



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**GRAPHICAL REPRESENTATION APPROACH
TO REASONING ABOUT COMPLEX SYSTEMS
BEHAVIOR FOR BROAD APPLICATIONS**

by

Christopher S. Ritter

September 2023

Thesis Advisor:
Second Reader:

Kristin M. Giammarco
Warren Vaneman

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 2023	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE GRAPHICAL REPRESENTATION APPROACH TO REASONING ABOUT COMPLEX SYSTEMS BEHAVIOR FOR BROAD APPLICATIONS			5. FUNDING NUMBERS	
6. AUTHOR(S) Christopher S. Ritter				
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) Model-based systems engineering (MBSE) has proven success in achieving significant, quantifiable defect reductions and cost savings across complex engineering systems. MBSE requires engineers to use formal languages to design complex systems, which often require organizations to invest in special training. Furthermore, key subject-matter experts and decision makers often do not know and are not expected to know a formal language well enough to reason using it. Monterey Phoenix (MP), a U.S. Navy-developed modeling language, approach, and tool, generates emergent system and process behaviors through its scope-complete scenario generation capability. Currently, MP users must code with a text-based language; however, many systems engineering practitioners commonly prefer working with graphical representations of models. This thesis answers the question: "To what extent can the MP language be represented graphically?" This research assesses prior work, selects a graphical representation, develops an open-source prototype software application, and maps to pertinent Systems Modeling Language (SysML) semantics. A graphical language approach that works with MP is expected to provide a more accessible method for systems engineers to reason about more complex system behaviors, providing a graphical gateway for formal methods novices to conduct scope-complete design and analysis and potentially leading to cost savings across the defense enterprise.				
14. SUBJECT TERMS model-based systems engineering, systems engineering, Monterey Phoenix, emergent behavior, digital engineering, MBSE, Systems Modelling Language, SysML			15. NUMBER OF PAGES 129	
			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	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**GRAPHICAL REPRESENTATION APPROACH TO REASONING ABOUT
COMPLEX SYSTEMS BEHAVIOR FOR BROAD APPLICATIONS**

Christopher S. Ritter
Civilian, Battelle Energy Alliance
BS, Virginia Polytechnic Institute and State University, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING MANAGEMENT

from the

**NAVAL POSTGRADUATE SCHOOL
September 2023**

Approved by: Kristin M. Giammarco
Advisor

Warren Vaneman
Second Reader

Oleg A. Yakimenko
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Model-based systems engineering (MBSE) has proven success in achieving significant, quantifiable defect reductions and cost savings across complex engineering systems. MBSE requires engineers to use formal languages to design complex systems, which often require organizations to invest in special training. Furthermore, key subject-matter experts and decision makers often do not know and are not expected to know a formal language well enough to reason using it. Monterey Phoenix (MP), a U.S. Navy-developed modeling language, approach, and tool, generates emergent system and process behaviors through its scope-complete scenario generation capability. Currently, MP users must code with a text-based language; however, many systems engineering practitioners commonly prefer working with graphical representations of models. This thesis answers the question: “To what extent can the MP language be represented graphically?” This research assesses prior work, selects a graphical representation, develops an open-source prototype software application, and maps to pertinent Systems Modeling Language (SysML) semantics. A graphical language approach that works with MP is expected to provide a more accessible method for systems engineers to reason about more complex system behaviors, providing a graphical gateway for formal methods novices to conduct scope-complete design and analysis and potentially leading to cost savings across the defense enterprise.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	BACKGROUND AND MOTIVATIONS	1
B.	RESEARCH QUESTION	2
C.	RESEARCH METHODOLOGY	2
D.	BENEFITS AND PURPOSE OF RESEARCH.....	4
E.	THESIS ORGANIZATION	4
II.	LITERATURE REVIEW	7
A.	MP EVENT GRAMMAR	7
	1. Event Decomposition and Flow	7
	2. Root Events and Parallel Logic	8
	3. Event Selection / OR Logic.....	9
	4. Event Iteration / Loop Logic.....	10
	5. Event Coordination.....	11
B.	FUNCTIONAL GRAPHICAL LANGUAGES.....	12
	1. Early Graphical Notations	13
	2. State Chart Graphical Notations.....	24
	3. Flow-Based Graphical Notations.....	27
	4. Sequential Graphical Notations.....	36
C.	CHAPTER SUMMARY.....	40
III.	OPEN-SOURCE DATA ANALYSIS.....	43
A.	GOOGLE TRENDS ANALYSIS	43
B.	GOOGLE SCHOLAR ANALYSIS.....	44
C.	SOCIAL MEDIA POPULARITY ANALYSIS.....	45
D.	GITHUB REPOSITORY ANALYSIS.....	47
E.	YCOMBINATOR ANALYSIS.....	49
	1. Methodology	49
	2. Results	54
F.	SYSTEMS ENGINEERING PROFESSIONAL ARTICLES ANALYSIS	58
G.	CHAPTER SUMMARY.....	59
IV.	CASE STUDY REPRESENTATIONS AND DISCUSSION.....	61
A.	WIEVLE PROBLEM OVERVIEW	61
B.	MP GRAMMAR REPRESENTATION.....	61

