system initialization and built-in test to verify the operational status of all aircraft sub-systems. The built-in test result is no-go and thus the aircraft disables the Auto GCAS functionality. With the Auto GCAS functionality being disabled, the pilot decides to abort and end the mission.

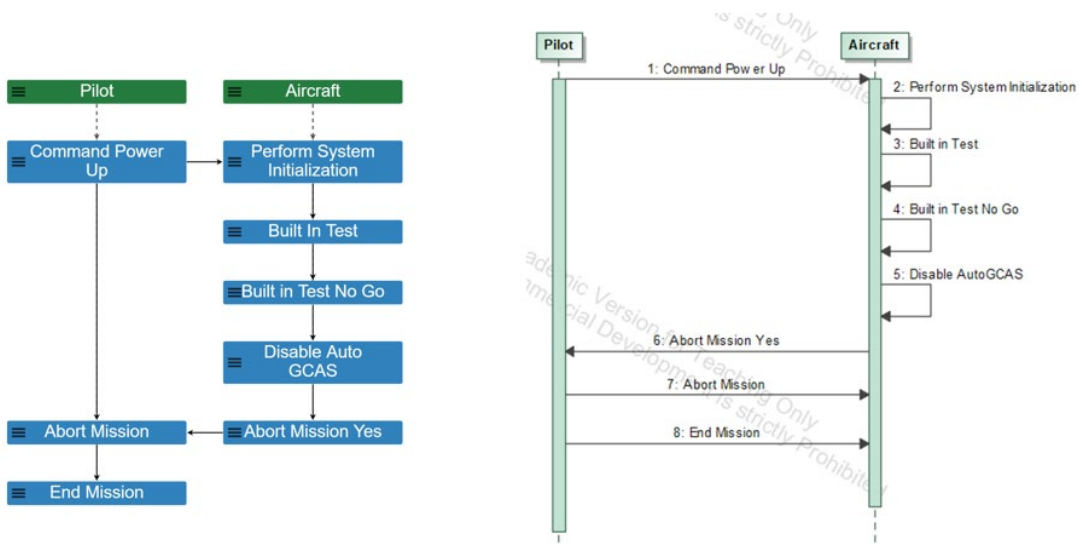


Figure 18. MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with “Built-in Test No Go”

The second scenario models a normal flight condition in which no imminent ground impact is detected. Figure 19 depicts how MP and SysML each illustrate parallel events and sequences in the diagrams. The main purpose of this scenario is to analyze how MP and SysML illustrate the event trace and sequence diagrams when there are multiple events/sequences occurring at the same time (concurrency). In this scenario, when the aircraft’s built-in test result is good, the Auto GCAS functionality is enabled and executed in the background during the flight. During the mission, the aircraft concurrently scans terrain data and displays Auto GCAS operational status in real-time. Once the pilot completes the mission, the pilot returns to base and ends the mission.

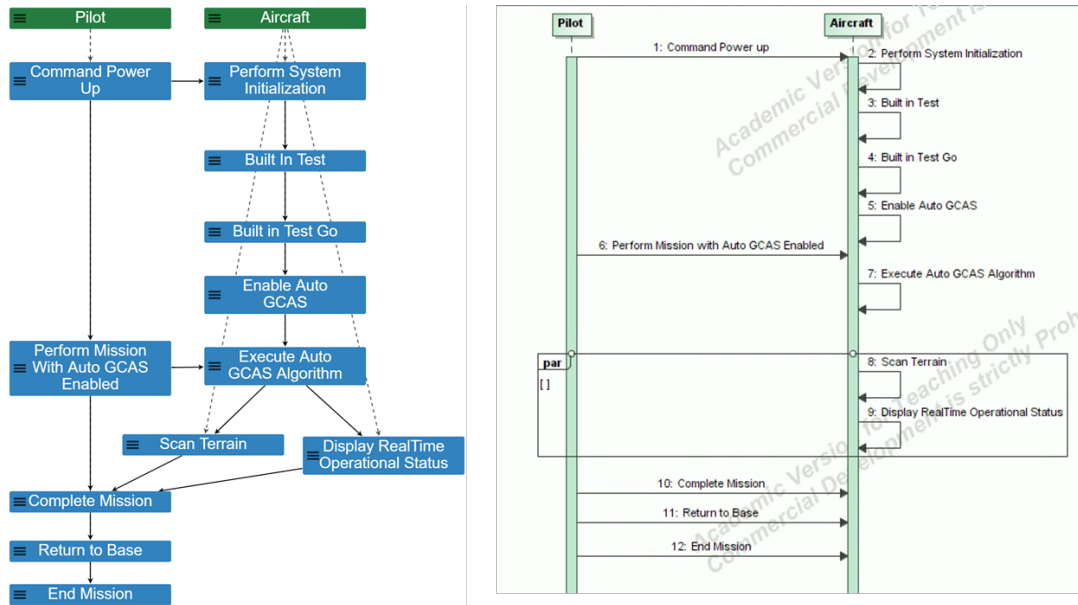


Figure 19. MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with “Auto GCAS Enabled”

The third scenario is illustrated in Figure 20 and models a flight scenario in which the Auto GCAS detects the imminent ground impact and provides warning alerts so that the pilot can successfully recover to the safety zone. In this scenario, while the pilot is performing the mission with Auto GCAS enabled, the Auto GCAS algorithm detects the imminent ground impact and provides warning alerts to the pilot. While the pilot is taking corrective actions, the aircraft continues to monitor and evaluate the flight condition. Once the Auto GCAS determines the aircraft has been recovered to safety condition, the aircraft stops the warning. The pilot completes the mission, returns to base, and ends the mission.

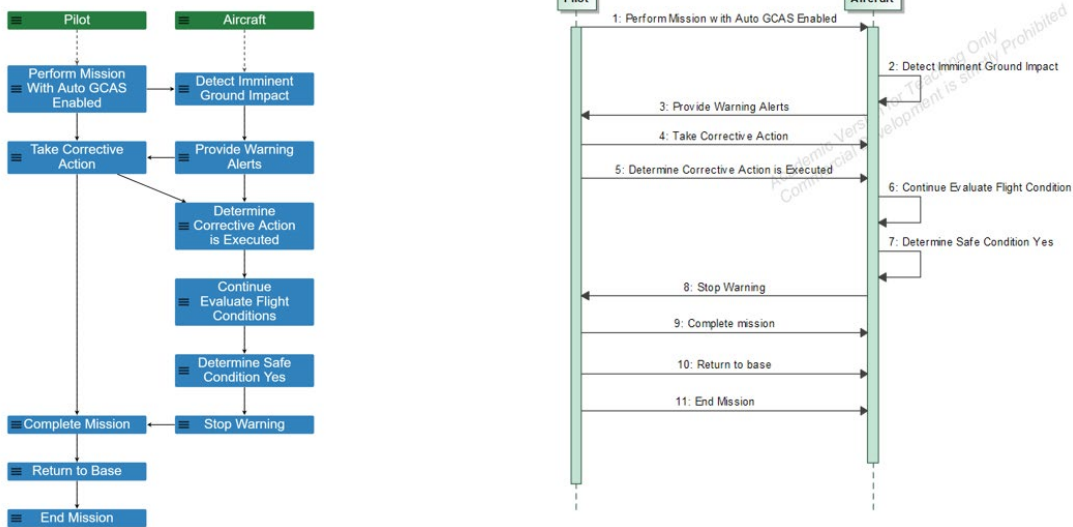


Figure 20. MP event trace (left) vs. SysML sequence diagram (right), illustrated using the Auto GCAS impact flight condition when the pilot takes corrective action

The fourth scenario models a more complex condition and is illustrated in Figure 21. In this scenario, the aircraft determines that no proper corrective action is executed and thus automatically activates the Auto GCAS function, gains temporary control of the aircraft, and executes the fly-up command. During this time, the aircraft displays the Auto GCAS fly-up cue to notify the pilot that the Auto GCAS algorithm is temporarily controlling the aircraft. Once the aircraft has reached the safe condition, the aircraft then activates autopilot mode. With autopilot mode activated, the aircraft will send the status to the mission commander and continue to monitor the surrounding conditions. Additionally, only after pilot input and feedback is detected, the autopilot mode is deactivated, and the aircraft returns control authority to the pilot. Once the pilot completes the mission, they return to base and end the mission.

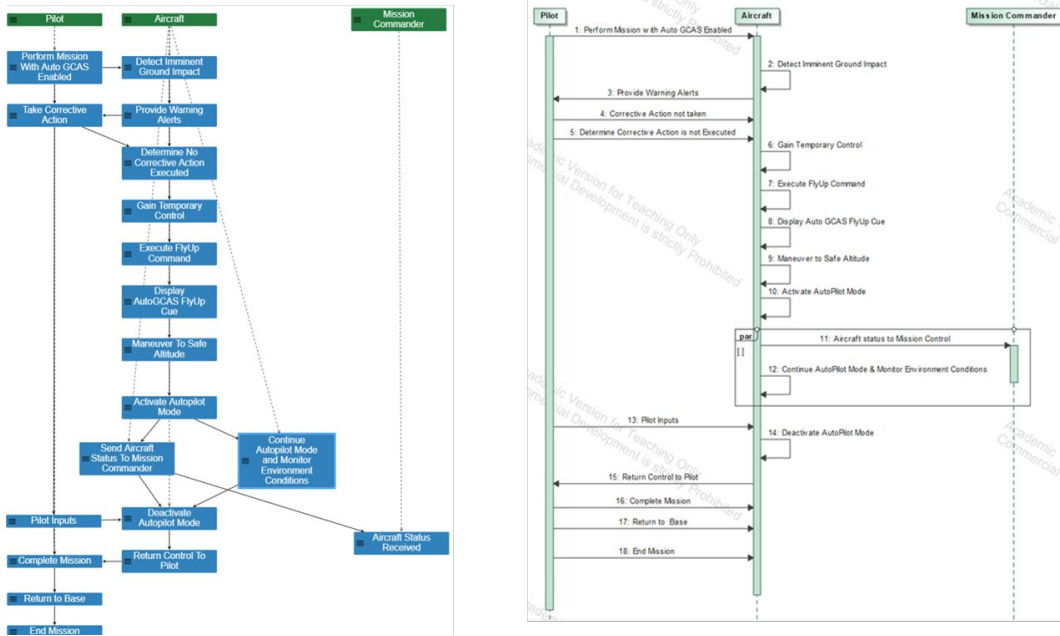


Figure 21. MP event trace (left) and SysML sequence diagram (right), illustrated using the Auto GCAS impact flight condition with fly-up executed

a. Similarities

From the diagrams in all four scenarios, we can see that both MP event traces and SysML sequence diagrams can illustrate similar operational events and sequences. For example, in Figure 19, the pilot “Command Power Up” to the aircraft. Then, the aircraft executes multiple events and actions such as “Perform System Initialization,” “Built In Test,” “Built In Test—Go,” and “Enable Auto GCAS.”

Also, both MP and SysML support displaying the actors and the sequence of the events in vertical order. As a result, the users can easily identify the actors that originate and command the specific events. In Figure 19, the actors are the pilot and the aircraft. MP groups the events performed by pilot and aircraft into vertical lifeline—pilot and aircraft. The MP event trace diagram shows that “Command Power Up,” “Perform Mission with Auto GCAS Enabled,” “Complete Mission,” “Return to Base,” and “End Mission” are performed by the pilot while other actions are performed by the aircraft. Similarly, SysML sequence diagrams show that these sequences and events originated from the pilot to the aircraft’s lifeline.

b. Differences

Aside from the similarities between MP event traces and SysML sequence diagrams, there are also many differences between them including the scope and scale of the diagrams, the placement of actor blocks on the lifelines, the method to illustrate events and sequences in parallel, and the meaning behind different types of arrows.

First, the most noteworthy difference between SysML sequence diagram and MP event trace diagram is the scope and scale of the diagram. In many cases, SysML sequence diagrams are used to illustrate the full scope of the behavior model in one diagram, with multiple event flows in one diagram. In contrast, MP event trace diagrams contain only one single flow through the behavior logic in the MP schema. MP event trace diagrams are derived automatically from coordinated root events specified in the schema as behavior examples or instances with the intended use of helping the model author ascertain whether or not the schema's behavior logic is correct in all possible expressions up to the scope limit. In contrast, SysML sequence diagrams are manually constructed and can be simulated step by step to verify the logic. For example, the SysML sequence diagram in Figure 16 illustrates and covers all four operational scenarios when compared to each individual MP event trace diagram in Figure 18 through Figure 21. When utilizing the alternative loop option (alt) in SysML, all four operation scenarios are combined utilizing the alt notations with the guard options as "Operational Status = No Go" and "Pilot Take Corrective Action = Y/N."

Second, although both MP event trace and SysML sequence diagrams display the actor/lifeline and events in vertical order, the notations are implemented differently. SysML utilizes arrows, which originate from one lifeline to the other lifeline, and places them on the vertical lifelines to represent the sequences of the event. In contrast, MP uses rectangular boxes and arrows to achieve similar purposes. Additionally, SysML places the legend of the sequences/events above the arrows while MP places the legend within the rectangular box. Especially, SysML automatically assigns a number along with the legend so that the users can easily identify the order of the sequences. For example, in Figure 19, MP places the "Command Power Up" within a rectangular box and places an arrow that originates from the pilot's lifeline to the aircraft's lifeline, meaning the pilot commands the

aircraft to power up and the aircraft will not “Perform System Initialization” without the precedence event—“Command Power Up” from the pilot. With the same concept, SysML places an arrow that originates from the pilot lifeline to the Aircraft lifeline and places the legend of the event and the event number above the arrow—“1: Command Power Up.” For the next event, SysML assigned the sequence incrementally as “2: Perform System Initialization.” In addition, the arrow for this sequence is a “Message to Self” type that has its head and tail attached to the same lifeline—the aircraft—to illustrate that the aircraft performs the action “2: Perform System Initialization” within its internal system.

MP also uses different colors to illustrate the hierarchy of events. For example, the root events (or lifeline in SysML) are illustrated in green color. The atomic events are illustrated in blue color, and composite events are illustrated in orange color. In Figure 19, MP applies the green color for the “Pilot” and “Aircraft” to illustrate that these are the root events. The other events such as “Command Power Up” and “Perform Mission with Auto GCAS Enabled” are atomic events that belong to the “Pilot” root event, while “Perform System Initialization” or “Built-In Test” are atomic events that belong to “Aircraft” root event. In contrast, SysML does not differentiate the hierarchy of events by colors. However, SysML does support displaying the label of the actors on the lifelines along with the customizable icons. Another subtle difference is that the actors on the lifelines are also different concepts in SysML and MP. In SysML, the actor is a component or a block while in MP, the actor is considered a root or composite event.

Third, MP and SysML illustrate parallel sequences and events that occur at the same time differently. In MP, the parallel events/sequences are illustrated by branching from the parent event to multiple lower-level events. In Figure 22, while the aircraft “Execute Auto GCAS Algorithm,” the aircraft concurrently “Scan Terrain” and “Display Real-Time Operational Status.” MP illustrates these parallel events by branching “Scan Terrain” and “Display Real-Time Operational Status” from the parent event - “Execute Auto GCAS Algorithm.” Once completed, the events are merged into the parent event - “Complete Mission.” In contrast, SysML uses the parallel option (Par) to illustrate actions that occur in parallel. The events are stacked on the lifeline and numbered ascendingly, such as “8: Scan Terrain” and “9: Display Real-time Operational Status.” Although both

MP and SysML support displaying events and sequences in parallel, the implemented method in the current SysML diagrams is not ideal. The team believes that stacking parallel events vertically on the lifeline gives them the appearance that these activities are in sequential order, not in parallel order, to novice users.

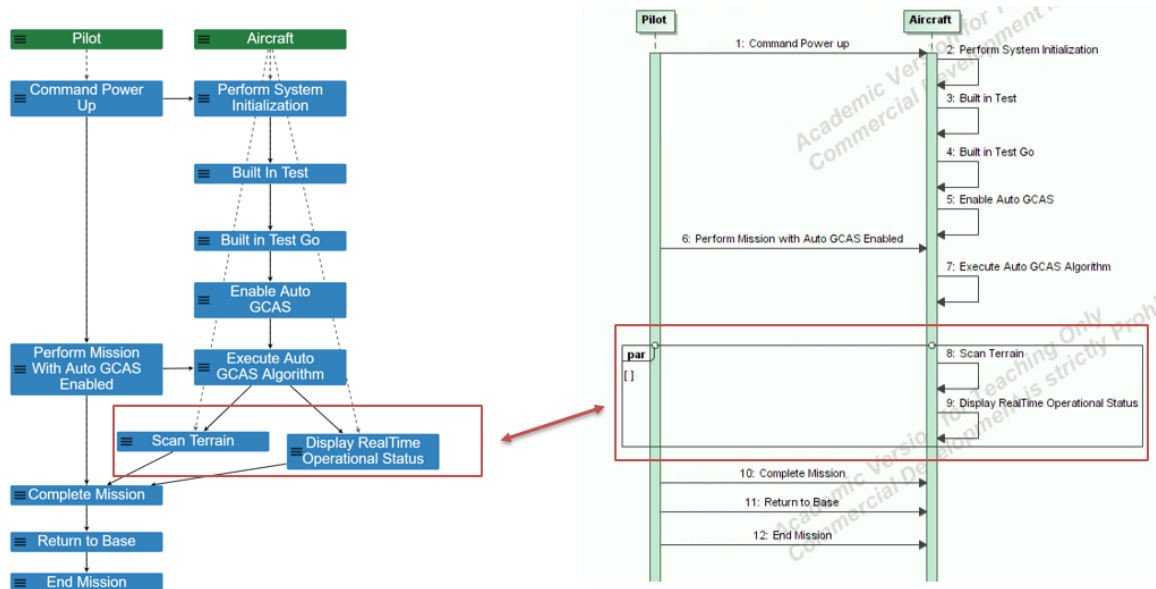


Figure 22. Differences in parallel events notational concepts between MP event trace (left) and SysML sequence (right) diagrams, illustrating using the Auto GCAS normal condition with Auto GCAS enabled

Additionally, the meanings behind the types of the arrow are different between MP and SysML. Because of high level of abstraction, MP uses only three types of arrows: the solid-line arrow, the dash-line arrow, and the solid blue arrow. MP uses solid-line arrows to represent the precedence relationship (ordering relation) between different events. The dash-line arrows represent the inclusion relationship between events. Even though the dashed arrow may be overlapped with the solid-line arrow, they are always present on the diagram and may be viewed by moving the boxes (MP event traces are directed acyclic graphs of dependencies (precedence, inclusion, user-defined) among events). In Figure 23, MP uses the solid-line arrow to illustrate that “Command Power Up” is the preceding event before the “Perform Mission Initialization” event can occur. In addition, MP uses the dash-

line that originates from the root event Pilot to atomic events such as “Command Power Up” or “Perform Mission with Auto GCAS Enabled” to illustrate that all of these atomic events are included within the Pilot root event. Also, MP supports user-defined relations that permit users to define any other relations between events with the blue solid-line arrow. The user can make user-defined relations start-end to one event so that the head and tail of the arrow are attached to the same event box.

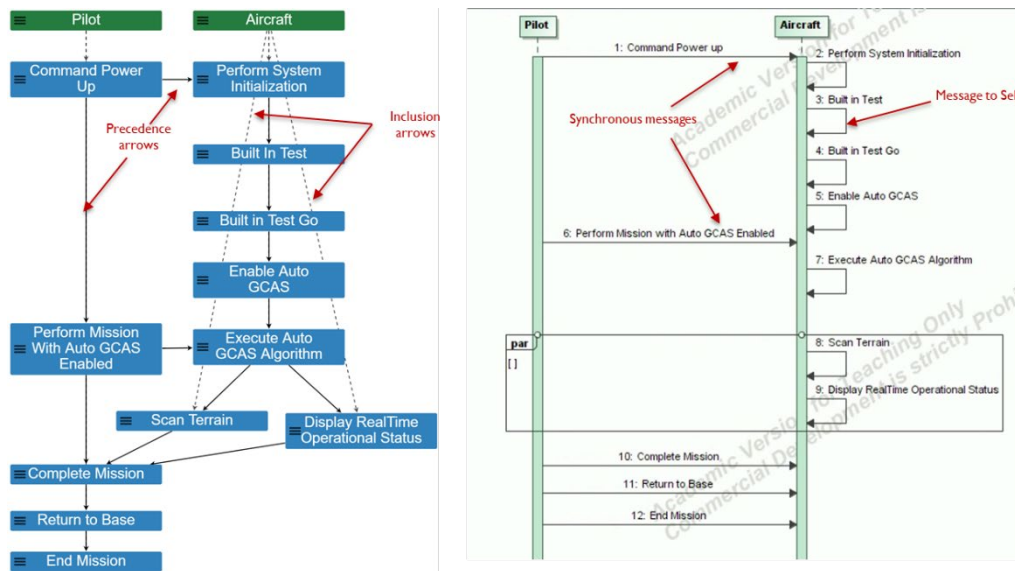


Figure 23. Differences in types of arrow notational concept between MP event trace (left) and SysML sequence (right) diagrams, illustrating using the Auto GCAS normal condition with Auto GCAS enabled

In contrast, SysML utilizes various types of arrows to illustrate the relationship between events and actors, which include synchronous messages, asynchronous messages, reply messages, and message to self. Depending on the type of the events, not all arrow types are utilized in SysML sequence diagram. For example, in Figure 23, only synchronous messages such as “1: Command Power Up” and Message to Self such as “2: Perform System Initialization” are utilized for this sequence diagram.

2. MP Activity Diagram vs. SysML Activity Diagram

To investigate the similarities and differences between MP and SysML activity diagrams, the Model Wreckers develop the activity diagrams using the same Auto GCAS operational scenarios. The team investigated the activity diagrams in two operational scenarios, which included 1) Normal flight condition where there is no imminent ground impact, and 2) Impact condition where Auto GCAS detects the imminent ground impact and provides proper alerts so that the aircraft can recover to a safe condition.

Figure 24 illustrates the SysML activity diagram in normal flight conditions. This activity diagram shows three possible operational scenarios that follow routes: a) 1 – 2 – 4 – 7 – 8, b) 1 – 3 – 5 – 8, and c) 1 – 3 – 6 – 8. Following route 1 – 2 – 4 – 7 – 8, once the pilot completes “Command Power-up” to power up the aircraft, the aircraft follows with “Perform System Initialization” and “Built-in Test” to determine the system operational status. With “Built-in Test” status as “Go,” the aircraft follows with “Enabled Auto GCAS” and “Execute Auto GCAS Algorithm” in the background. The pilot follows with “Perform Mission with Auto GCAS Enabled” and during this time, the aircraft performs both “Scan Terrain” and “Display Real-time Operational Status.” These activities are executed simultaneously (in parallel). Once the pilot performs “Complete Mission,” the pilot performs “Return to Base” and then “End the Mission.” Following route 1 – 3 – 5 – 8, with the “Built-in Test” status as “No-Go,” the aircraft follows with “Disable Auto GCAS” and the pilot decides to “Abort Mission,” then “End Mission.” Alternatively, following routes 1 – 3 – 6 – 8, the pilot decides not to abort the mission and continues to “Perform Mission without Auto GCAS.” Once the pilot performs “Complete Mission,” they “Return to Base” and then “End Mission.” Clearly, SysML activity diagrams can illustrate multiple activities and operational scenarios in one diagram.

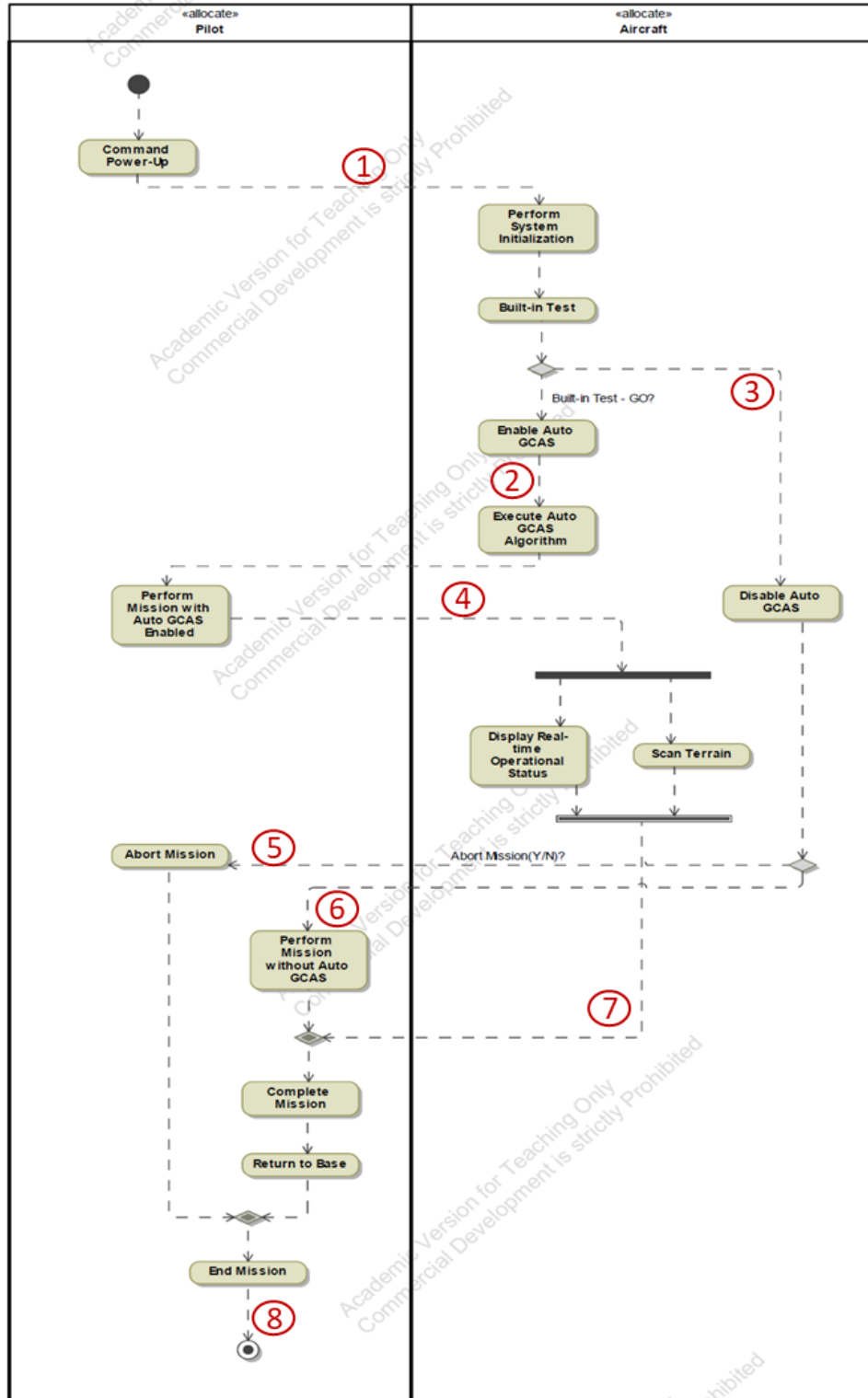


Figure 24. SysML activity diagram, illustrated using the Auto GCAS normal flight condition model

Similarly, Figure 25 illustrates the SysML activity diagram when an imminent impact condition is detected. In addition to the notations used in Figure 24, some other SysML concepts such as send signal action (“Aircraft Status to Mission Control”) and accept event action (“Pilot Inputs”) are also utilized in this activity diagram. The diagram shows multiple possible operational scenarios that may follow various routes such as a) 1 – 2 – 3 – 5 – 10, b) 1 – 2 – 3 – 6 – 2 – 3 – 5 – 10, c) 1 – 2 – 4 – 7 – 8 – 9 – 10, and so on. Following routes a) and b), while the pilot completes “Perform Mission with Auto GCAS Enabled,” the aircraft follows with “Detect Imminent Ground Impact” and “Provide Warning Alerts.” While the pilot performs “Take Corrective Action,” the aircraft will “Continue Evaluate Flight Condition.” If the condition is determined as safe, the aircraft will follow route a), which proceeds with “Stop Warning.” Once the pilot performs “Complete Mission,” they “Return to Base” and “End Mission.” However, if the condition is determined not safe, the aircraft will follow route b), in which the aircraft will continue to “Provide Warning Alerts” and repeat the other activities as illustrated in the diagram. Following route c), when the aircraft determines the corrective action is not received and/or not correctly executed, it then activates the Auto GCAS algorithm, “Gain Temporary Control” and automatically performs other activities in sequences including “Execute Fly-up Command,” “Display Auto GCAS Fly-up Cue,” “Maneuver to Safe Altitude” and then “Activate Autopilot Mode.” Once autopilot mode is activated, the aircraft will perform “Continue Autopilot Mode & Monitor Environment Condition” and send “Aircraft Status to Mission Control” concurrently. The aircraft will remain in autopilot mode until the Accept Event Signal—“Pilot Inputs” is received. Then, routes 9–10 will be continued where the aircraft will “Deactivate Autopilot Mode,” “Return Control to Pilot,” and allow the pilot to “Complete Mission,” “Return to Base,” and “End Mission.”

The SysML activity diagram in Figure 24 and MP event trace diagrams in Figure 18 and Figure 19 are developed for normal flight conditions. Similarly, the SysML activity diagram in Figure 25 and MP event trace diagrams in Figure 20 and Figure 21 are developed for impact flight conditions. In the next section, we will compare and contrast the similarities and differences between MP and SysML activity diagrams.

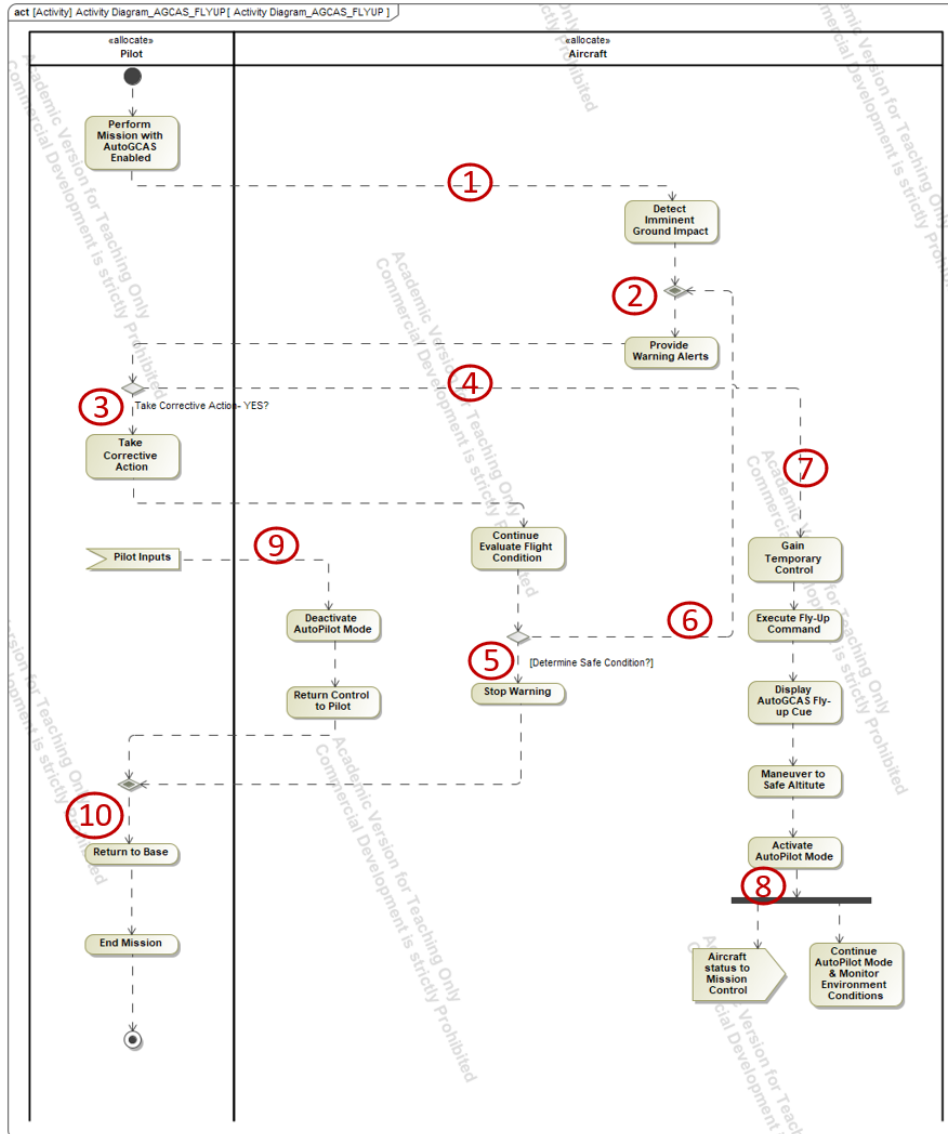


Figure 25. SysML activity diagram, illustrated using the Auto GCAS impact flight conditions model

a. Similarities

Figure 26 demonstrates the side-by-side comparison between MP and SysML activity diagram in normal flight conditions. The figure shows that both MP and SysML support several similar notational concepts for activity diagrams, which include the initial node, action box, decision node, and final activity node.

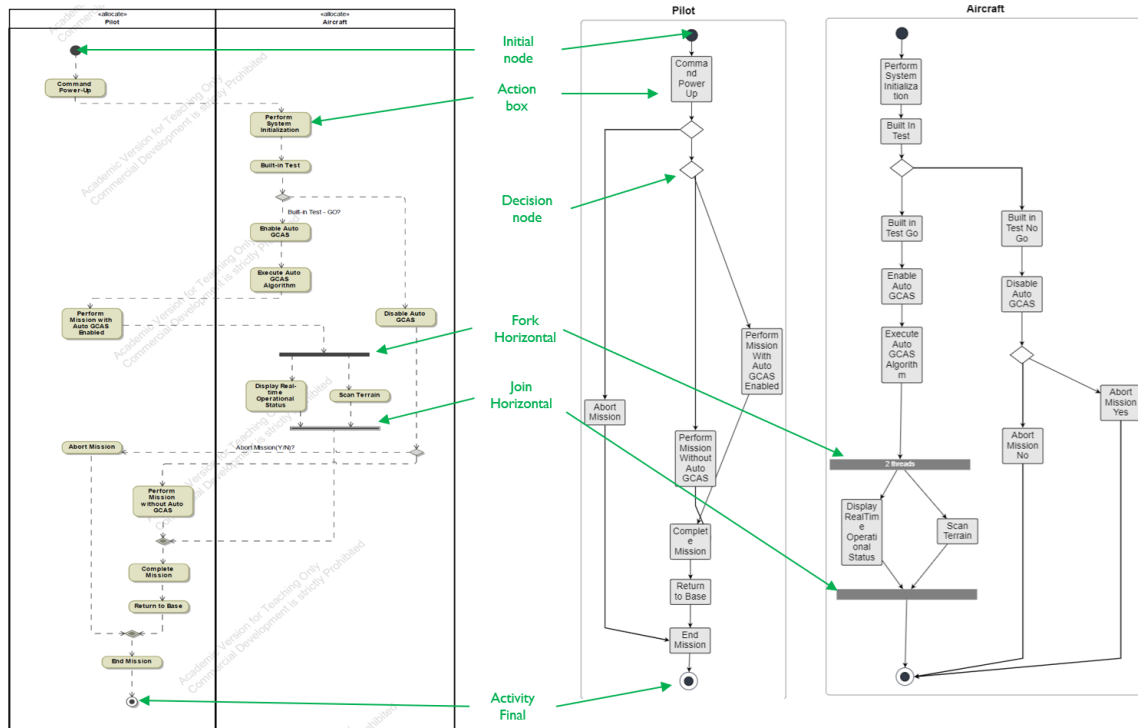


Figure 26. Notational concepts in common between SysML (left) and MP (right), illustrated using the Auto GCAS normal flight conditions model

With the initial node, end activity node, and decision node, both MP and SysML utilize the same notations and methodology to illustrate these notations on the activity diagrams. Both MP and SysML utilize the hollow diamond shape to illustrate the decision node. With the action box, both MP and SysML use the rectangular block to represent the action box notation and place the name of the activity within the box.

Also, both MP and SysML utilize similar mechanisms to split the activity flow using fork horizontal notation and merge the activities using join horizontal notation. For example, in Figure 26—SysML activity diagram, following the “Performing Mission with Auto GCAS Enabled” activity, the activity flow is split into two activities to illustrate that these two activities occurred concurrently at the same time. In the diagram, the “Display Real-time Operational Status” and “Scan Terrain” are split with the fork horizontal notation. Then, the activity stream is merged utilizing the join horizontal notation before the activity stream continues to the next activity—“Complete Mission.” Similar to SysML

activity diagram, MP also implements the fork and join mechanism when the activity stream is split into multiple sub-activities or merged back to the main activity stream.

b. Differences

Aside from the similarities, there are also multiple differences between MP and SysML activity diagrams including notational concepts such as the type of the arrows, merge node, fork horizontal, join horizontal, accept event action, send event action, the precedence relation between multiple lifelines/actors, and the implementation of condition loops. The biggest difference may be that while each SysML activity diagram in MSOSA is manually constructed, the activity diagrams in MP are automatically generated from the event grammar rules in the schema. This automated generation capability provides a means for verifying the schema's behavior logic across all possibilities. Since the MSOSA activity diagrams are manually derived, there is a high probability that possible activities may be overlooked.

Arrows types are differently specified between MP and SysML. With the activity diagram, MP utilizes only one type of arrow—the solid line arrow to connect each activity on the diagram. In contrast, SysML utilizes two types of arrows: the dashed line arrow and the solid line arrow. SysML utilizes the dashed line arrow to illustrate the control flow between activities while using the solid-line arrow to illustrate the object flow from one activity to another. In Figure 27, only the control flow arrow type is utilized in this SysML activity diagram.

Second, while SysML supports the merge node, which combines multiple alternate flows to a single outgoing flow, MP does not currently support this control node. In SysML, the Merge node is represented using a filled diamond-shape notation to differentiate from the hollow diamond-shape notation that represents the decision node. Figure 28 illustrates the differences between the merge node and decision node in SysML.

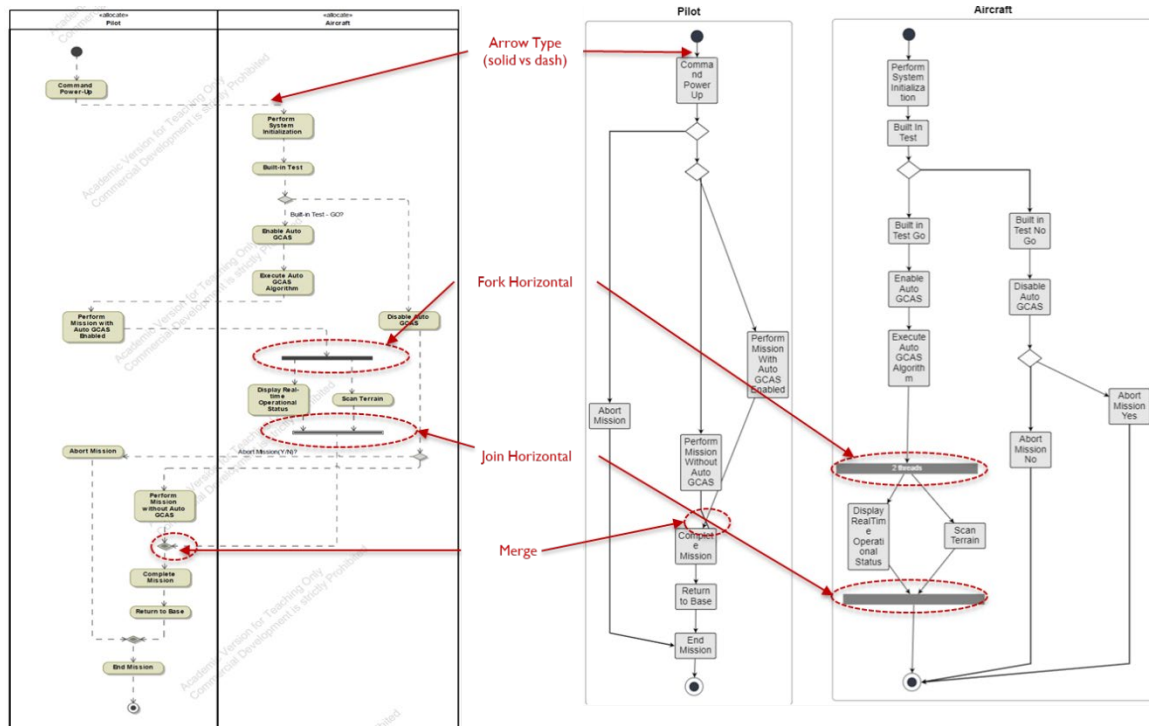


Figure 27. Differences in notational concepts between SysML (left) and MP (right), illustrated using the Auto GCAS normal flight conditions model

Third, although both MP and SysML support the fork horizontal and join horizontal, the graphical notations between MP and SysML are different. SysML utilizes a solid bar to illustrate the Fork node and the hollow bar to represent the merge node. In contrast, MP utilizes the same solid bar to illustrate both fork and merge nodes. However, with the fork node, MP displays the number of sub-threads. For example, in Figure 29, MP shows the total number of sub-threads under the fork node, which are “Display Real-time Operational Status” and “Scan Terrain.”