C.	FLOWCHART REPRESENTATION.....	64
D.	IDEF0 REPRESENTATION.....	67
E.	EFFBD REPRESENTATION	68
F.	UML SEQUENCE DIAGRAM REPRESENTATION	70
G.	SYSML/UML REPRESENTATION	72
H.	BPMN REPRESENTATION.....	76
I.	LML ACTION DIAGRAM REPRESENTATION	78
J.	CHAPTER SUMMARY.....	80
V.	MP GRAPHICAL LANGUAGE.....	81
A.	LANGUAGE SELECTION	81
B.	SOFTWARE FUNCTIONAL ARCHITECTURE	83
C.	SYSML IMPLEMENTATION TO MP MAPPING	87
1.	Root Events.....	87
2.	Atomic Events.....	87
3.	Decomposition and Flow	88
4.	Parallel Logic.....	89
5.	Event Selection / OR Logic.....	89
6.	Coordination.....	90
7.	Loop and Iteration	91
8.	MP Compiler Integration.....	93
D.	CHAPTER SUMMARY.....	93
VI.	CONCLUSION AND FUTURE WORK	95
A.	CONCLUSION	95
B.	FUTURE WORK.....	96
1.	Activity Diagram Decomposition.....	96
2.	Seabird Usability Study.....	96
3.	Seabird Modeling Validation Assistant	96
4.	Natural Language MP	97
	LIST OF REFERENCES.....	99
	INITIAL DISTRIBUTION LIST	107

LIST OF FIGURES

Figure 1.	MP Composite Event Example.....	8
Figure 2.	MP Sequential Example.....	8
Figure 3.	MP Parallel Example	9
Figure 4.	MP OR Logic Example.....	10
Figure 5.	MP Loop Logic Example.....	11
Figure 6.	MP Coordination Example	12
Figure 7.	Process Chart Representation of Loading Rifle Grenade. Source: Gilbreth and Gilbreth (1921).	14
Figure 8.	Process Chart Representation of Parallel Paths. Source: Gilbreth and Gilbreth (1921, 15).....	15
Figure 9.	Example Petri Net Demonstrating Early Formal Logic. Source: Chen and Lee (1992, 44).	16
Figure 10.	Original Standard Process Chart Symbols from 1947 with Little Resemblance Today. Source: The American Society of Mechanical Engineers (1947, 3).....	17
Figure 11.	An Early Flowchart Template Demonstrating Common Symbols from the 1950s, Still Recognizable. Source: Smithsonian (n.d.).	18
Figure 12.	FFBD Example from Vitech’s Well-Known Implementation of the Representation. Source: Vitech Corporation (n.d.).....	19
Figure 13.	Example N2 Chart Demonstrating Item Flow between Functions. Source: FAA (2006, 4.4-12).	20
Figure 14.	Example SADT Diagram Which Preceded the IDEF0. Source: U.S. Department of Commerce (1976, 5-29).....	21
Figure 15.	SREM R_Net Example Used in DOD’s First MBSE Tool, DCDS. Source: Alford and Lawson (1979).....	23
Figure 16.	Example IDEF0 Formalized from SADT. Source: Syque (n.d.).	25
Figure 17.	Example Harel State Chart Demonstrating State Decomposition. Source: Harel (1987).....	26