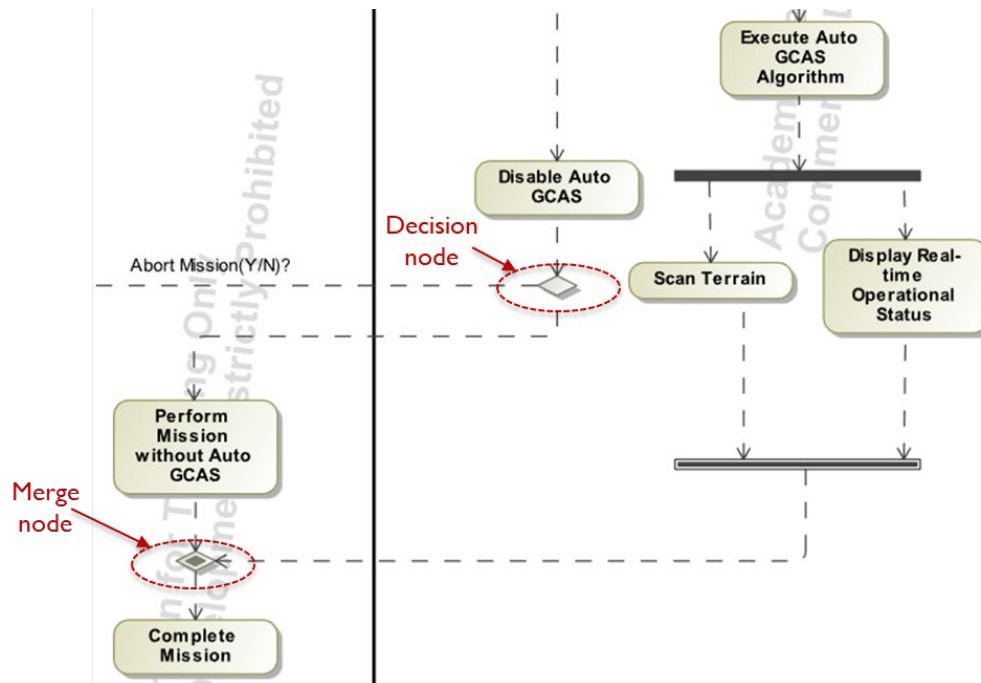


Figure 28. Differences between SysML merge node and decision node on SysML activity diagram

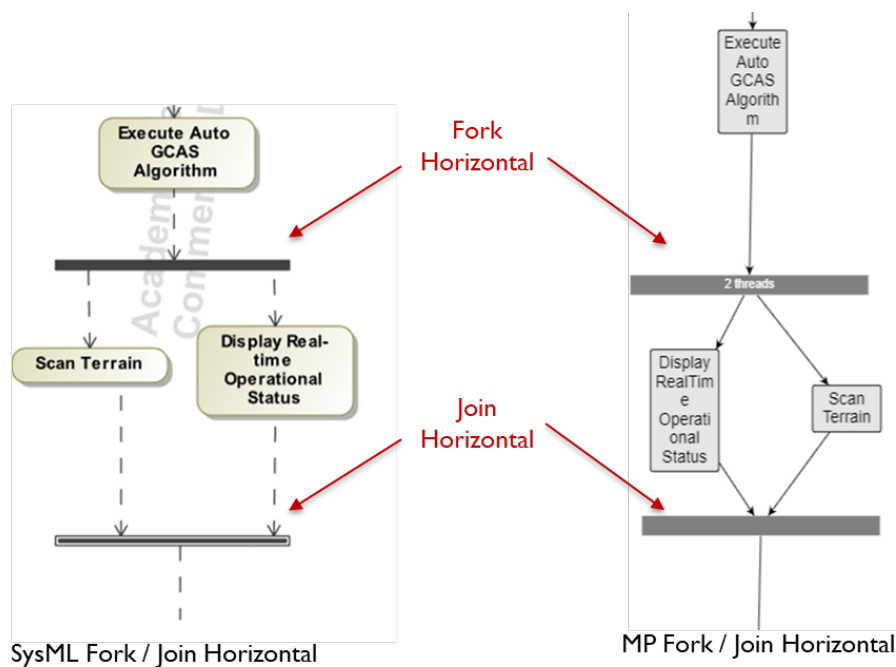


Figure 29. Differences in fork horizontal and join horizontal notational concepts between SysML (left) and MP (right) activity diagram

SysML activity diagrams also support additional notations beside the action box event, such as send signal action and accept event action. In Figure 30 (left), the SysML activity diagram shows the “Aircraft Status to Mission Control” as a send signal action and the “Pilot Inputs” as an accept event action. In contrast, MP does not graphically differentiate these event types. MP illustrates all action and event types utilizing the rectangular-shaped box.

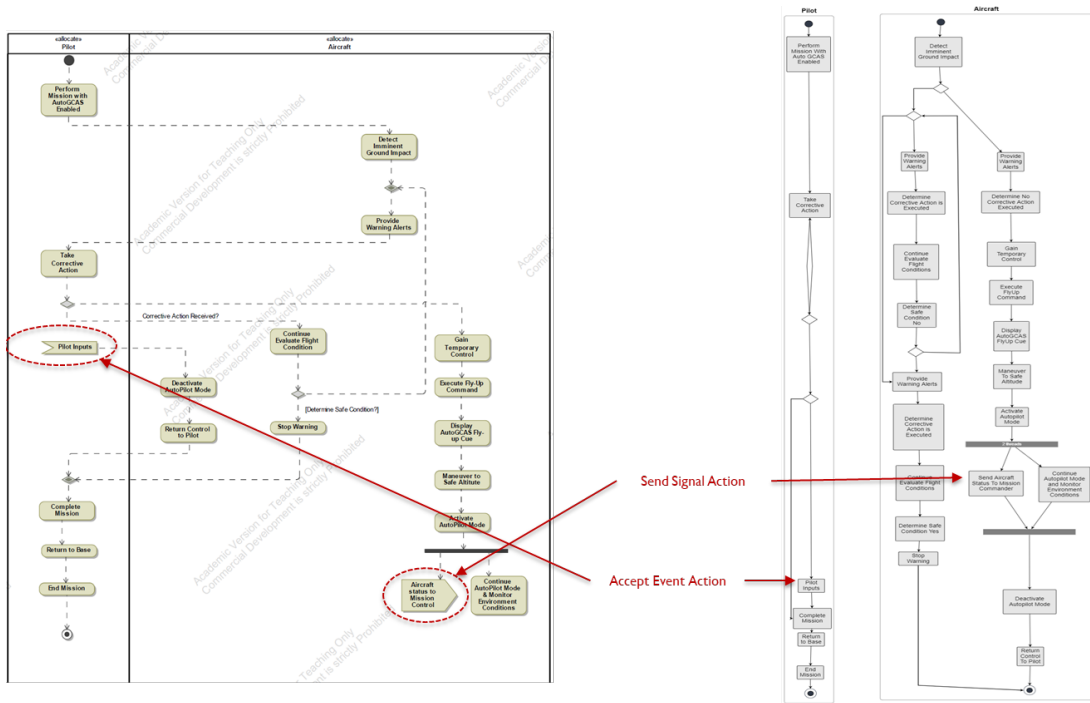


Figure 30. Differences in send signal action and accept event action notational concepts between SysML (left) and MP (right) activity diagram, illustrated using the Auto GCAS with impact condition model

Also, another major difference between MP and SysML is that MP tools currently do not support graphing the precedence relation created with coordinate statements between multiple lifelines/actors within the activity diagram. For example, in Figure 31, “Command Power Up” from the pilot is the preceding action for “Perform System Initialization” from the aircraft. Similarly, the pilot cannot “Perform Mission with Auto GCAS Enabled” before the aircraft performs the “Execute Auto GCAS Algorithm” action.

Since MP was initially designed for software architecture modeling and exhaustive testing of paths through software logic, the activity diagrams generated from MP are

straightforward visualizations of event grammar rules in the code and built from information in the schema itself without even needing to generate traces. Therefore, although MP tools do not currently graph the coordinated precedence relations between multiple actors to provide the complete integrated view, the MP language itself does contain the necessary data to generate this view. Since this is a tool difference and not a language capability shortcoming, the team will provide more discussion and recommendations in Chapter V — MP Activity Diagram.

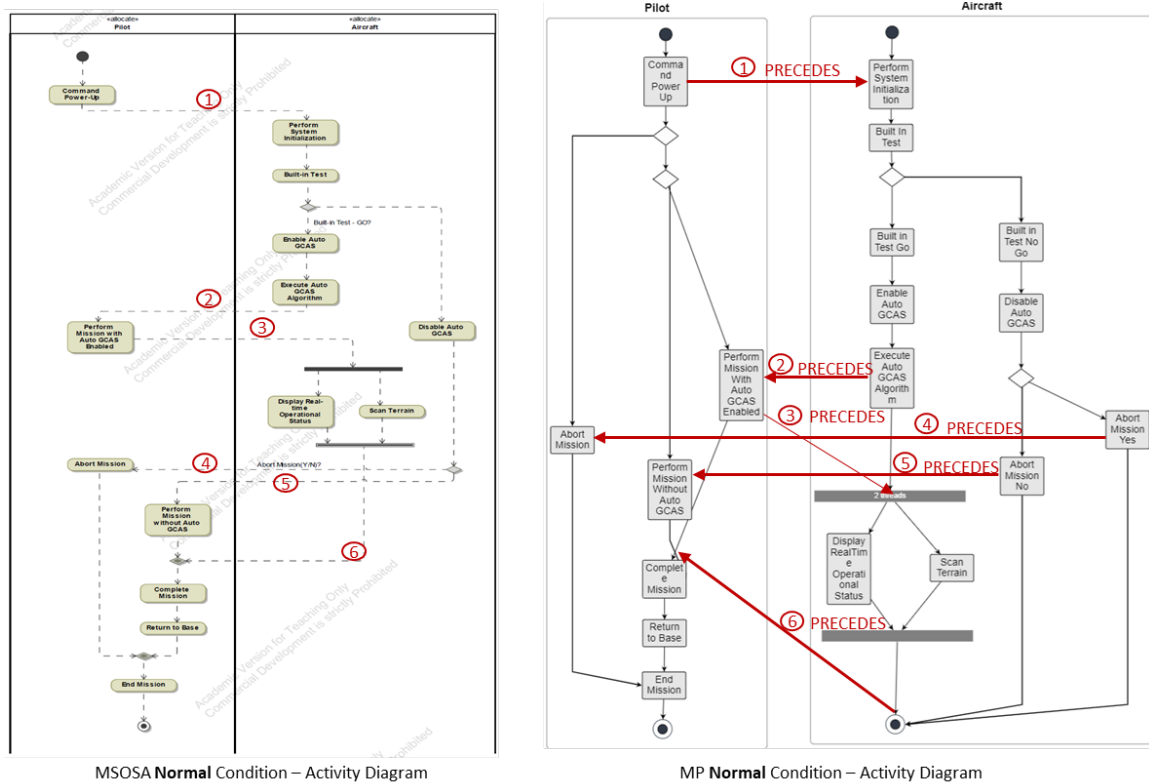


Figure 31. Differences in precedence relationship notational concepts between SysML (left) and MP (right) activity diagram, illustrated using the Auto GCAS normal flight conditions model

Another difference between MP and SysML is the mechanism where MP and SysML handle and illustrate the conditional action. Figure 32 illustrates this difference between MP and SysML. With SysML, the condition action (or guard) is built-in in the flow. For example, at the decision node, the “Built-in Test – Go?” guard is built-in and can be displayed above the arrow. When the result is Yes, the aircraft will “Enable Auto

GCAS” and when the result is “No,” the aircraft will “Disable the Auto GCAS.” Because MP is a simple event grammar with a single abstract concept of event with two basic relations of precedence and inclusion, it does not display the conditional action (or guard) as a separate concept as in SysML. MP uses the single event concept to model conditions such as “Built-In Test Go” or “Built-In Test No Go” as first-class events in the event stream of each branch in the activity diagrams.

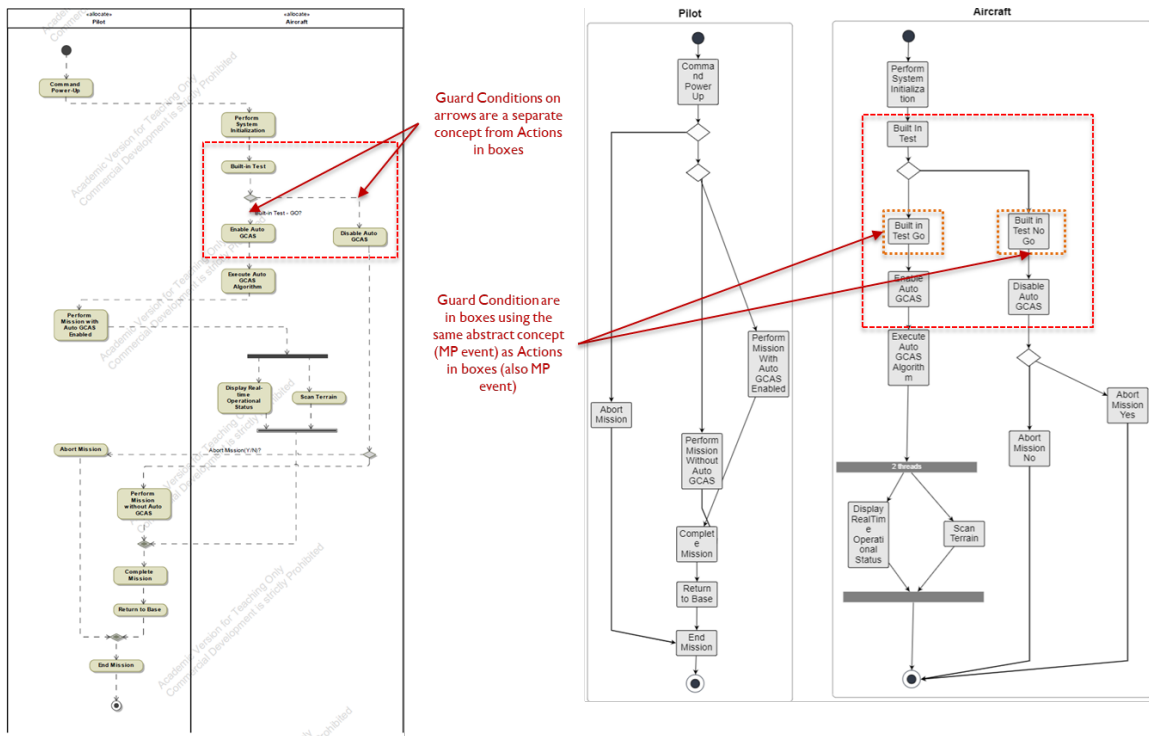


Figure 32. Differences in guard condition notational concepts between SysML (left) and MP (right) activity diagram

Lastly, MP and SysML implement different mechanisms to illustrate conditional activities in the diagram. With complex loops with two or more conditions, it appears that MP requires more overlapped activities to illustrate the conditions. For example, in Figure 33, with the SysML activity diagram, following route 2 – 4 – 5 when the aircraft “Detect Imminent Ground Impact,” it then “Provide Warning Alerts” and waits for corrective action feedback from the pilot and “Continue Evaluate Flight Condition.” If the aircraft determines the safe condition has not been met, then following path 5 -2-4, it will continue

to “Provide Warning Alerts.” However, if the safe condition is met, the aircraft will “Stop Warning” and follows the activity as in path 6. However, with the MP activity diagram, MP has to repeat paths 2–4 twice in order to illustrate the difference in paths 5 and 6. In Figure 33, MP repeats displaying “Provide Warning Alerts,” “Determine Corrective Action is Executed” and “Continue Evaluate Flight Condition” twice in order to illustrate the “Determine Safe Condition – No” and “Determine Safe Condition – Yes” events. As a result, with even more complex condition scenarios, MP will require more spaces and condition loops to illustrate the complete scope of the model.

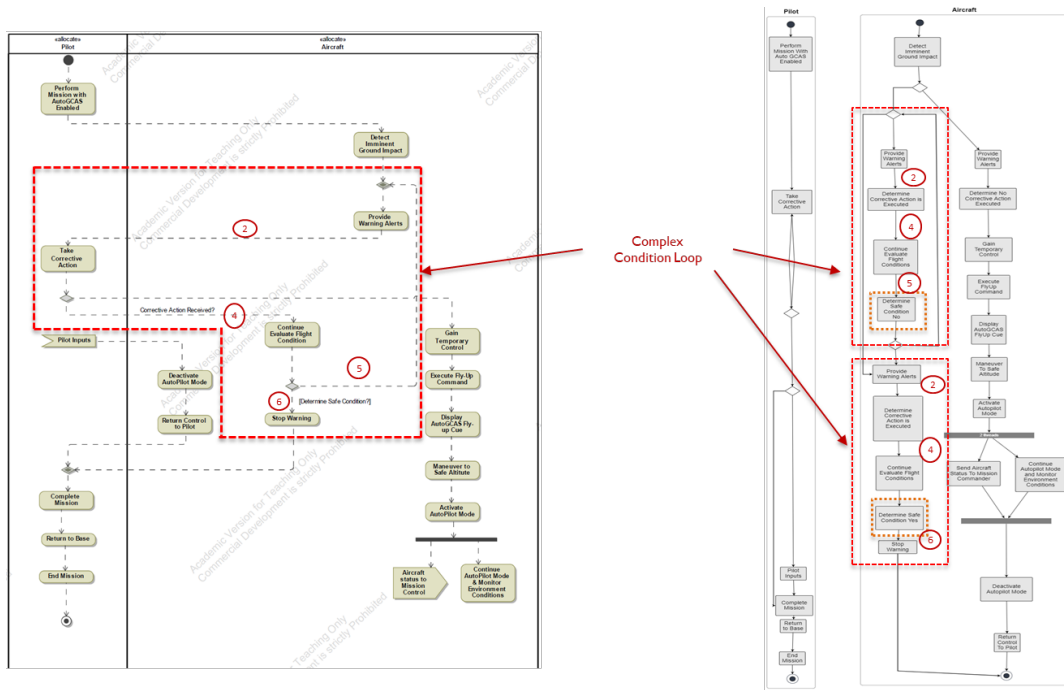


Figure 33. Differences in conditional loop notational concepts between SysML (left) and MP (right) activity diagram

With Dr. Giammarco’s recommendation, the team tried an alternate approach to simplify the MP conditional loop. Figure 34 illustrates the original (left) and simplified (right) MP condition loop designed based on the same model concept. With the simplified model, the “Conditions Safe” and “Conditions Unsafe” are designed as guard conditions. When the Aircraft determines “Conditions Safe” is valid, it then “Stop Warning.” However, when the “Condition Unsafe” remains valid, the Aircraft “Continues Warning” and then follows the “Condition Unsafe” lower-level Guard conditional loop. In this loop,

the Aircraft continues the activity sequences which include “Provide Warning Alerts,” “Determine Corrective Action is Executed,” and “Continue Evaluate Flight Conditions” until “Conditions Safe” is determined. The simplified diagram proves that with different approaches and designs, MP can generate activity models that satisfy the overall purposes and needs.

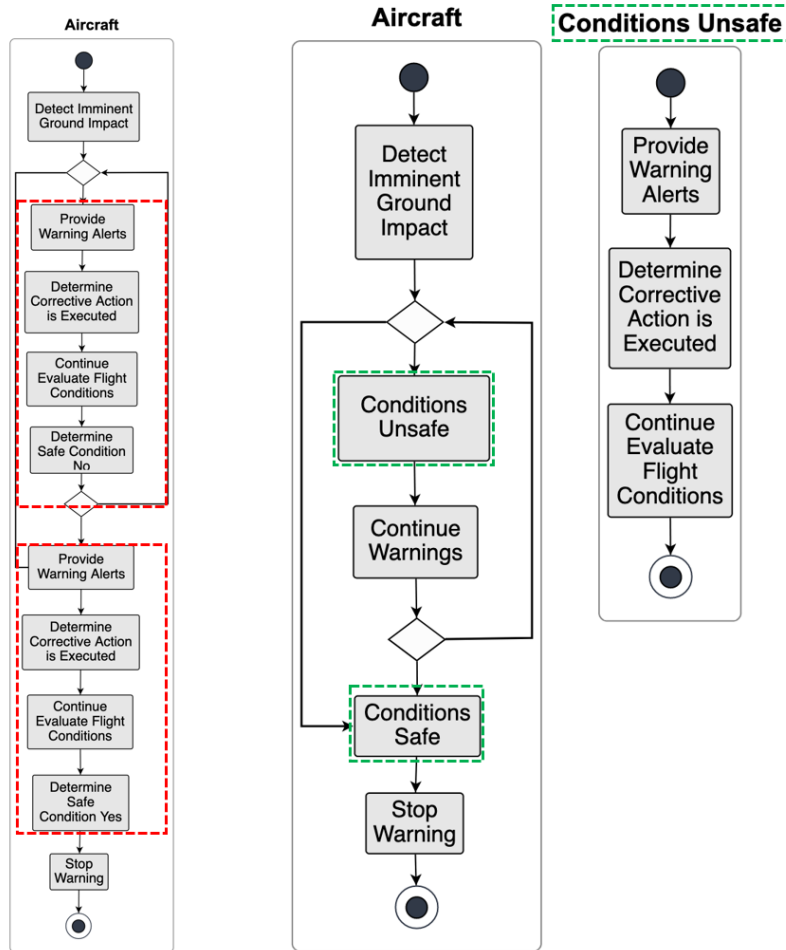


Figure 34. Proposed simplified conditional loop between MP original (left) and simplified (right) activity diagram, illustrated using the Auto GCAS impact flight conditions model

3. MP State Machine vs. SysML State Machine Diagram

To investigate the similarities and differences between MP and SysML state machine diagrams, the Model Wreckers developed the state machine diagrams using the same Auto GCAS operational scenarios. Given the team’s limited experience with MP

modeling and the advanced nature of creating State diagrams in MP, the team chose to focus on developing a standalone schema that generates the State diagram without attention to detail on optimizing the resulting event traces and activity diagrams for a fully integrated model. The MP schema specifically developed to generate the state machine diagram is listed in Appendix A — *AutoGCAS_State_Machine_Diagram_V1_3*.

Figure 35 illustrates the SysML state machine diagram and Figure 36 illustrates the corresponding MP state machine diagram. Both MP and SysML state machine diagrams are developed based on the same Auto GCAS operational scenario. Initially, the aircraft is in a power-off state. Upon the pilot commanding the *Power Up*, the aircraft is powered on and starts the system initialization process. The aircraft changes the state to *Start-up Self-Test*. If the test result is *Go*, the state then changes to *Auto GCAS Enabled*. However, if the self-test result is *No-Go*, the aircraft changes the state to *Auto GCAS Disabled*. With *Auto GCAS Disabled*, the aircraft state can either shift to *Aircraft Power Off* when *Abort Mission* is decided or change to *Fly without Auto GCAS* if the pilot decides to *Continue Mission*. Once the aircraft *Return to Base*, its state change to *Weight-on-Wheel*, and upon receiving the *Power Off* command, its states shift to *Aircraft Power Off*.

When the aircraft state is *Auto GCAS Enabled*, once take-off occurs, it changes the state to *Auto GCAS Algorithm Running*. During the mission, if the aircraft detects an imminent impact, its state change to *Trigger Warning Alerts*. When corrective action is received and safe condition is met, the aircraft changes state to *Stop Warning Alert*. However, when corrective action is not received, the aircraft changes state to *Auto GCAS Gain Temporary Control*, followed by *Auto GCAS Fly-up Activated*. When the safe condition is reached, the aircraft change state to *Auto Pilot Activated*, and only when pilot input is detected, the aircraft change state to *Autopilot Mode Deactivated*. Then, upon the pilot completes the mission, the aircraft returns to base, changes its state to *Weight-on-Wheel*, and then *Aircraft Power Off* when the power-off command is received.

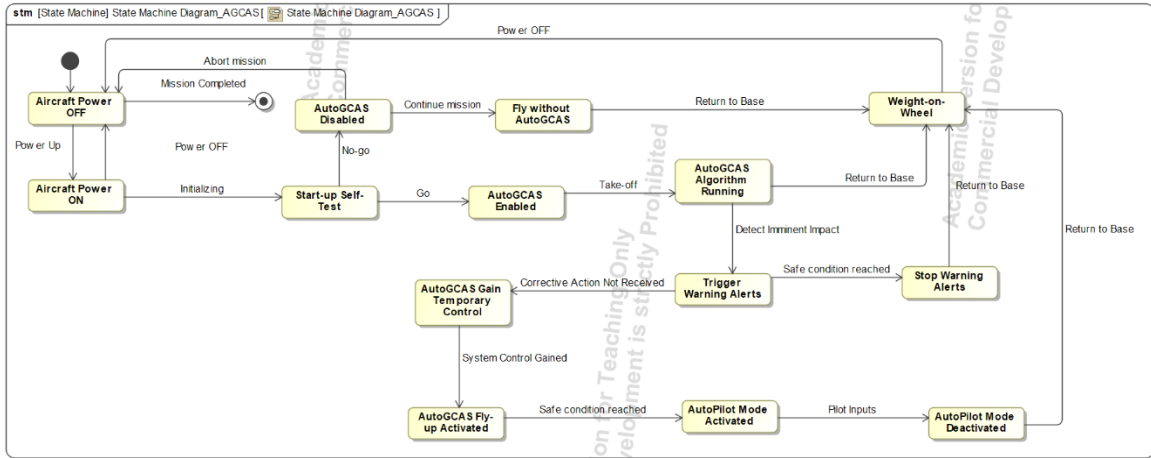


Figure 35. SysML state machine diagram, illustrated using the Auto GCAS model

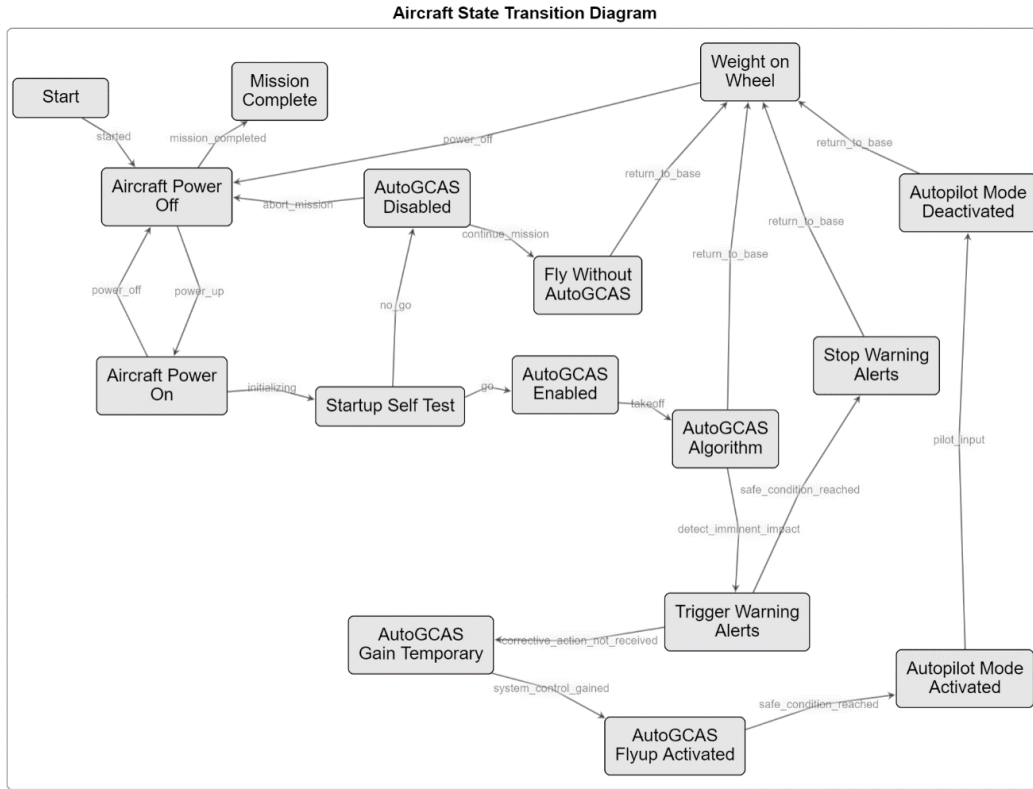


Figure 36. MP state machine diagram, illustrated using the Auto GCAS model

a. Similarities

The SysML state machine diagrams in Figure 35 and MP state machine in Figure 36 show that both SysML and MP have similar state machine diagram notation when using the basic state diagram concepts (states and transitions). The state block, transition arrows, and legends are similar between MP and SysML.

b. Differences

Besides the similarities, there are some notational differences between MP and SysML state machine diagrams. Figure 37 illustrates the side-by-side comparison between MP and SysML. The most noticeable difference is in the representations of initial and final state nodes. While SysML provides the initial node and final state node as a solid circle and solid circle within a circle respectively, MP utilizes the same rectangular box for all of the states, treating initial and final states as any other state in its simple event grammar using the same abstract concept of “event.”

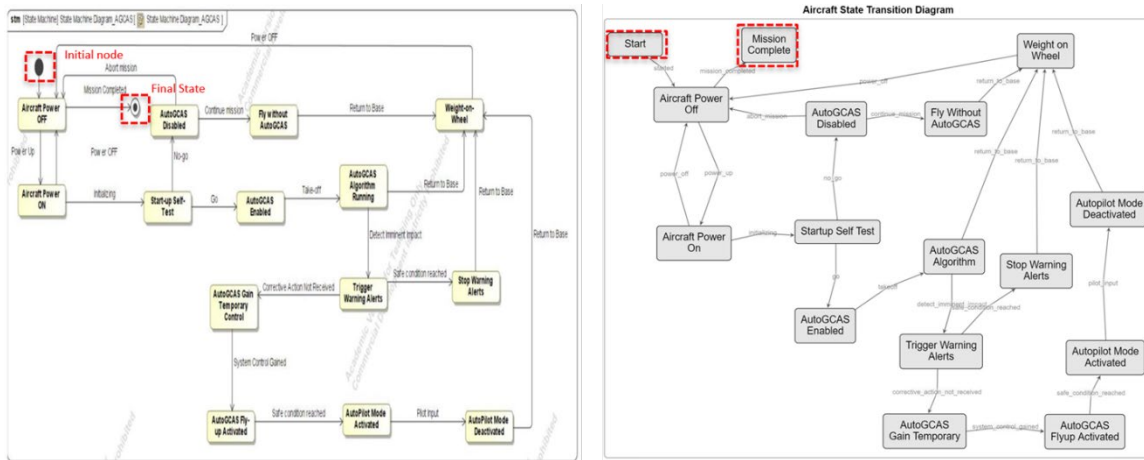


Figure 37. Differences in notational concepts between SysML (left) vs. MP (right) state machine diagram

Additionally, while SysML allows the creation of nested state diagrams using composite state notations, this notation is not currently available in MP. Figure 38 illustrates an example of a composite state within a diagram. Even though the composite state notation is not currently supported in a single MP schema, separate MP schemas can

be used to generate separate state machine diagrams that depict the full scope of the model, as described in the next section.

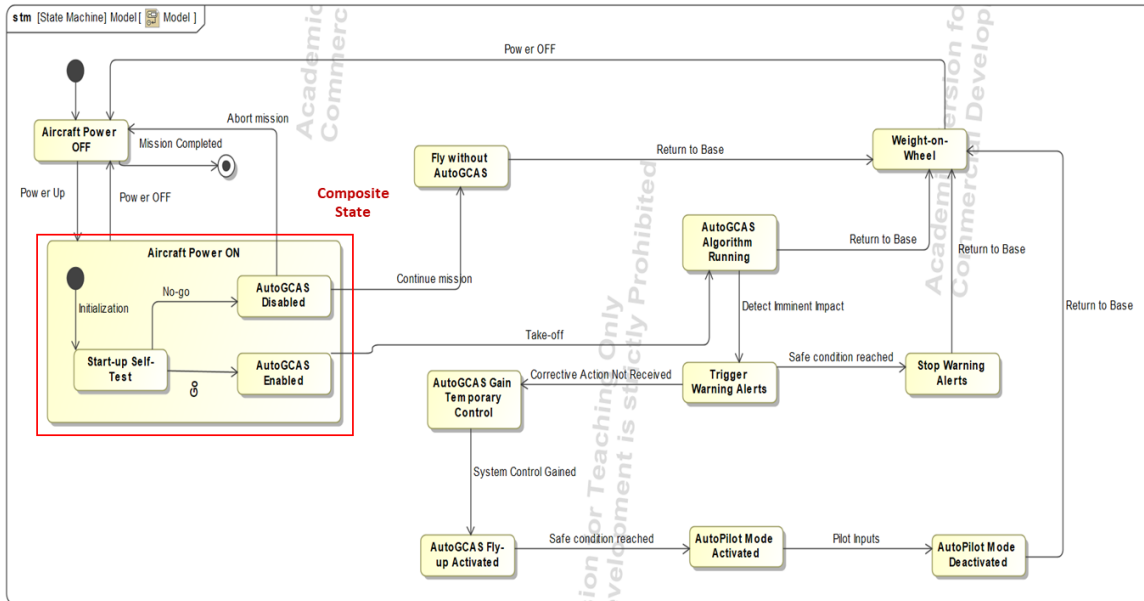


Figure 38. SysML state machine diagram with composite state notational concept, illustrated using the Auto GCAS model

B. OBSERVATIONS ON OPTIMIZING MODEL STRUCTURE

This section covers the Firearm Safety Model case study. This case study shows the proper behavior between the weapon operator and the weapon itself. The weapon changes states by the operator providing input to the weapon, and this behavior is captured in the Firearm Safety Model by the weapon experiencing states and the operator providing state transitions.

This case study was created to address the challenges associated with the state machine for the Auto GCAS System in Monterey Phoenix. The MP Auto GCAS state machine was identical to the Auto GCAS state machine in MSOSA. However, the seven event diagrams that created the MP Auto GCAS state machine contained an overdeveloped aircraft root event and an underdeveloped pilot root event. This issue can be seen in Figure 39 with the second Auto GCAS event trace diagram. The aircraft root event contains all the events and transitions for the Auto GCAS state machine. However, the pilot root event

only contains a minimal set of states that affect some of the transitions in the aircraft root event.

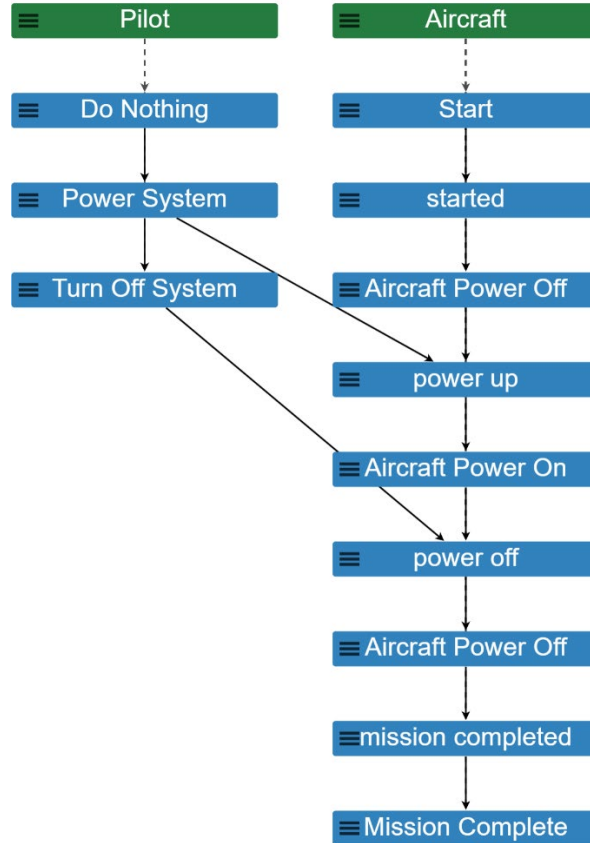


Figure 39. MP event trace diagram number 2 run at scope 1, illustrated using the Auto GCAS state machine model

This preliminary Auto GCAS state machine method is an easy way to create complex single-system state machines in Monterey Phoenix because it reduces coding complexity, but the method is inaccurate for a system of systems model perspectives because the pilot root is underdeveloped.

1. Compact Event Trace Structure Method

In order to create a Firearm Safety Model state machine with realistic event traces, the Model Wreckers team used a compact event trace structure. This event trace structure places the states under the weapon root event and the transitions under the operator root

event. This can be seen in Figure 40. The events with all capital letters are the start/end states. The events that are present tense verbs with one capitalized letter are the weapon states, and the lower case and past tense events are the transitions. The weapon root event always contains the states for the state machine, and the operator root event always contains the transitions for the state machine. In the figure, a weapon state precedes an operator transition, and the transition then precedes the next weapon state. This zig-zag pattern from the start state to the end state is the compact event trace structure that the Model Wreckers created for the Firearm Safety Model.

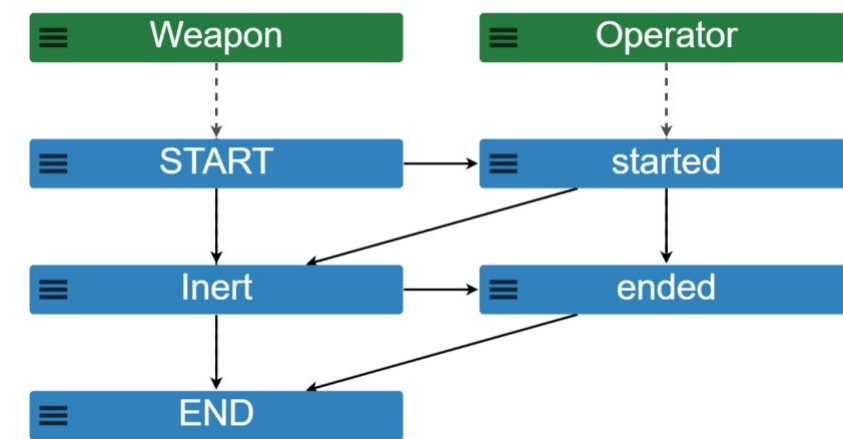


Figure 40. Monterey Phoenix event trace diagram number 1 run at scope 1, illustrated using the Firearm Safety Top-level model

Initially, the team attempted to create an MP model of the complete Firearm Safety Model as shown in Figure 17. However, the model complexity was too high for the team during the time of model creation. To simplify the original Firearm Safety Model while preserving all states and transitions, the team decided to partition the original model into two separate Monterey Phoenix models. These two models are described in the next section.

2. MP State Machine Diagram Comparisons

In order to simplify the firearm safety model in Figure 17, the lower half of the firearm safety model was encapsulated into a subsystem called the *Operational* state. The process of encapsulation allows system designers to place functionality within a subsystem

boundary with inputs and outputs. The functions within this boundary can be hidden from the user of the system like a black box (Delligati 2014). This encapsulation step allowed the team to divide the complicated Firearm Safety Model into two simple models.

The first Firearm Safety Model is the Top-Level model. The Microsoft Visio representation of this model is shown in Figure 41. This figure shows all the states and transitions for the upper half of the original Firearm Safety Model along with the encapsulated state named *Operational*. The Top-Level model and Operational model are captured in separate Monterey Phoenix schemas.

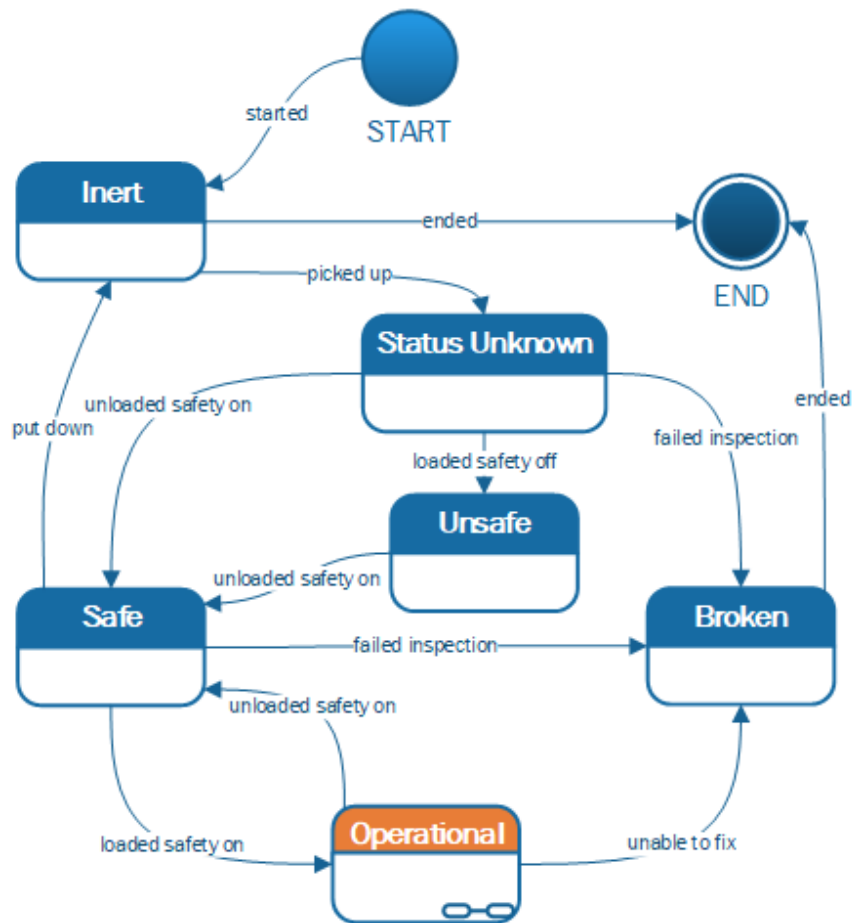


Figure 41. Firearm Safety Top-level model, illustrated using Microsoft Visio

Figure 41 shows the encapsulated state called *Operational* along with the states in the upper half of the original Firearm Safety Model. However, Figure 42 shows the internal

states of the operational subsystem. These states are in the lower half of the original Firearm Safety Model before the encapsulation process. The input in the operational block is the *loaded safety on* transition, and the outputs for the operational block are *unable to fix* and *unloaded safety on* transitions.