Figure 18.	Example UML State Machine with Guarded ORs and Sub-Machines. Source: Sparx Systems (n.d.).	27
Figure 19.	Example EFFBD Representation Adding Item Flow to FFBDs. Source: Long (2000).	29
Figure 20.	BPMN Collaboration Diagram. Source: OMG (2010).	30
Figure 21.	BPMN Choreography Diagram. Source: OMG (2010).	31
Figure 22.	LML Action Diagram with All Constructs. Source: Lifecycle Modeling Language (2015).	32
Figure 23.	UML 2 Activity Diagram Closely Inspired from EFFBD Logic. Source: Bock (2003, 169).	34
Figure 24.	SysML 1.x Activity Diagram Demonstrating Commonality with UML 2's Representation. Source: No Magic (n.d.).	35
Figure 25.	Message Sequence Chart Example Which Inspired the Modern Sequence Diagram. Source: International Telecommunication Union (1994, 22).	37
Figure 26.	Example UML 2.x Sequence Diagram Demonstrating Interaction Flow Between Actors. Source: UML Diagrams (n.d.).	38
Figure 27.	Example LSC. Source: Damm and Harel (2001).	40
Figure 28.	Functional Graphic Language History	42
Figure 29.	Google Trends Analysis Chart Demonstrating Flowchart's Popularity Dominance	44
Figure 30.	Google Scholar Popularity Chart Again Demonstrating Flowchart's Overwhelming General Popularity	45
Figure 31.	Social Media Mention Chart Demonstrating Flowchart's General Popularity	46
Figure 32.	Social Media Sentiment Analysis Showing Positive Sentiment for All Representatives Except Flowcharts	47
Figure 33.	GitHub Graphical Language Chart Demonstrated Dominance by UML for Software Graphical Representation	49
Figure 34.	MP Activity Diagram of NLP and NLU Analysis Approach	54
Figure 35.	Number of Comments by Diagram Topic Type	55

Figure 36.	YCombinator Sentiment Analysis Demonstrating Low Favor for UML Among Software Developers	56
Figure 37.	Flowchart Word Cloud with Positive Words Commonly Appearing	57
Figure 38.	UML Word Cloud Demonstrating Only One Negative Word, “Hard”	57
Figure 39.	Systems Engineering Professional Articles Analysis	58
Figure 40.	WIEVLE Scenario Flowchart Diagram	64
Figure 41.	WIEVLE Scenario Sequential Flowchart Diagram	66
Figure 42.	WIEVLE Scenario IDEF0 Diagram	67
Figure 43.	WIEVLE Scenario EFFBD Diagram.....	69
Figure 44.	WIEVLE Scenario Sequence Diagram	71
Figure 45.	WIEVLE Scenario SysML/UML Interrupting Region Diagram.....	73
Figure 46.	WIEVLE Scenario fUML-Compatible SysML/UML Diagram	75
Figure 47.	WIEVLE Scenario BPMN Diagram	77
Figure 48.	WIEVLE Scenario LML Action Diagram.....	79
Figure 49.	Seabird Functional Context IDEF0.....	85
Figure 50.	Open Seabird Functional Architecture IDEF0.....	86
Figure 51.	Root Event Example in Seabird.....	87
Figure 52.	Atom Action Example in Seabird	88
Figure 53.	Sequential Action Flow in Seabird	88
Figure 54.	Parallel Logic in Seabird.....	89
Figure 55.	Event Selection Example in Seabird.....	90
Figure 56.	Optional Event Example in Seabird.....	90
Figure 57.	Coordination Example Using Input/Outputs in Seabird	91
Figure 58.	Zero or More Loop Example in Seabird	92
Figure 59.	One or More Loop Example in Seabird.....	93

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Google Scholar Popularity Data (in hundreds).....	44
Table 2.	GitHub Graphical Language Data	48
Table 3.	Initial Language Selection Table	82
Table 4.	Final Language Selection Table.....	83