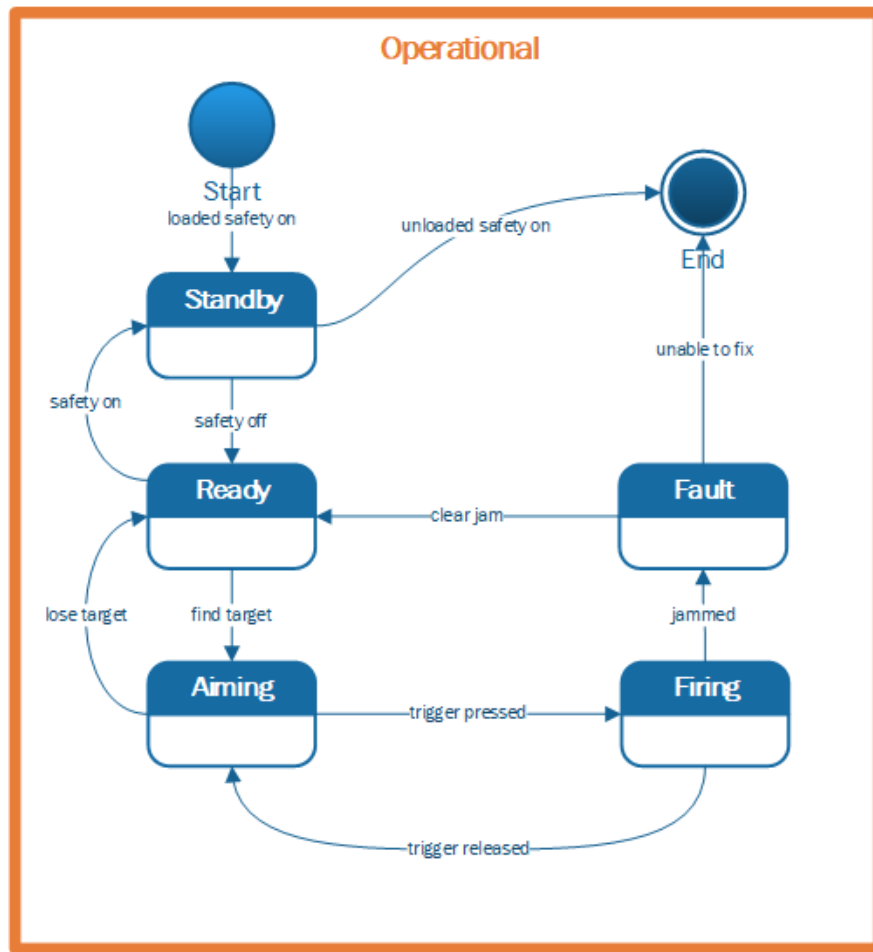


Figure 42. Drill down of the Firearm Safety operational state diagram from the Firearm Safety Top-level model, illustrated using Microsoft Visio

Figure 43 shows the MP model for the Firearm Safety Top-Level state machine. The state machine was built from the ten separate event traces shown in Table 5. All ten event traces travel through the different sections of the state machine so that the ten event traces inform the generation of the state machine diagram. The event traces in the table follow the event and transition naming conventions earlier defined.

Event trace 1, in Table 5, shows how the operator changes the states in the weapon. The weapon begins in the *START* state, and the operator initiates the *started* transition. The started transition changes the weapon state to the *Inert* state. After the weapon is in the *Ready* state, the operator initiates the *ended* transition. This changes the weapon's state to *END*. This is the simplest event trace for the Firearm Safety Top-Level model. This event trace is where the operator does not interact with the weapon, but it shows the concept of the user performing actions that change the state of the weapon. All ten event traces for the Top-Level Firearm Safety state machine follow the compact event trace structure. Event trace 3 for the Firearm Safety Top-Level Model is shown in Figure 44 as a second example from Table 5. Event trace 1 and 3 are both highlighted in the table.

Table 5. MP main state machine event traces

Event Trace Number	Event Trace
1	START, started, Inert, ended, END
2	START, started, Inert, picked up, Status Unknown, failed inspection, Broken, ended, END
3	START, started, Inert, picked up, Status Unknown, unloaded safety on, Safe, put down, Inert, ended, END
4	START, started, Inert, picked up, Status Unknown, unloaded safety on, Safe, loaded safety on, Operational, unable to fix, Broken, ended, END
5	START, started, Inert, picked up, Status Unknown, unloaded safety on, Safe, loaded safety on, Operational, unloaded safety on, Safe, put down, Inert, ended, END
6	START, started, Inert, picked up, Status Unknown, unloaded safety on, Safe, loaded safety on, Operational, unloaded safety on, Safe, failed inspection, Broken, ended, END
7	START, started, Inert, picked up, Status Unknown, unloaded safety on, Safe, loaded safety on, Operational, unloaded safety on, Safe, loaded safety on, Operational, unable to fix, Broken, ended, END
8	START, started, Inert, picked up, Status Unknown, loaded safety off, Unsafe, unloaded safety on, Safe, put down, Inert, ended, END
9	START, started, Inert, picked up, Status Unknown, loaded safety off, Unsafe, unloaded safety on, Safe, failed inspection, Broken, ended, END
10	START, started, Inert, picked up, Status Unknown, loaded safety off, Unsafe, unloaded safety on, Safe, loaded safety on, Operational, unable to fix, Broken, ended, END

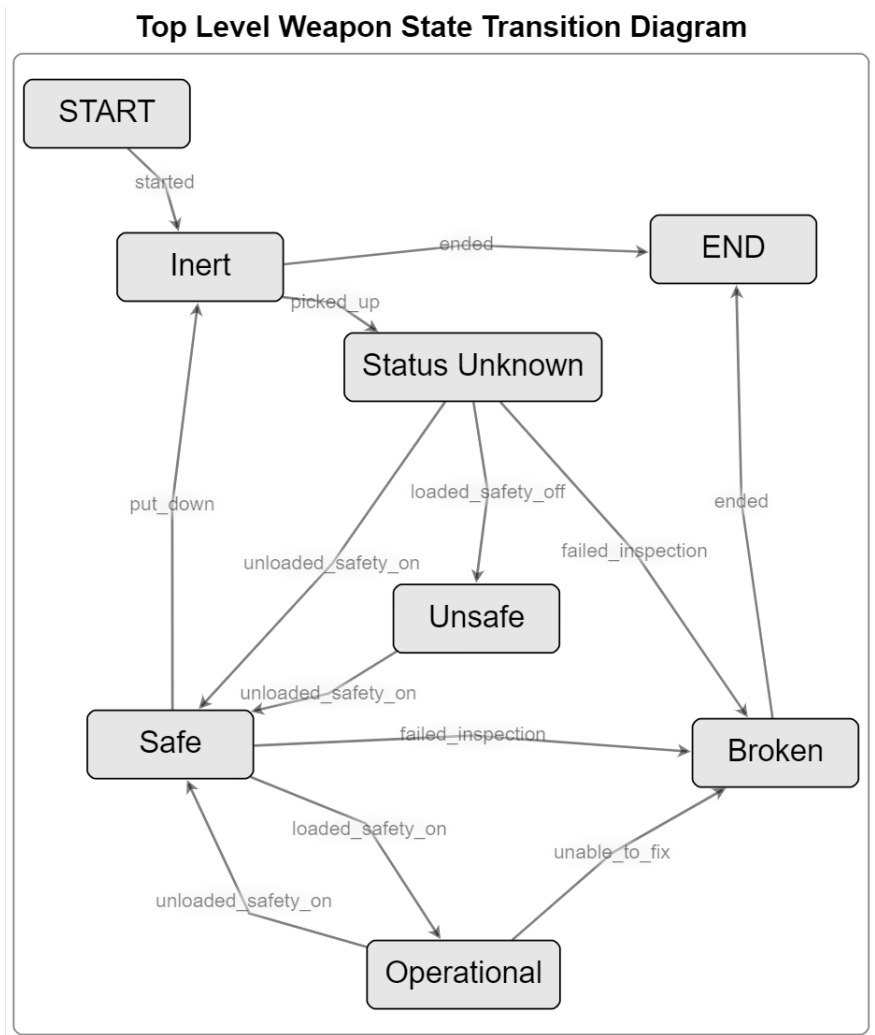


Figure 43. Monterey Phoenix state machine diagram, illustrated using the Firearm Safety Top-level model

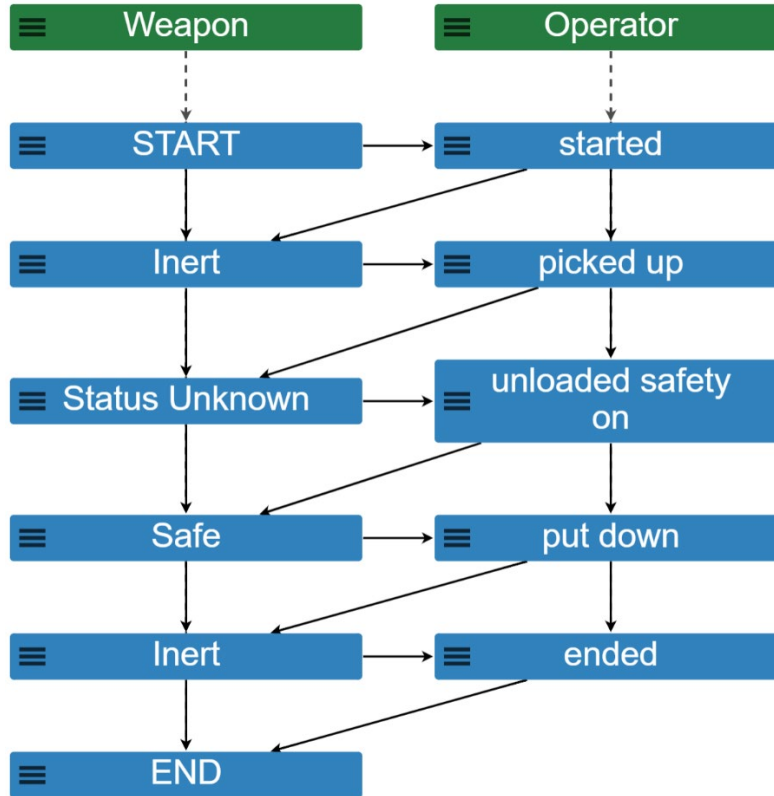


Figure 44. MP event trace diagram number 3 run at scope 1, illustrated using the Firearm Safety Top-level model

Figure 45 is the state machine for the Firearm Safety Operational state machine. This is the encapsulated operational subsystem in the Top-Level state machine. The Operational state machine is much less complicated than the Top-Level state machine in Figure 43 because it only requires four event trace diagrams for state machine generation. The four-event trace diagrams are shown in Table 6. The four-event traces use the compact event trace structure where the user transitions change the state of the firearm. Also, the MP code for the operational model is more straightforward. This is due to the team gaining MP coding experience over time.

Operational State Transition Diagram

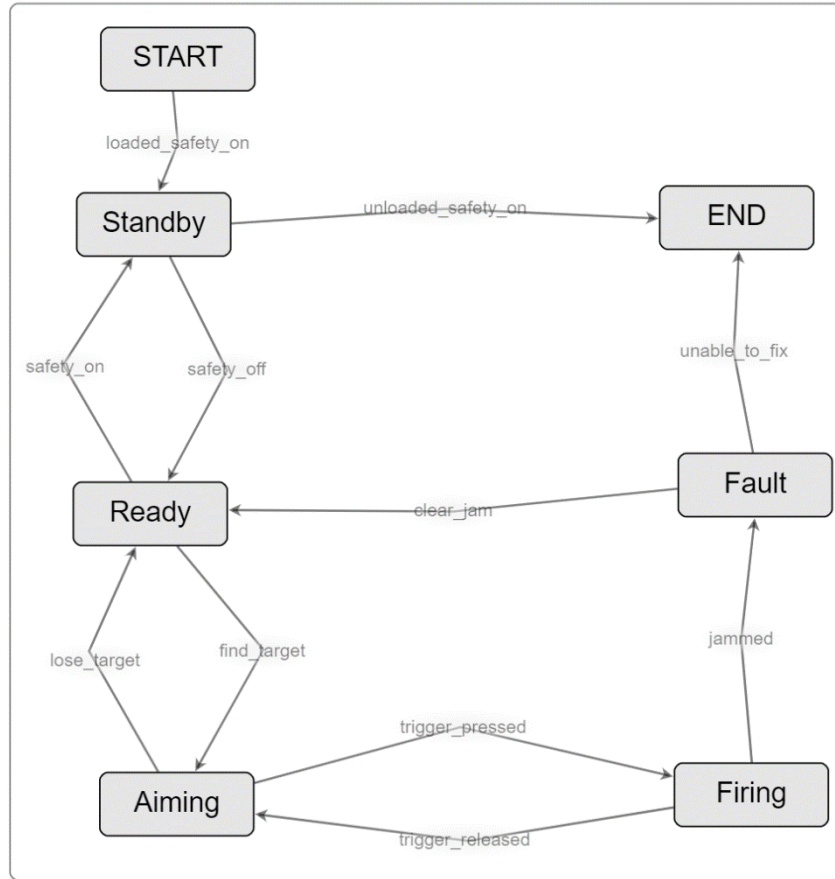


Figure 45. Monterey Phoenix state machine diagram, illustrated using the Firearm Safety Operational model

Table 6. MP operational state machine event traces

Event Trace Number	Event Trace
1	START, loaded safety on, Standby, unloaded safety on, END
2	START, loaded safety on, Standby, safety off, Ready, find target, Aiming, trigger pressed, Firing, jammed, Fault, clear jam, Ready, safety on, Standby, unloaded safety on, END
3	START, loaded safety on, Standby, safety off, Ready, find target, Aiming, trigger pressed, Firing, trigger released, Aiming, lose target, Ready, safety on, Standby, unloaded safety on, END
4	START, loaded safety on, Standby, safety off, Ready, find target, Aiming, trigger pressed, Firing, jammed, Fault, unable to fix, END

C. CHAPTER SUMMARY

This chapter covered two separate analysis topics related to SysML and Monterey Phoenix (MP). The first section focused on the similarities and differences between SysML and MP. The model that was used to highlight the distinctions between the modeling language was the Auto GCAS model. In this section, SysML was compared with MP in three ways. The first comparison was between the SysML sequence diagram and the MP event trace diagram. The second comparison was between the activity diagrams between SysML and MP. And, finally, the third comparison was between the SysML and MP state machines.

The second section in this chapter focused on the modeling and coding challenges associated with creating the Auto GCAS state machine. Because the Auto GCAS state machine had an overdeveloped aircraft root event and an underdeveloped pilot root event, the team decided to create the compact event trace structure to create state machines where both the user root event and system root event were equally developed. The Firearm Safety model was created to demonstrate the compact event trace structure method where the weapon root event contained the states of the weapon and the operator root event contained the transitions caused by the operator. However, due to the team's model inexperience, the original model was partitioned into a Top-Level model and an Operational model. This way, both models were easier to program in MP.

The next section will discuss the conclusions of the research in this capstone paper along with recommendations for SysML and Monterey Phoenix.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS

This research has produced two important contributions to the systems engineering community. First, the Model Wreckers have proposed modifications to MP diagrams that will make MP more useful to the SysML community. The second contribution is that the two Firearm Safety Models are the first known complex models where all three diagrams (event trace, activity, and state machine) are accurately generated from a central schema.

Due to a relatively shallow learning curve, most of the systems engineering community has accepted and adopted the SysML behavior diagrams and the system modeling language as a whole, in its efforts to better understand system requirements and early architecture. The power that resides in MP's ability to create multiple behavior diagrams from a single model of event relationships is not being used to its full potential. The Model Wreckers believe that MP has not gained wider acceptance in the systems engineering community due to non-SysML graphical outputs and a steeper upfront learning curve to building the necessary model through defined relationships. We have shown that the MP diagrams can tell the same story as a SysML equivalent diagram. The issue remains that modeling in MP (tapping into its power of scope-complete event trace generation) requires a lot of guided study along with trial and error when compared to SysML modeling.

As the team initially set out to do, the differences in graphical outputs have been identified. The team has also laid out the first course of blocks on a future path that could eventually see an integration of MP into various SysML modeling tools. Presented below are the Model Wrecker's conclusions, recommendations, and ideas for future work to keep laying the blocks along the path to better understanding system behaviors.

A. CONCLUSION

This section presents a summary and conclusions that address the first research question in Chapter I: "1. What are the differences that impede the consistent interpretation of behaviors expressed in MP v4.0 and SysML v1.6 behavior diagrams?" In the following subsections, the team provides summary comparisons between the MP event trace vs. the

SysML sequence diagram, followed by MP vs. SysML activity diagram, and lastly the MP vs. SysML state machine diagram. Then, the team discusses noteworthy differences between MP-Firebird and SysML MSOSA tools based on the team’s experiences and concludes the section with several observations on the development of a state machine diagram in MP.

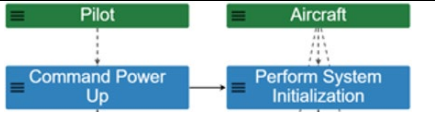
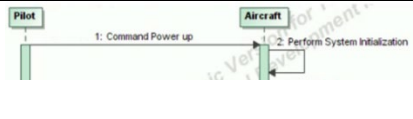
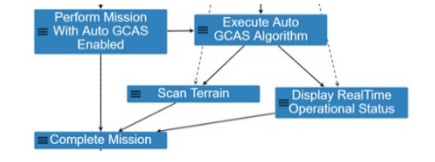
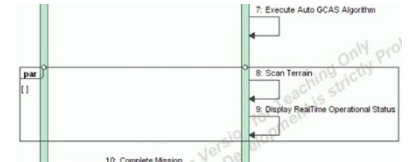
1. MP Event Trace vs. SysML Sequence Diagrams

Based on this study, the Model Wreckers observe the similarity that between MP event traces and SysML sequence diagrams both diagrams can illustrate operational events and sequences and deliver the same stories to the readers.

Aside from this similarity, there are multiple differences including the diagram scopes and notational concepts. Table 7 provides a summary of the differences between MP event traces and SysML sequence diagrams.

Table 7. Differences between MP event trace and SysML sequence diagrams

Description	MP Event Trace Diagram	SysML Sequence Diagram
Diagram Scope	<ul style="list-style-type: none"> – Used to illustrate exactly one single path through the behavior logic in the MP schema – Each diagram is part of an exhaustive set of traces automatically generated for a given iteration scope 	<ul style="list-style-type: none"> – Used to illustrate single or multiple paths through the behavior logic in one diagram – Diagrams are not part of an exhaustive set of traces automatically generated for a given iteration scope
Event Trace / Sequence Notational Concept	<ul style="list-style-type: none"> – Utilizes rectangular blocks to illustrate events – Places the event’s name inside the rectangular blocks 	<ul style="list-style-type: none"> – Utilizes arrows to illustrate events – Places the event’s name above the arrows

Description	MP Event Trace Diagram	SysML Sequence Diagram
		
<p>Parallel Event Notational Concept</p>	<ul style="list-style-type: none"> – Illustrated by branching from the parent event to sub-level events – Parallel event notations can be freely arranged on the diagram 	<ul style="list-style-type: none"> – Illustrated by using the parallel (Par) notational concept – Parallel events are stacked vertically on the lifeline 
<p>Type of Arrows</p>	<ul style="list-style-type: none"> – Specifies 3 arrow types: <ul style="list-style-type: none"> ○ Unlabeled solid line arrow: precedence relationship ○ Unlabeled dashed arrow: inclusion relationship ○ Labeled solid line arrow: user-defined relationship 	<ul style="list-style-type: none"> – Specifies multiple arrow types such as: <ul style="list-style-type: none"> ○ Solid line arrow with solid head: synchronous message ○ Solid line arrow with hollow head: asynchronous message ○ Dashed arrow with hollow head: reply message ○ Solid arrow with a head/ tail point to the same lifeline: “Message to Self” message

2. MP vs. SysML Activity Diagram

The study enumerates multiple similarities and differences in the implementation and notational concepts between the MP and SysML activity diagrams. Table 8 shows a summary of similarities and Table 9 shows the comparison of the differences between MP and SysML activity diagrams.

Table 8. Similarities between MP and SysML activity diagrams

Description	MP Activity Diagram	SysML Activity Diagram
Notational Concepts	<ul style="list-style-type: none"> – Both MP and SysML support identical notational concepts: <ul style="list-style-type: none"> ○ Initial node ○ Action block ○ Decision node ○ Final activity 	
Mechanism to Fork Activity Flow	<ul style="list-style-type: none"> – Both MP and SysML utilize similar mechanisms to fork the activity flow into multiple concurrent flows with the fork horizontal notional concept 	
Mechanism to Join Activity Flow	<ul style="list-style-type: none"> – Both MP and SysML utilize similar mechanisms to join multiple concurrent flows into the main activity stream with the join horizontal notional concept 	
Sequence of Activities	<ul style="list-style-type: none"> – Without considering the Precedence relations between swimlanes, the sequence of activities is depicted similarly between SysML and MP- generated activity diagrams using directional arrows. 	

Table 9. Differences between MP and SysML activity diagrams

Description	MP Activity Diagram	SysML Activity Diagram
Precedence relations between swimlanes	<ul style="list-style-type: none"> – Currently does NOT produce swimlane-style activity diagram showing precedence relations between events in different actors 	<ul style="list-style-type: none"> – Swimlane-style activity diagrams show precedence relations between actors' events for complete integrated views
Type of arrows	<ul style="list-style-type: none"> – Specifies only one solid line arrow for control flows in activity diagrams 	<ul style="list-style-type: none"> – Specifies two arrow types for activity diagrams: <ul style="list-style-type: none"> ○ Solid line arrow: illustrate object flow ○ Dashed arrow: illustrate control flow
Merge node	<ul style="list-style-type: none"> – Not currently graphed on the activity diagram 	<ul style="list-style-type: none"> – Employed as part of the activity diagrams in SysML
Fork Horizontal & Join Horizontal	<ul style="list-style-type: none"> – No graphical differences between a fork and a join node in MP 	<ul style="list-style-type: none"> – Fork and join horizontal are graphically distinct in SysML (MSOSA)
Guard condition	<ul style="list-style-type: none"> – Modeled as an MP event block at the head of an event stream of each branch in the activity diagram 	<ul style="list-style-type: none"> – Guard condition is a distinct concept to control branch selection with its name displayed above the arrow
Accept Event Action & Send Event Action	<ul style="list-style-type: none"> – Modeled as MP event blocks, but not graphically distinct from other MP events 	<ul style="list-style-type: none"> – Graphically supported in SysML to distinguish between action event types

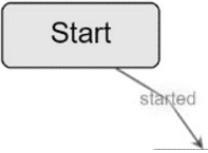
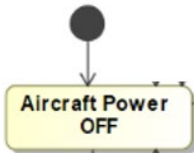
3. MP vs. SysML State Machine Diagram

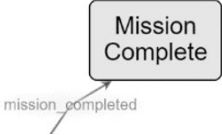
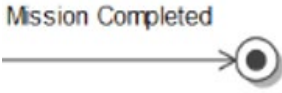
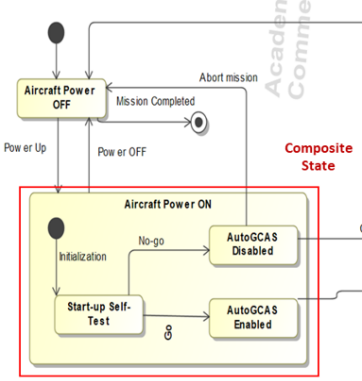
Among all behavior diagrams, the state machine diagrams have the most similarities between MP and SysML. Especially when designing with a simple state machine layout, MP can automatically generate state machine diagrams that are almost graphically identical to SysML. From this study, Table 10 shows all the similarities, and Table 11 shows all the differences between MP and SysML state machine diagrams.

Table 10. Similarities between MP and SysML state machine diagrams

Description	MP State Machine	SysML State Machine
Similarities	<ul style="list-style-type: none"> – Both MP and SysML support similar notational concepts when designing with a basic and simple layout – Similar notational concepts include: <ul style="list-style-type: none"> ○ State block ○ Transition arrows ○ Names for state and transition arrows 	

Table 11. Differences between MP and SysML state machine diagrams

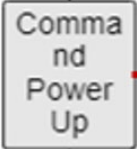
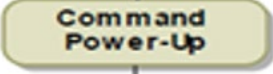
Description	MP State Machine	SysML State Machine
Initial node	<ul style="list-style-type: none"> – Currently implemented as an event block – Appears as a rectangular box 	<ul style="list-style-type: none"> – Graphically represented in SysML – Appears as a solid circle 
Final node	<ul style="list-style-type: none"> – Currently implemented as an event block 	<ul style="list-style-type: none"> – Graphically represented in SysML

Description	MP State Machine	SysML State Machine
	<ul style="list-style-type: none"> Appears as a rectangular box 	<ul style="list-style-type: none"> Appears as a solid circle within a circle 
Composite state	<ul style="list-style-type: none"> Can generate a simple state machine layout at one level of abstraction for each MP schema 	<ul style="list-style-type: none"> Lower-level state machines can be graphically embedded inside a higher-level diagram 

4. Comparison between MP-Firebird and SysML MSOSA Tools

Besides the similarities and differences with the generated MP and SysML behavior diagrams utilizing MP-Firebird and MSOSA respectively, the Model Wreckers observe several differences in tool functionality and mechanism. Table 12 shows the differences between MP-Firebird and SysML MSOSA tools that the team observed during this study.

Table 12. Differences between MP-Firebird and SysML MSOSA tools

Description	MP-Firebird	MSOSA
Tool Automation Capability	<ul style="list-style-type: none"> Automatically generates event trace, activity, and state machine diagrams from event grammar rules 	<ul style="list-style-type: none"> Users have to manually construct each sequence, activity, and state machine diagram individually
Precedence relations between multiple actors in activity diagrams	<ul style="list-style-type: none"> Currently does NOT support graphing coordinated precedence relations between multiple actors to provide a fully integrated view 	<ul style="list-style-type: none"> Supports designing and graphing precedence relations between actions in different swimlanes per users' preferences.
Word wrapping and alignment on the activity diagram	<ul style="list-style-type: none"> Does not automatically wrap and align by whole words; users can manually adjust the box after generation 	<ul style="list-style-type: none"> Automatically wrap and align by whole words 

5. Monterey Phoenix State Machine Observations

Rendering a state machine in MP version 4 is challenging for new modelers. The Auto GCAS state machine was the Model Wrecker's first successful attempt at creating a complex state machine using MP. However, the team overdeveloped the aircraft root event and underdeveloped the pilot root event. All the states and transitions for the state machine were located under the aircraft root event, while the pilot root event contained a few states

for minimal representation. This was an easy way to program a state machine in Monterey Phoenix because few precedence relations went back and forth between the root events. The state machine diagram was perfect, but the event trace diagrams were unrealistic as sequence diagrams (though they still represented valid traces through the state machine diagram).

To create more realistic state machine models in MP, the Model Wreckers used a compact event trace structure to represent the Firearm Safety Model. The compact event trace structure placed the states of the Firearm Safety Model under the weapon root event, and it also placed the transitions caused by the operator under the operator root event. This was a very elegant method for creating a model where both the system and the user interactions were fully represented.

However, due to MP programming inexperience, the Model Wreckers ran into coding challenges while creating the original full Firearm Safety Model in MP. Therefore, the Firearm Safety Model was partitioned into a Top-Level model and an Operational model, and both models utilized the compact event trace structure. By partitioning the original model into two smaller models, the team was able to reduce the model complexity below eight states for each model. Reducing the model complexity reduced the MP coding complexity as well. This allowed the Model Wrecker's team to complete the Top-Level model and the Operational model. The Top-Level model was created first, so the MP code is rough. The Operational model coding was more straightforward, so this shows that the team coding experience improved over time.

Even though the compact event trace structure was challenging to implement in MP code, it allowed the Model Wrecker's team to create a behavioral schema that defined the event trace diagrams, activity diagrams, and the state machine diagram elegantly and accurately. In both the Firearm Safety Top-Level model and the Firearm Safety Operational model, the event trace diagrams were well balanced. The weapon root event contained the state events, and the operator contained the transition events. By using this structure, the root events were not overdeveloped or underdeveloped. With the same MP schema, the compact event trace structure generated an accurate state machine diagram and a well-formed activity diagram as well. These two MP models are the first examples to

demonstrate the extraction of event trace, activity, and state machine diagrams that were accurate and consistent with each other from the same integrated behavior schema. For reference, the activity diagrams for the Firearm Safety Top-Level model and the Firearm Safety Operational model are located in Appendix B: Firearm Safety Model Activity Diagrams.

B. RECOMMENDATIONS

In this section, the Model Wreckers present our findings for the second and third questions posed in Chapter I: “2. What changes does MP v4.0 require in order for MP to produce diagrams consistent with SysML v1.6 standards?” and “3. What changes, if any, does SysML v1.6 require to support MP behavior concepts?”

In subsections 1–3, the team discusses the recommendations and changes required for each MP diagram type, including event trace, activity, and state machine diagrams, in order for MP to produce diagrams consistent with SysML standards. In subsection 4, the team discusses and provides recommendations for future MP development to support generating use case diagrams in MP. Finally, in subsection 5, the team discusses the recommendations and changes that SysML requires in order to support MP behavior concepts.

1. MP Event Trace Diagram

At the moment, the Model Wreckers does not have further recommendations on improving the current MP 4.0 in generating event trace diagram view capability. However, to better serve the SysML community, the Model Wreckers see the opportunity to provide additional capability in MP to convert existing event trace diagrams to standard SysML sequence diagrams with the following recommendations:

1. Provide the capability to illustrate precedence arrows labeled with events between actors

In MP, the precedence relation between events is illustrated by an unlabeled solid line arrow between one event to another event. For example, in Figure 46, the “Command Power Up” is an atomic event that belongs to the Pilot root event. The “Command Power

Up” is a precedence event and originates from the pilot to the aircraft’s lifeline. It states that the pilot has to “Command Power Up” first for the aircraft to execute the “Perform System Initialization” event. When translating to a SysML sequence diagram, the precedence relation between the pilot and the aircraft shall remain unchanged. It shall be illustrated with a solid line arrow that originates from the Pilot lifeline to the Aircraft lifeline. In addition, the name of the atomic root event “Command Power Up” shall be placed above the same message arrow.

2. Provide the capability to illustrate the Message to Self notational concept

When an event is not related to another event under another root actor, that event shall be illustrated using the Message to Self notational concept. The Message to Self has its head and tail pointing to the same lifeline. For example, in Figure 46, the “Perform System Initialization” is an event within the Aircraft root. It does not have any precedence relation to any event under the Pilot root. Therefore, when translating to SysML sequence diagram, the “Perform System Initialization” shall be illustrated using Message to Self notation.

3. Provide the capability to illustrate parallel events

Although both MP and SysML support illustrating events in parallel, the implementation concept is different between these two languages. When translating to SysML, parallel events shall be placed in a vertical stack on the lifeline using the “Par” notation that wraps around all related parallel events. For example, in Figure 46, the “Scan Terrain” and “Display Real-time Operational Status” are parallel events that are illustrated by using the Message to Self and the “Par” notations.

4. Provide the capability to automatically assign the sequence numbers

SysML also displays the order of the sequence by assigning the number for each message based on vertical order. In MSOSA, the sequence numbers are automatically assigned for each sequence. Therefore, the Model Wreckers recommends enhancing MP tool capability by providing an option to automatically assign the sequence number for each event based on vertical order.

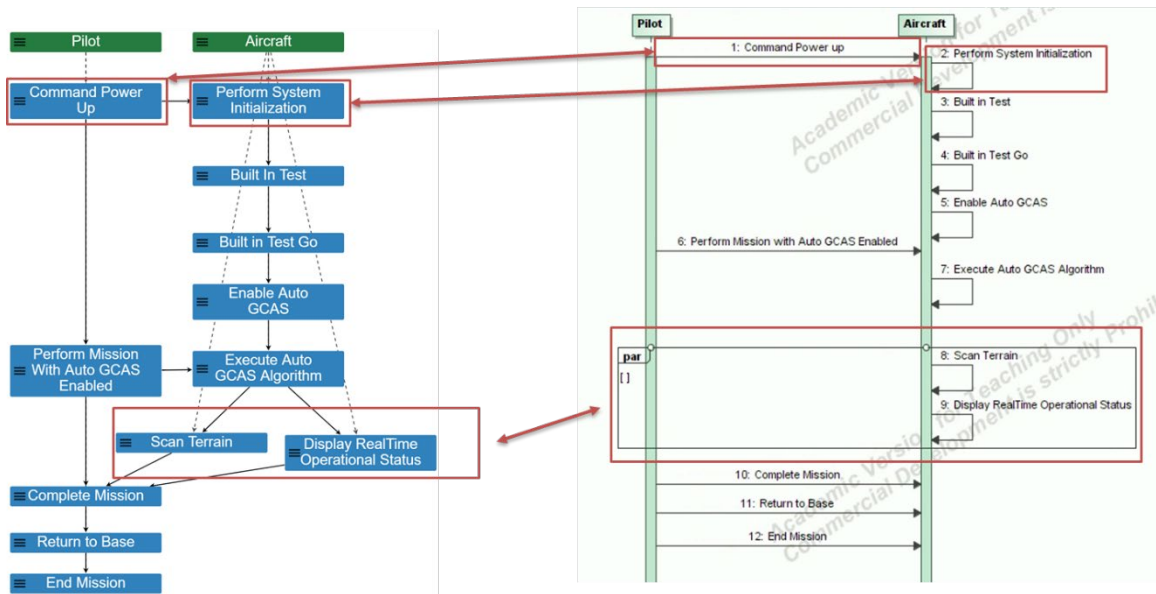


Figure 46. Recommendation for translating MP event trace (left) into SysML sequence diagram (right), illustrated using the Auto GCAS normal condition with Auto GCAS enabled

2. MP Activity Diagram

The study enumerates multiple similarities and differences in the implementation and notational concepts between MP and SysML activity diagrams. The MP activity diagram has some utility to the SysML community, but the MP activity diagram can be improved in multiple ways. From the analysis, the Model Wreckers identified an opportunity to improve the existing MP activity diagram to match standard SysML activity diagrams with the following recommendations:

1. Support the merge node notational concept

While MP supports decision nodes, the merge node notational concept is not currently supported in the activity diagram. The differences are illustrated in Figure 27 and Figure 28. In Figure 27 (right), without the merge node, the arrows from “Perform Mission without Auto GCAS” and “Perform Mission with Auto GCAS Enabled” go straight into the box or end activity node, rather than into a merge node first as in Figure 27 (right). Although the merge node is not needed for generating the activity diagram from the control logic of the MP schema, it is needed by tools that use the merge node as a cue for the

simulation control logic. In order to standardize with SysML notations, the team also recommends supporting and implementing the merge node notational concept in future MP versions.

2. Provide support to graph precedence relations between multiple roots (actors)

Although MP does not currently graph the coordinated precedence relations between multiple actor activity diagrams to provide the complete integrated view, the MP language itself does contain the necessary data to generate this view. Figure 31 illustrates this lack of a “swimlanes” activity view in MP compared to the SysML “swim lanes” activity diagram. Therefore, to match with the common SysML activity diagram view, it is recommended that MP tools support graphing the precedence relations between various roots (actors) onto an integrated activity diagram view.

3. Update fork horizontal and join horizontal notation

The current MP 4.0 language already supports the fork horizontal and join horizontal notations. However, there is no graphical differentiation between these two constructs in MP tools besides the number of threads annotated in the MP fork node. The differences are illustrated in Figure 29. As a result, it is recommended to update the fork horizontal and join horizontal notations in MP to more closely match the SysML standard.

4. Differentiate different types of arrows

In the SysML activity diagram, the solid line arrow represents object flow, while the dashed arrow represents control flow. In contrast, MP utilizes the solid line arrow to represent all precedence types between the activities and provides the user-defined relations as a means to establish other types of relations separately from the ordering relation. Therefore, the team also recommends future versions of MP tools support graphical differentiation of these two types of SysML precedence relationships in MP activity diagrams.

5. Support more action notational concepts including the accept event action and send event action notations

The current MP tools do not graphically support all SysML action notations including accept event action and send event action notation. Instead, the intentionally

simple MP event grammar uses the same single event block to capture the content for these notations. Therefore, the team also recommends providing the capability to differentiate the accept event action and send event action notational concepts in future MP versions.

3. MP State Machine Diagram

The state machine functionality in Monterey Phoenix can be improved in multiple ways. The Model Wreckers team struggled with various issues related to both programming and graphics in relation to the MP state machine functionality. Some of the graphical issues existed in MP-Firebird, but they did not exist in MP-Gryphon. The first set of recommendations is true for MP-Firebird, but not true for MP-Gryphon, and the second set of recommendations is true for both MP-Firebird and MP-Gryphon. Figure 47 shows a comparison between the Auto GCAS state machines for both MP-Gryphon and MP-Firebird. From the analysis, the Model Wreckers see the opportunity to improve the existing MP state machine diagram to match standard SysML state machine diagrams with the following recommendations:

1. Fix the state name truncation issue in MP-Firebird

MP-Firebird truncates event names within the state box if the name is too long. MP-Gryphon does not have this issue because MP-Gryphon will show the entire name for the state box.

2. Allow the user to change the size and color of transition names in MP-Firebird

The transition name is always a light gray with no way to change the transition size. This causes visualization and presentation problems, especially in Microsoft PowerPoint. In MP-Gryphon, the transition names can be changed in color and size.

3. Improve the way MP-Firebird places the states in the diagram

When a state diagram is generated in MP-Firebird, the program places the states in sub-optimal locations in the workspace, and this looks like a spider web that must be untangled for further analysis. In contrast, MP-Gryphon places all the states in a circle; this circle allows the analyst to untangle the state machine diagram with ease.

The next set of recommendations applies to both MP-Gryphon and MP-Firebird.

4. Standardize the mechanism to illustrate the underscore notation in the state machine diagrams

Both Monterey Phoenix tools suppress the underscore in the state blocks of state machine diagrams, but the underscore is not suppressed in the transition names. For consistency, the underscores should be suppressed in the transition names as well.

5. Support initial node and final node state notional concepts similar to SysML

SysML includes both initial nodes and final nodes which are different shapes from the other states. State machines graphed from MP schemas should include initial nodes and final nodes as well. Currently, the analyst must create a regular state named “Initial” and another regular state named “Final” to substitute for the initial and final nodes in SysML.

6. Simplify the state machine creation process

This recommendation relates to the MP code which defines and generates the state machine. There is a block of code that must be composed or inserted into the MP schema file and tailored to create the state machine diagrams. It would be helpful for the developers to reduce the code complexity and streamline the state machine creation process. After the MP schema is created and the event trace diagrams describe the system, the creation of the activity diagram involves one single line of code. From a user perspective, it would be helpful if the state machine was created in a single line of code as well. However, to create the state machine, the programmer must create a large block of code that specifies which events are states and which events are transitions. This process can become tedious and difficult for large MP models. The future work ideas described in the following section (in particular, in Chapter V Section C.3 “Monterey Phoenix User Interface Capability“) may be appropriate for consideration for simplifying the user’s task of graphing a state machine diagram.

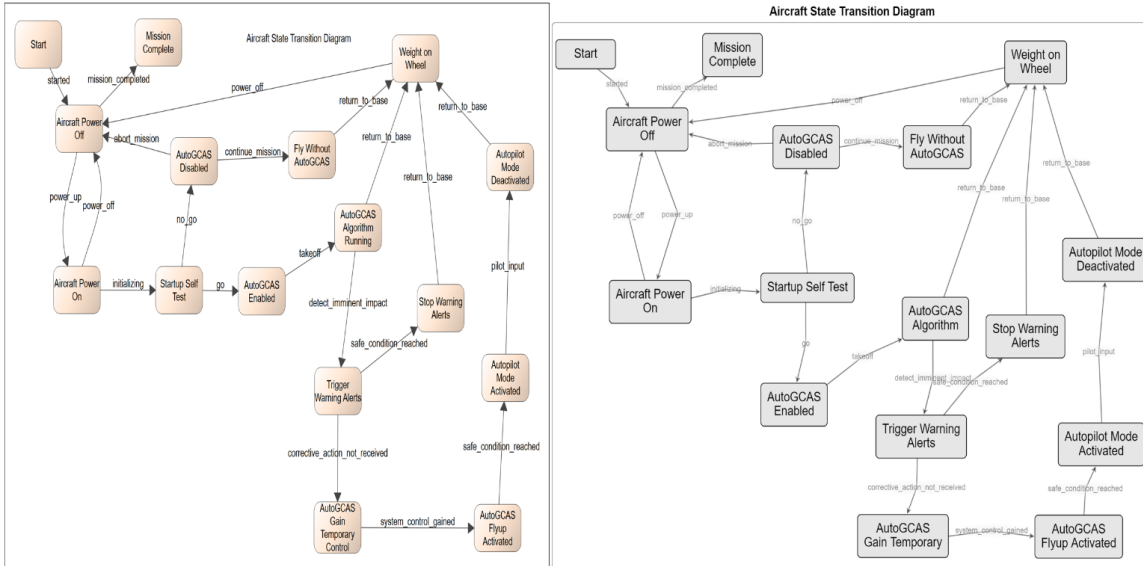


Figure 47. Comparison of Auto GCAS state machine models between MP Gryphon (left) and MP Firebird (right)

4. Recommendation for MP Use Case Diagram Development

Since MP-Firebird does not currently support use case diagrams, the team did not use MP to automate the generation of the use case diagram in this study. Instead, the team used MSOSA and the Firearm Safety Model to generate a use case diagram. New functionality in future versions of the MP modeling language and tools to generate these diagrams would be valuable additions. The Model Wreckers constructed a concept of a use case diagram that could be generated from MP schemas, shown in Figure 48.

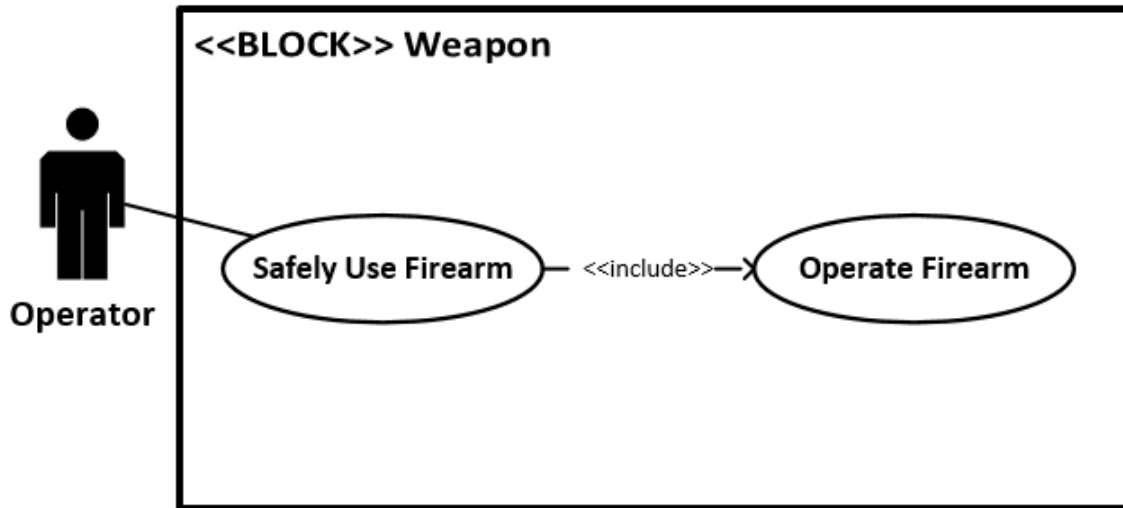


Figure 48. Firearm Safety model use case diagram concept, illustrated using Microsoft Visio

The MP use case diagram concept uses the Firearm Safety Model as an example. However, in this example, the Firearm Safety Top-Level model is renamed as the Safely Use Firearm model, and the Firearm Safety Operational model is renamed as the Operate Firearm model. The models are renamed in this example so that the ovals contain well-named use cases, which could be taken from the MP schema name in the future.

The various use cases will each correspond to an MP schema with the same name. Within the schema are root events, which for this example are the Operator and Weapon root events. The boundary box will be named for the root event which is the system of interest. In this example, the Weapon is the system of interest. All other root events will be actors placed outside of the boundary box in the use case diagram. The system of interest root event must be defined for use case diagram creation. After the system of interest is defined in the MP schema, a command such as “SHOW USECASE DIAGRAM Weapon” could be used to generate the use case diagram. By placing the Weapon root event name in the command, MP knows which root event is the system of interest, and thus, what root event the boundary box should be named after.

The root events specified as actors will be connected to the main use case with a solid line, and the main use case will connect to the subordinate use cases with an include or extend connector depending on the relationship to the main use case.

Another new layer of functionality could link multiple schemas that involve the same root events and use cases. If the MP modeler decided to create a new schema named Store Firearm that is neither included in nor extended from the other use cases (just a separate oval), the modeler would need a way to show this new schema alongside the others (Safely Use Firearm and Operate Firearm) on the same diagram. The new use case diagram would visualize the three related MP schemas as use cases (Safely Use Firearm, Operate Firearm, and Store Firearm) along with the connectors as well.

Figure 49 illustrates the use case diagram for the Firearm Safety Model, developed in MSOSA. The figure shows the two use cases for the Firearm Safety Model (Safely Use Firearm and Operate Firearm). The Safely Use Firearm use case has an include arrow which connects to the Operate Firearm use case. The Operator actor is the root event which is not the system of interest, and the block is the root event which is the system of interest (Weapon). There is only one difference between the proposed MP use case diagram in Figure 48 and the MSOSA diagram in Figure 49. The system of interest name in the proposed MP use case diagram is to the right of the <<block>> designator. In the MSOSA use case diagram, the system of interest name is in the label in the top left corner of the diagram.

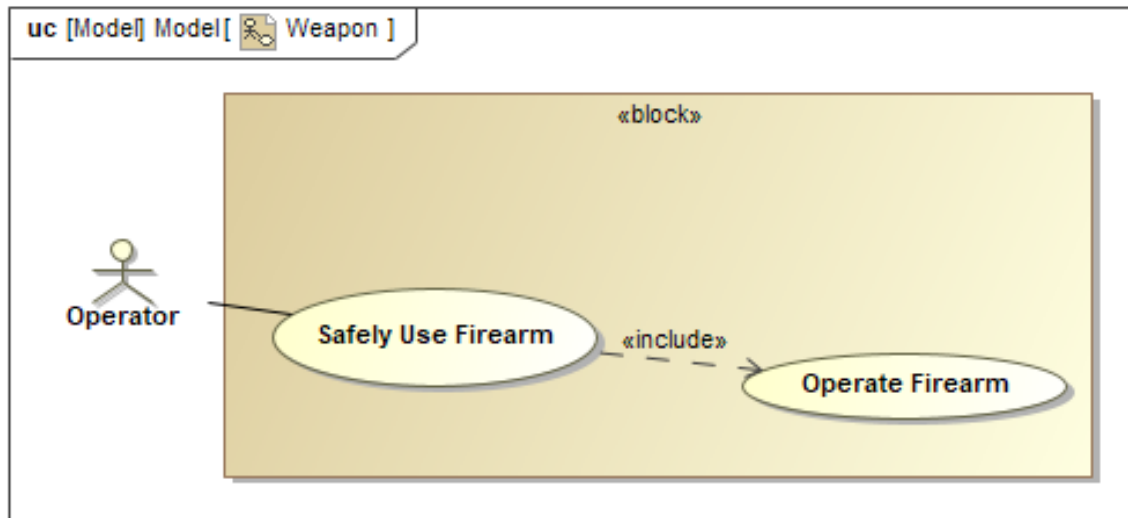


Figure 49. Firearm Safety model use case diagram concept, illustrated using MSOSA

5. Enhancement Recommendation for SysML Behavior Diagrams

Within the scope and analysis results of this project, Dr. Giammarco and the Model Wreckers noted no changes that would be needed to the SysML specification to accommodate MP-generated behavior diagrams. This absence of a finding suggests a promising future for the visualization of MP schemas and the traces and global views produced from them as SysML diagrams. During the research, we did note one area within SysML sequence diagram that can be improved. On the sequence diagram, MP and SysML illustrate parallel sequences and events that occur at the same time differently. In MP, the parallel events/sequences are illustrated by branching from the parent event to multiple lower-level events as shown in Figure 22 (left). These parallel events can be rearranged freely to satisfy the visual presentation needs. In contrast, SysML uses the parallel option (Par) to illustrate actions occurring in parallel. As illustrated in Figure 22 (right), the parallel events are stacked on the lifeline and numbered ascendingly, such as “8: Scan Terrain” and “9: Display Real-time Operational Status.” This illustration, however, gives them the appearance that these activities are in sequential order, not in parallel to novice users. The team believes that parallel events should be illustrated on the same horizontal timeline whenever possible. As a result, the Model Wreckers suggests consideration of an

option to illustrate parallel events in the same horizontal plane on the sequence diagram by expanding the lifeline left or right. Figure 50 (right) demonstrates how the team believes parallel events could be displayed on the sequence diagram, but without extensive testing or consideration of implementation constraints. On this modified diagram, the Par box is rotated horizontally, and additional parallel events can be illustrated side by side in the same horizontal plane. As a note, Figure 50 is for demonstration purposes only and it is not known if a current or future SysML tool could support illustrating parallel events in this configuration.

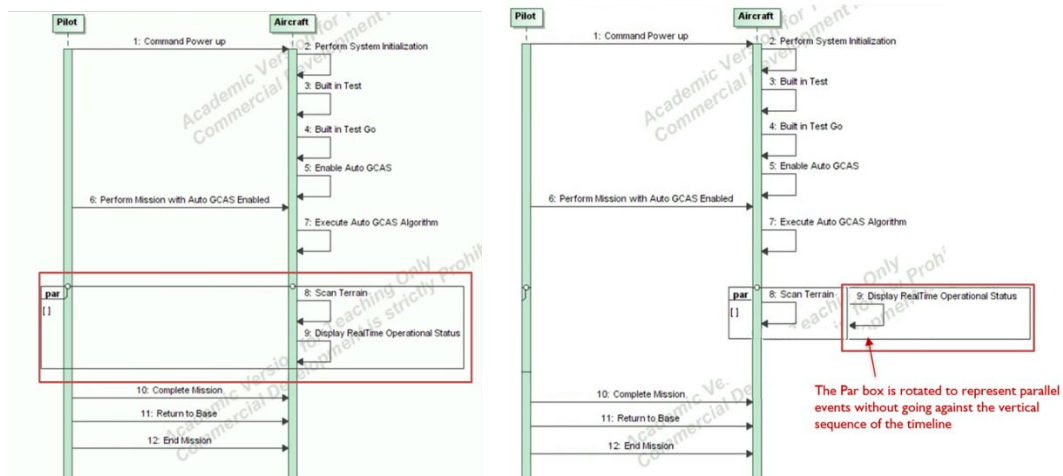


Figure 50. Original SysML sequence diagram (left) vs. modified SysML sequence diagram (right), illustrated using the Auto GCAS normal condition model

C. FUTURE WORK

The Model Wreckers propose the following ideas for future research to keep MP's diagrams relevant and gain acceptance among practicing systems engineers. The concepts stem from personal experiences among the Model Wreckers team during the building of the case study models in MP. These recommendations for future work could prove to push a greater acceptance of and facilitate a greater understanding of the power behind MP behavior modeling.

1. Event Trace Diagram Table Tool

In order to create the event trace tables in the document (Table 5 and Table 6), the team manually recorded the ten event traces from the Firearm Safety Top-Level model and the four event traces from the Firearm Safety Operational model. The team carefully followed each event trace precedence relation for each event trace in the Firearm Safety Top-Level model and the Firearm Safety Operational model as well. This information was placed in the fields of Table 5 for the Firearm Safety Top-Level model and Table 6 for the Firearm Safety Operational model. This was a very time-consuming process that involved multiple checks for accuracy. Both tables were manually created in Microsoft Excel by the team. However, MP can automatically generate tables, but the team did not focus on the table generation capabilities of MP for this capstone research.

It would be very useful for the modeler if MP could automatically generate the event trace tables after the MP schema is modeled. This tool would be called the event trace diagram table tool. The event trace diagram table tool could be either a Graphical User Interface (GUI) tool or a specific MP command or block of code that would instruct MP to generate an event trace table for the MP schema. This table could be exported as a table or a figure, and the exported table or figure would show all the event traces in the MP schema file in a simple and compact way for scholarly articles.

2. Use Case Diagrams in Monterey Phoenix

Monterey Phoenix can create event trace diagrams, activity diagrams, and state machine diagrams. However, currently, MP is not able to create use case diagrams from the MP schema file. The actors in the use case diagrams would be the root events, and the use case blocks could be schema names or a subset of the atomic or composite events within MP. The MP modeler may need to specify the include and extend connectors if MP cannot derive them directly. This MP use case diagram capability was discussed in detail in Chapter V Section B.4 “Recommendation for MP Use Case Diagram Development.”

3. Monterey Phoenix User Interface Capability

One of the challenges with Monterey Phoenix is the coding interface. There are numerous examples of custom software/programming languages being moved into the background from the common/intended user. The Windows operating system replacing DOS could be the most successful example. It made the personal computer more usable to a broader population of people, forever changing our society.

To create a set of diagrams for the system, the modeler must specify the schema using the MP coding language. Within the MP coding language, there is a lot of repetition in assigning variables to events, only to use the variables for assigning precedence relationships. One of the future improvements for MP would be to implement a conversational programming interface (CPI) that assists the modeler during MP model creation.

A concept for the implementation of this CPI stems from the manufacturing world. Computer Numerical Controlled machines all run on a custom programming language, that requires intimate knowledge of the syntax for hundreds of canned command codes. These command codes control every aspect of the machine's movements. There are numerous software packages that exist to allow the operators of these machines to generate the program code in the background through a graphical user interface (GUI) (that is then uploaded into the machine), while some machines come with a built-in CPI. The CPI allows for faster programming by walking the operator through a bunch of prompt windows that represent the common operations of the machine to achieve product dimensions. By allowing the user to define the desired dimensions, the CPI automatically generates the underlying command code to run the machine. This type of interface could be a great first step in opening MP's utility to a broader audience and greater utility.

Translating this CPI concept into MP model building could result in a set of window prompts that would walk the user through the creation of the model. In the first phase of model construction, MP starts with a conversational window that asks the modeler for the number of root events in the schema, along with the names of the root events. The next set of conversational windows would ask the modeler for the composite and atomic events

associated with each root event. MP then creates all the root, composite, and atomic events, and the MP conversational window would create the inclusion and precedence relations for each individual root event.

In the second phase of model construction, the conversational window asks the user to specify precedence relations that go to and from atomic and composite events that have differing root events. This conversational window would be very useful because the Model Wreckers team struggled with this step while creating the event traces for both the Firearm Safety Top-Level model and the Firearm Safety Operational model. The conversational window would solve any conflicts and other issues between precedence relation statements in the MP code.

The final phase of model construction would ask the user if a state machine diagram and/or an activity diagram should be created in the model. If the user says yes to the state machine diagram, a new conversational window will ask the modeler the names of the states, the names of the transitions, and any extra information on how the states and transitions connect to each other. After this step, the MP conversational window creates a state diagram, activity diagram, or both.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MP MODEL SOURCE CODE

1. AutoGCAS_Normal_Condition_V1_4

```
/*
*
* [ Title and Authors ]
* Normal event trace model of the Auto GCAS system
* Created by Joe Hall on 7 May 2022
*
*
* [ Purpose ]
* This is the MP Model that represents the
* normal operating condition for the Auto GCAS
* system. This model contains 3 traces in the normal
* Auto GCAS mode.
*
*
* [ Description ]
* This is one of two MP models for the Auto GCAS
* system (Lockheed Martin 2021). The model contains
* 3 event trace diagrams which correspond to the
* MSOSA activity diagram which displays normal Auto
* GCAS behavior (no impact detected).
*
*
* [ References ]
* 1. "Automatic Ground Collision Avoidance System."
* 2021. Lockheed Martin.
* https://www.lockheedmartin.com/en-us/products/autogcas.html
*
*
* [ Search Terms ]
* diagram, state, event, trace, Auto, GCAS, sequence
* activity
*
*
* [ Instructions ]
* Run at Scope 1 only.
* [ Run Statistics ]
* Scope 1: Creates 3 traces, less than 1 second
*
*/
```

SCHEMA AutoGCAS_Normal_Condition_V1_4

ROOT Pilot:

```
Command_Power_Up
(
  (Perform_Mission_With_Auto_GCAS_Enabled |
   Perform_Mission_Without_Auto_GCAS
  )
Complete_Mission
```

```

Return_to_Base
Abort_Mission
)
End_Mission;

ROOT Aircraft:

Perform_System_Initialization
Built_In_Test

(Built_in_Test_Go
Enable_Auto_GCAS
Execute_Auto_GCAS_Algorithm
    {Display_RealTime_Operational_Status,
    Scan_Terrain
    }
Built_in_Test_No_Go
Disable_Auto_GCAS
    (Abort_Mission_Yes
    Abort_Mission_No
    )
);

/* Coordinate Statements */
COORDINATE $x: Perform_Mission_With_Auto_GCAS_Enabled,
           $y: Execute_Auto_GCAS_Algorithm
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Command_Power_Up,
           $y: Perform_System_Initialization
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Abort_Mission_No,
           $y: Perform_Mission_Without_Auto_GCAS
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Abort_Mission_Yes,
           $y: Abort_Mission
           DO ADD $x PRECEDES $y; OD;

/* This last coordinate statement allows */
/* for both Scan_Terrain and Display_RealTime_Operational_Status */
/* to have a precedence relation to Complete_Mission */
/* The IF statement is necessary because I only want these precedence */
/* */
/* to Complete_Mission if Scan_Terrain and */
/* Display_RealTime_Operational_Status */
/* exist */

IF #Scan_Terrain > 0 AND #Display_RealTime_Operational_Status > 0 THEN

    COORDINATE $x: Display_RealTime_Operational_Status,

```

```
        $y: Scan_Terrain,  
        $z: Complete_Mission  
  
    DO  ADD $x PRECEDES $z;  
        ADD $y PRECEDES $z;  
    OD;  
FI;  
  
/* Plots */  
GLOBAL  
  
SHOW ACTIVITY DIAGRAM Pilot;  
SHOW ACTIVITY DIAGRAM Aircraft;
```

2. AutoGCAS_Impact_Condition_V1_4

```
/*
*
* [ Title and Authors ]-----
* Impact event trace model of the Auto GCAS system
* Created by Joe Hall on 9 May 2022
*
*
*
* [ Purpose ]-----
* This is the MP Model that represents the
* impact operating condition for the Auto GCAS
* system. This impact operating condition is where
* the Auto GCAS system detects an imminent impact and
* then takes action to prevent the aircraft from
* crashing. This model contains 3 traces for the
* impact condition, but the model must be run at a
* scope of 2.
*
*
* [ Description ]-----
* This is the second of two MP models for the Auto
* GCAS system (Lockheed Martin 2021). The model
* contains 3 event trace diagrams which correspond to
* the MSOSA activity diagram which displays impact
* auto GCAS behavior.
*
*
* [ References ]-----
* 1. "Automatic Ground Collision Avoidance System."
* 2021. Lockheed Martin.
* https://www.lockheedmartin.com/en-us/products/autogcas.html
*
*
* [ Search Terms ]-----
* diagram, state, event, trace, Auto, GCAS, sequence
* activity
*
*
* [ Instructions ]-----
* RUN AT SCOPE 2 FOR ALL 3 TRACES.
* [ Run Statistics ]-----
* Scope 2: Creates 3 traces, less than 1 second
*
*/
```

SCHEMA AutoGCAS_Impact_Condition_V1_4

ROOT Pilot:

```
Perform_Mission_With_Auto_GCAS_Enabled
(+ Take_Corrective_Action +)
[Pilot_Inputs]
Complete_Mission
Return_to_Base
End_Mission;
```

ROOT Aircraft:

```
Detect_Imminent_Ground_Impact

(Provide_Warning_Alerts
Determine_No_Corrective_Action_Executed
Gain_Temporary_Control
Execute_FlyUp_Command      /* Section where pilot does not provide
*/
Display_AutoGCAS_FlyUp_Cue /* Any inputs */
Maneuver_To_Safe_Altitude
Activate_Autopilot_Mode
{Send_Aircraft_Status_To_Mission_Commander,
Continue_Autopilot_Mode_and_Monitor_Environment_Conditions
}
Deactivate_Autopilot_Mode
Return_Control_To_Pilot

(* Provide_Warning_Alerts
Determine_Corrective_Action_is_Executed
Continue_Evaluate_Flight_Conditions /* Section where pilot does
*/
Determine_Safe_Condition_No      /* Provide Inputs */
*)
Provide_Warning_Alerts
Determine_Corrective_Action_is_Executed
Continue_Evaluate_Flight_Conditions
Determine_Safe_Condition_Yes
Stop_Warning
);
```

ROOT Mission_Commander:

```
[Aircraft_Status_Received];

/* Coordinate Statements */

COORDINATE $x: Perform_Mission_With_Auto_GCAS_Enabled,
           $y: Detect_Imminent_Ground_Impact
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Provide_Warning_Alerts,
           $y: Take_Corrective_Action
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Take_Corrective_Action,
           $y: (Determine_Corrective_Action_is_Executed
Determine_No_Corrective_Action_Executed )
           DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Send_Aircraft_Status_To_Mission_Commander,
           $y: Aircraft_Status_Received
           DO ADD $x PRECEDES $y; OD;
```

```
COORDINATE $x: Pilot_Inputs,  
           $y: Deactivate_Autopilot_Mode  
DO ADD $x PRECEDES $y; OD;  
  
COORDINATE $x: ( Stop_Warning | Return_Control_To_Pilot ),  
           $y: Complete_Mission  
DO ADD $x PRECEDES $y; OD;  
  
/* Show Plots */  
GLOBAL  
  
SHOW ACTIVITY DIAGRAM Pilot;  
SHOW ACTIVITY DIAGRAM Aircraft;  
SHOW ACTIVITY DIAGRAM Mission_Commander;
```

3. AutoGCAS_State_Machine_Diagram_V1_3

```
/*
*
* [ Title and Authors ]
* Operational state machine of the Auto GCAS Model
* Created on 14 April 2022
*
*
* [ Purpose ]
* This is the state machine of the Auto GCAS model.
* This model is an exact representation of the
* original state machine created in MSOSA.
*
*
* [ Description ]
* This is the state machine model for the Auto GCAS
* system (Lockheed Martin 2021). The model contains
* 16 states and around 20 transitions. This state
* machine is an accurate representation of the MSOSA
* state machine model, but the event trace diagrams
* are not fully developed. the aircraft root event
* contains all of the states and transitions, but the
* pilot root event contains very few states and
* transitions. The imbalance of states and
* transitions between the pilot and aircraft actors
* is further explored in the firearm safety top level
* and operational state machines models. Also,
* the state events are capitalized, and the
* transition events are lower case and past tense.
* This allows for the programmer to determine which
* atomic event is a state and which is a transition
* when viewing the event trace diagrams.
*
*
* [ References ]
* 1. "Automatic Ground Collision Avoidance System."
* 2021. Lockheed Martin.
* https://www.lockheedmartin.com/en-us/products/autogcas.html
* 2. Top Level State Machine for Firearm Safety Model
* https://staging.firebird.nps.edu/project/b222d845-4473-4f11-95d9-
4b2089a553bd
* 3. Operational State Machine for Firearm Safety
* https://staging.firebird.nps.edu/project/8ff5259b-a370-43b7-9317-
4067fe26199c
*
*
* [ Search Terms ]
* diagram, state, transition, Auto, GCAS
*
*
* [ Instructions ]
* Run at Scope 1 only.
* [ Run Statistics ]
* Scope 1: Creates 7 traces, less than 1 second
*
*
*/
```



```
L*|-----*/
```

```
SCHEMA AutoGCAS_State_Machine_Diagram_V1_3
```

```
ROOT Pilot:
```

```
Do_Nothing
(*
  Power_System
  [Pilot_Joystick_Input]
  Turn_Off_System
*)
;
```

```
/* In this state machine model, most of the states and transitions are
in */
```

```
/* the Aircraft root event. This is the easiest method to create a
complicated */
```

```
/* state machine. States are capitalized, and transitions are lower
case and */
```

```
/* past tense. */
```

```
ROOT Aircraft:
```

```
Start
started
Aircraft_Power_Off
(*
  (power_up      /* Shortest path, Aircraft_Power_On to
Aircraft_Power_Off */
  Aircraft_Power_On
  power_off
  power_up
  Aircraft_Power_On
  initializing
  Startup_Self_Test
  (
    no_go
    (
      AutoGCAS_Disabled /* No Go Section1 */
      abort_mission
      AutoGCAS_Disabled
      continue_mission /* No Go Section 2 */
      Fly_Without_AutoGCAS
      return_to_base
      Weight_on_Wheel
      power_off
    )
  )
  go
  AutoGCAS_Enabled /* Part of Go Sections 1-3 */
  takeoff
  AutoGCAS_Algorithm_Running
  [detect_imminent_impact
  Trigger_Warning_Alerts
```

```

        (corrective_action_not_received
        AutoGCAS_Gain_Temporary_Control
        system_control_gained
        AutoGCAS_Flyup_Activated      /* Go Section 2 */
        safe_condition_reached
        Autopilot_Mode_Activated
        pilot_input
        Autopilot_Mode_Deactivated

        safe_condition_reached
        Stop_Warning_Alerts           /* Go Section 3 */
    )
]
return_to_base
Weight_on_Wheel
power_off
    )
)
Aircraft_Power_Off
*)

mission_completed
Mission_Complete;

/* Coordinate Statements */
/* Power Related Coordinate Statements */

COORDINATE $a: Power_System, $b: power_up
    DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Turn_Off_System, $b: (power_off | abort_mission)
    DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Pilot_Joystick_Input, $b: pilot_input
    DO ADD $a PRECEDES $b ; OD;

/* State Chart */

GRAPH StateDiagram{ TITLE("Aircraft State Transition Diagram"); };

WITHIN StateDiagram{
    /* The sort method for state diagrams tends to work better for the
    larger diagrams */
    COORDINATE
        <SORT> $statal:
        ( Start | Aircraft_Power_Off | Aircraft_Power_On |
        Startup_Self_Test | AutoGCAS_Disabled
        | Fly_Without_AutoGCAS | AutoGCAS_Enabled |
        AutoGCAS_Algorithm_Running
        | Trigger_Warning_Alerts | AutoGCAS_Gain_Temporary_Control |
        Stop_Warning_Alerts
        | AutoGCAS_Flyup_Activated | Autopilot_Mode_Activated |
        Autopilot_Mode_Deactivated
        | Weight_on_Wheel),

```

```

        <SORT> $transition:
        ( started | power_up | initializing | no_go | go |
continue_mission | takeoff
        | detect_imminent_impact | corrective_action_not_received
        | safe_condition_reached | system_control_gained |
pilot_input
        | return_to_base | abort_mission | power_off |
mission_completed),

        <SORT> $state2:
        ( Aircraft_Power_Off | Aircraft_Power_On | Startup_Self_Test |
AutoGCAS_Disabled
        | Fly_Without_AutoGCAS | AutoGCAS_Enabled |
AutoGCAS_Algorithm_Running
        | Trigger_Warning_Alerts | AutoGCAS_Gain_Temporary_Control |
Stop_Warning_Alerts
        | AutoGCAS_Flyup_Activated | Autopilot_Mode_Activated |
Autopilot_Mode_Deactivated
        | Weight_on_Wheel | Mission_Complete)
    DO
        ADD LAST ($state1) ARROW($transition) LAST ($state2);

    OD;
};

/* Plot the charts */
GLOBAL

SHOW StateDiagram;
/* SHOW ACTIVITY DIAGRAM Pilot;
SHOW ACTIVITY DIAGRAM Aircraft; */

```

4. Firearm_Safety_Top_Level_V1_4

```
/*
*
* [ Title and Authors ]
* Top Level state machine of the Firearm Safety Model
* Created by Joe Hall with assistance from Mike
* Savacool on 7 May 2022
*
*
* [ Purpose ]
* This is the top level state machine of the
* Firearm Safety Model. The top level state machine
* model contains the top level states and the
* encapsulated state called "operational." This model
* uses the compact event trace structure where the
* weapon root event contains the weapon states and
* the operator root event contains the state
* transitions.
*
*
* [ Description ]
* This is one of two models of the original Firearm
* Safety Model. The original model was broken into a
* a top level model and an encapsulated model named
* "Operational." This was necessary because the full
* model was too complicated for the team.
* Breaking the original firearm state machine
* into two state machines allowed the team to finish
* the models. The encapsulation concept is from
* (Delligati 2014).
*
*
* [ References ]
* Delligati, Lenny. 2014. "SysML Distilled: A Brief
* Guide to the Systems Modeling Language." Addison
* Wesley.
*
*
* [ Search Terms ]
* top level, diagram, state, firearm, safety
*
*
* [ Instructions ]
* Run at Scope 1 only.
* [ Run Statistics ]
* Scope 1: Creates 10 traces in less than a second
*
*/
```

SCHEMA Firearm_Safety_Top_Level_V1_4

ROOT Weapon: /* This System has states and transitions. Use lower-case, passive voice, past tense verbs for transitions. Begin and end in the same state.*/

```

START
Inert

(* Status_Unknown
  (Broken
   [Unsafe]
   Safe

   [Operational
    Safe
   ]

   (Inert
    Broken
    Operational
    Broken
   )
  *)

END

;
/* Both the weapon and operator root actors mirror each other in
structure. */
/* This is the simplest way to synch both root actors while creating
the */
/* state machine. */
ROOT Operator:
  started

  (* picked_up
    (failed_inspection
     [loaded_safety_off]
     unloaded_safety_on

     [loaded_safety_on
      unloaded_safety_on
     ]

     (put_down
      failed_inspection
      loaded_safety_on
      unable_to_fix
     )
    *)

  ended
;

/* Base Coordinate Statements */

COORDINATE $a: START, $b: started
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: ( put_down | started ), $b: Inert

```

```

DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: ended, $b: END
DO ADD $a PRECEDES $b ; OD;

/* Coordinate statements going through Status_Unknown, Safe, and Broken
*/

COORDINATE $a: picked_up, $b: Status_Unknown
DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: loaded_safety_off, $b: Unsafe
DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: ( failed_inspection | unable_to_fix ), $b: Broken
DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: unloaded_safety_on, $b: Safe
DO ADD $a PRECEDES $b ; OD;

/* Operational Section */

COORDINATE $a: loaded_safety_on, $b: Operational
DO ADD $a PRECEDES $b ; OD;

/* If Statements */

/* Handles issues with Inert and Broken both going to ended*/
/* in different traces */
IF #Inert > 0 AND #Broken == 0 THEN
COORDINATE $a: Inert, $b: ( ended | picked_up)
DO ADD $a PRECEDES $b ; OD;
FI;

IF #Inert > 0 AND #Broken > 0 THEN
COORDINATE $a: Inert, $b: picked_up
DO ADD $a PRECEDES $b ; OD;
COORDINATE $a: Broken, $b: ended
DO ADD $a PRECEDES $b ; OD;
FI;

/* Handles issues with Status_Unknown and Unsafe showing up together */
/* in a trace */
IF #Status_Unknown > 0 AND #Unsafe > 0 THEN
COORDINATE $a: Status_Unknown, $b: loaded_safety_off
DO ADD $a PRECEDES $b; OD;
COORDINATE $a: Unsafe, $b: unloaded_safety_on
DO ADD $a PRECEDES $b; OD;
FI;

/* The rest of these IF statements allow for edge case event traces
that */
/* need to be kept to generated the state machine */
IF #Safe > 0 THEN
COORDINATE $a: Safe, $b: (put_down | failed_inspection |
loaded_safety_on )

```

```

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

IF #Status_Unknown > 0 AND #Operational == 0 AND #Unsafe == 0 THEN
COORDINATE $a: Status_Unknown, $b: ( unloaded_safety_on |
failed_inspection )
DO ADD $a PRECEDES $b ; OD;
FI;

IF #Status_Unknown > 0 AND #Operational > 0 AND #Unsafe == 0 THEN
COORDINATE $a: ( Status_Unknown | Operational ) , $b: (
unloaded_safety_on | unable_to_fix )
DO ADD $a PRECEDES $b ; OD;
FI;

IF #Status_Unknown > 0 AND #Operational > 0 AND #Unsafe > 0 AND
#unable_to_fix > 0 THEN
COORDINATE $a: Operational , $b: unable_to_fix
DO ADD $a PRECEDES $b ; OD;
FI;

/* State Chart */

GRAPH StateDiagram{ TITLE("Top Level Weapon State Transition Diagram");
};

COORDINATE
<CUT_END> $state1: /* these are Weapon's states */
( START | Broken | Inert | Safe | Unsafe |
Status_Unknown | Operational | END),

$transition: /* these are Weapon's state transitions,
which are
triggered by Shooter's inputs */
( started | failed_inspection | picked_up |
unloaded_safety_on |
loaded_safety_off | loaded_safety_on |
unable_to_fix | put_down | ended ),

<CUT_FRONT> $state2: /* these are Weapon's states */
( START | Broken | Inert | Safe | Unsafe |
Status_Unknown | Operational | END )

DO WITHIN StateDiagram { ADD LAST ($state1)
ARROW($transition)
LAST ($state2);
};

OD;

/* Plot the charts */
GLOBAL

SHOW StateDiagram;
/*SHOW ACTIVITY DIAGRAM Weapon;
SHOW ACTIVITY DIAGRAM Operator; */

```

5. Firearm_Safety_Operational_V1_4

```
/*
*
* [ Title and Authors ]
* Operational state machine of the Firearm Safety
* Model
* Created by Joe Hall with assistance from Mike
* Savacool on 7 May 2022
*
*
* [ Purpose ]
* This is the operational state machine of the
* Firearm Safety Model. The operational state machine
* is an encapsulated model within the Firearm Safety
* top level model. This model uses the compact event
* event trace structure where the weapon root event
* contains the weapon states and the operator root
* event contains the state transitions.
*
*
* [ Description ]
* This is one of two models of the original Firearm
* Safety Model. The original model was broken into a
* a top level model and an encapsulated model named
* "Operational." This was necessary because the full
* model was too complicated for the team
* Breaking the original firearm state machine
* into two state machines allowed the team to finish
* the models. The encapsulation concept is from
* (Delligati 2014).
*
*
* [ References ]
* Delligati, Lenny. 2014. "SysML Distilled: A Brief
* Guide to the Systems Modeling Language." Addison
* Wesley.
*
*
* [ Search Terms ]
* diagram, state, firearm, safety, operational
*
*
* [ Instructions ]
* Run at Scope 1 only.
* [ Run Statistics ]
* Scope 1: Creates 4 traces in less than a second
*
*/
```

SCHEMA Firearm_Safety_Operational_V1_4

ROOT Weapon: /* This System has states and transitions. Use lower-case, passive voice, past tense verbs for transitions. Begin and end in the same state.*/


```

(+
  START
  Standby
  [Ready
  Aiming
  Firing

      (Fault
      Ready
      Standby
      Aiming
      Ready
      Standby
      Fault
      )
  ]

  END
+)
;
/* Both the weapon and operator root actors mirror each other in
structure. */
/* This is the simplest way to synch both root actors while creating
the */
/* state machine. */
ROOT Operator:

(+
  loaded_safety_on
  (unloaded_safety_on
  safety_off
  find_target
  trigger_pressed
  (jammed
  clear_jam
  safety_on
  unloaded_safety_on
  trigger_released
  lose_target
  safety_on
  unloaded_safety_on
  jammed
  unable_to_fix
  )
  )
+)
;

/* Base Coordinate Statements */

COORDINATE $a: START, $b: loaded_safety_on
DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: (unloaded_safety_on | unable_to_fix ), $b: END
DO ADD $a PRECEDES $b ; OD;

```

```

COORDINATE $a: (lose_target | clear_jam | safety_off), $b: Ready
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: (loaded_safety_on | safety_on), $b: Standby
  DO ADD $a PRECEDES $b ; OD;

/* Ready-Aiming-Firing Loop */

COORDINATE $a: trigger_pressed, $b: Firing
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Firing, $b: ( trigger_released | jammed)
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: jammed, $b: Fault
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Fault, $b: ( unable_to_fix | clear_jam)
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Ready, $b: ( safety_on | find_target )
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: ( find_target | trigger_released ), $b: Aiming
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Aiming, $b: ( lose_target | trigger_pressed )
  DO ADD $a PRECEDES $b ; OD;

COORDINATE $a: Standby, $b: (unloaded_safety_on | safety_off)
  DO ADD $a PRECEDES $b ; OD;

/* State Machine Diagram */

GRAPH StateDiagram{ TITLE("Operational State Transition Diagram"); };

COORDINATE
  <CUT_END>   $state1:
              ( START | Standby | Ready | Aiming | Firing | Fault
| END),

              $transition:
              ( loaded_safety_on | unloaded_safety_on |
find_target
              | lose_target | trigger_pressed |
trigger_released
              | jammed | unable_to_fix | clear_jam |
safety_off
              | safety_on ),

  <CUT_FRONT> $state2:
              ( START | Standby | Ready | Aiming | Firing | Fault
| END )

```

```
DO WITHIN StateDiagram { ADD    LAST ($state1)
                                ARROW($transition)
                                LAST ($state2);
                                };
OD;

/* Create the plots */

GLOBAL

SHOW StateDiagram;
/* SHOW ACTIVITY DIAGRAM Weapon, Operator; */
```

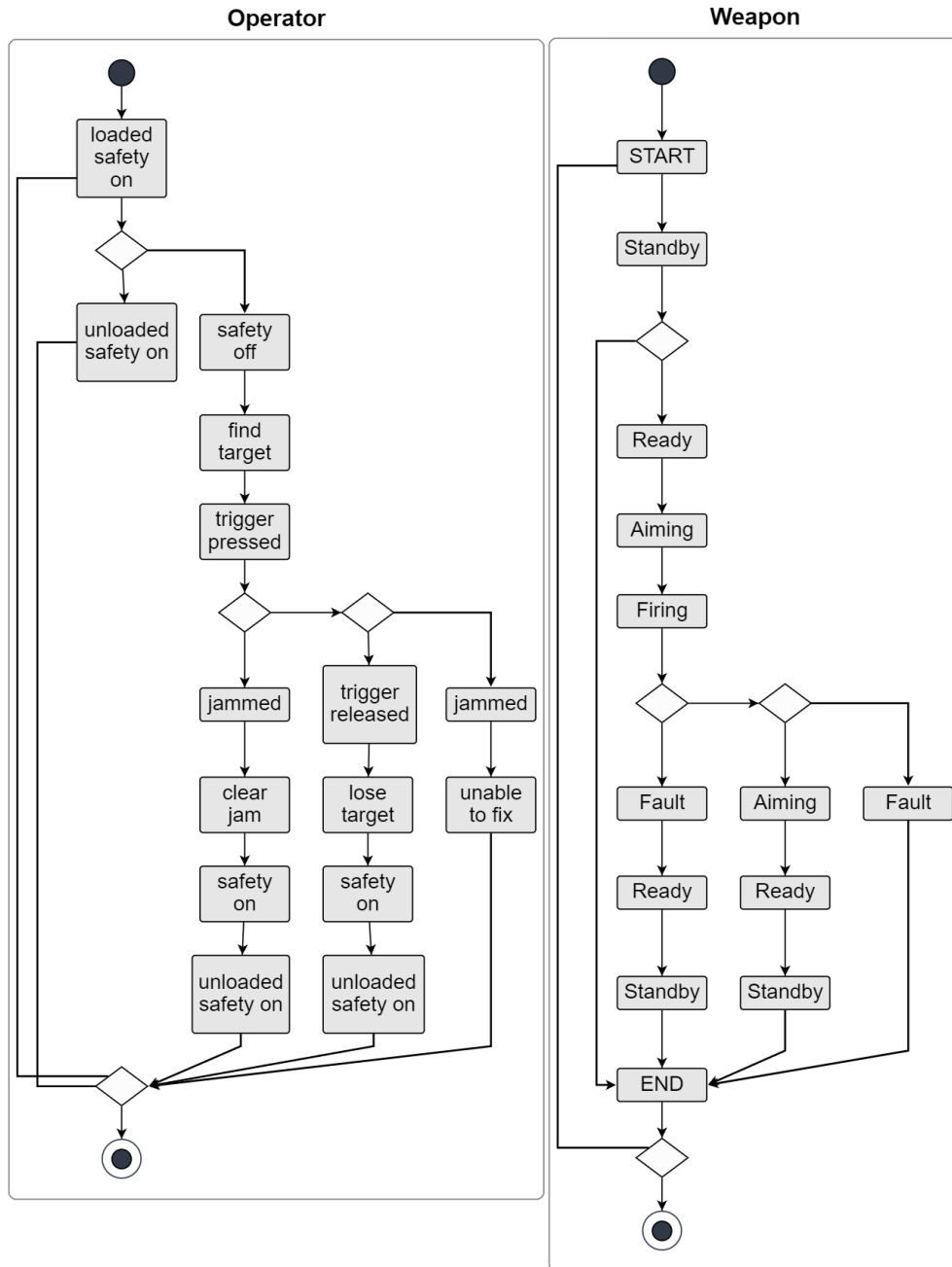



Figure 52. Firearm Safety Operational model activity diagram, illustrated by Monterey Phoenix

LIST OF REFERENCES

- Auguston, Mikhail. 2009. "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. OOPSLA 2009. Orlando, Florida.
- Auguston, Mikhail. n.d. "Example 32: Model of ATM Withdrawal with State Chart." <https://firebird.nps.edu/>
- Beaver, Joshua P. 2021. "Analyzing Emergent Behavior of Supply Chains for Personal Protective Equipment in Response to COVID-19." Master's thesis. Naval Postgraduate School. <http://hdl.handle.net/10945/68296>
- Delligatti, Lenny, 2014. SysML Distilled: A brief guide to the Systems Modeling Language. Addison-Wesley.
- Department of the Army. 2019. Training and Qualification – Individual Weapons. Washington, DC: Department of the Army. https://armypubs.army.mil/epubs/DR_a/pdf/web/ARN19574_TC_3-20.40-Incl_C1_FINAL_WEB.pdf
- Fakhroutdinov, K., 2022. Unified Modeling Language (UML) description, UML diagram examples, tutorials and reference for all types of UML diagrams - use case diagrams, class, package, component, composite structure diagrams, deployments, activities, interactions, profiles, etc. Uml-diagrams.org. Available at: <https://www.uml-diagrams.org/> [Accessed 21 July 2022].
- Giammarco, Kristin. 2022. "Exposing and Controlling Emergent Behaviors Using Models Within Human Reasoning." In Emergent Behavior in System of Systems Engineering: CRC Press Taylor & Francis Group: Boca Raton, FL, USA. pp. 23–61.
- Giammarco, Kristin and Len Troncale. 2018. "Modeling Isomorphic Systems Processes Using Monterey Phoenix." MDPI Systems. Systems 2018 6, 18.
- Giammarco, K; Carlson, C., Blackburn, M. 2018. "Verification and Validation (V&V) of System Behavior Specifications." Final Technical Report SERC-2018-TR-116; Systems Engineering Research Center: Hoboken, NJ, USA.
- Griffin, Edward and Russell Turner, Shawn Whitcomb, Donald Swihart, James Bier, Kerianne Hobbs, and Amy Burns. 2012. "Automatic Ground Collision Avoidance System Design for Pre-Block 40 F-16 Configurations." 2012 Asia-Pacific International Symposium on Aerospace Technology. Nov 13–16. Jeju, Korea.

- Lockheed Martin, n.d. “Automatic Ground Collision Avoidance System (Auto GCAS).”
<https://www.lockheedmartin.com/en-us/products/autogcas.html>
- Nambi, Karthick. 2020. “How a Simple Bug Paralyzed a U.S. Warship and a U.S. Fighter Jet.” Medium. Accessed August 17, 2022. <https://medium.com/predict/how-a-simple-bug-paralyzed-a-us-warship-and-a-us-fighter-jet-fa294e7fe2f8>.
- National Rifle Association of America, General Operations. 2022. “NRA Gun Safety Rules.” Accessed May 22, 2022. <https://gunsafetyrules.nra.org/>
- NPS Wiki. n.d. “About - Monterey Phoenix.” Access March 1, 2022.
<https://wiki.nps.edu/display/MP/About>
- Object Management Group (OMG) Systems Modeling Language (SysML). n.d. “What is SysML.” Accessed July 16, 2022. <https://www.omgsysml.org/what-is-sysml.htm>.
- Object Management Group. 2019. “OMG Systems Modeling Language.” November, 2019. <https://www.omg.org/spec/SysML/1.6CATiA> No Magic, Inc. 2020. Magic Systems of Systems Architect 2021x.
- Quartuccio, John. 2020. “Identification of Behavior Patterns In System-of-Systems.” PhD diss., Naval Postgraduate School, 2020. <https://calhoun.nps.edu/handle/10945/64902>
- Rowton, Amanda A. 2020. “Using Behavior Modeling to Enable Emergency Responder Decision-Making.” Master’s thesis. Naval Postgraduate School.
<http://hdl.handle.net/10945/66135>.
- Zdanis, Larry, Cloutier, Robert, Ph.D. 2007.”The Use of Behavioral Diagrams in SysML.” In 2007 IEEE Conference on Systems, Applications and Technology 108. https://www.researchgate.net/publication/4275660_The_Use_of_Behavioral_Diagrams_in_SysML.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California