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASME	American Society of Mechanical Engineers
BPMI	Business Process Management Initiative
BPML	Business Process Modeling Language
BPMN	Business Process Modeling Notation
BPQL	Business Process Query Language
CMMN	Case Management Model and Notation
DMN	Decision Model and Notation
DOD	Department of Defense
DRD	Decision Requirements Diagram
DSM	design structure matrix
EFFBD	Enhanced Functional Flow Block Diagrams
FFBD	Functional Flow Block Diagrams
ICAM	Integrated Computer-Aided Manufacturing
ICOM	input, control, output, mechanism
INCOSE	International Council on Systems Engineering
ITU	International Telecommunications Union
LML	Lifecycle Modeling Language
LSC	Live Sequence Chart
MBSE	model-based systems engineering
MP	Monterey Phoenix
MSC	Message Sequence Chart
NIST	National Institute of Standards and Technology
NLP	natural language processing
NLU	natural language understanding
NPS	Naval Postgraduate School
OMT	Object Modeling Technique
SADT	Structured Analysis and Design Technique

SREM	Systems Requirements Engineering Methodology
SysML	Systems Modeling Language
UML	Unified Modeling Language
UUV	unmanned underwater vehicle

EXECUTIVE SUMMARY

Naval Postgraduate School (NPS) provides a tool, Monterey Phoenix (MP), with the ability to generate scope-complete sets of scenarios. Scope-complete behavior modeling is a systems modeling technique that can be used to discover emergent and unexpected behaviors of a system. Existing MP tools require users to write in a computational text-based language and do not include a graphical notation for composing MP models. This thesis explores which aspects of MP event grammar rules and constraints can be represented through graphical semantics via the study of historical languages, derivation of usability heuristics, study of language adoption, selection of a graphical language, and automated translation from a graphical to computational language.

Graphical modeling languages for systems engineering have been in development for over 100 years, from the first flow chart symbology to the second major iteration of Systems Modeling Language (SysML). In the 1920s, Frank and Lillian Gilbreth invented “Process Charts” to formalize functional behavior modeling for the first time (The Gilbreth Network n.d.). “Process Charts” were later formalized as the “Operation and Flow Process Charts” standard in 1947 (The American Society of Mechanical Engineers 1947). A decade later, a team at TRW, Inc., through the influence of the ASME specification, developed the Functional Flow Block Diagram (FFBD) (Aizier et al. 2012, 2). The FFBD and its decedents ultimately directly influenced SysML 1, Unified Modeling Language (UML) 2, and today’s SysML 2. Every graphical language has demonstrated heavy influence from the languages which proceeded it, developing small tweaks at each historical iteration.

An analysis across Google Search results, Google Scholar results, and social media found that, for general applications, the flowchart remains the most popular behavioral modeling language. However, UML has a similar lead for formal modeling applications. Each of these languages was studied with an unmanned underwater vehicle (UUV) case study to prove a language’s ability to depict and simulate common behavioral constructs. Flowcharts were found to be too informal for transforming graphical language to formal semantics (i.e., transpile), while generally the SysML activity diagram, UML activity

diagram, and Enhanced Function Flow Block Diagram (EFFBD) include sufficient formalism for model simulation.

Generally, across common systems engineering modeling languages, a three-way tie was reached across common selection criteria such as expressiveness, popularity, and executability. Ultimately, SysML and UML were combined into a single criteria category, lifting the SysML / UML category above the others. It is important to note that SysML and UML are not necessarily superior to other languages but were adequate as a first graphical language for MP transpilation.

A software application, Seabird, was developed to provide both SysML activity diagram drawing capability and a transpiler to MP code. Seabird uses swimlanes as the graphical construct to represent MP schema roots. Similarly, atomic events are represented by action SysML symbols. The concept of event precedence is supported through SysML's concept of precedence. Likewise, parallel logic uses SysML fork and join symbols. The loop, selection, and optional logical operations utilize the decision and merge symbols from SysML. Lastly, coordination between events uses item flow relationships between activities across separate swimlanes.

Seabird can be used by Navy and academic users alike to discover scenario side effects previously undiscoverable to the modeling community. This research demonstrates, using Seabird, the inception of automatic MP code generation from SysML activity diagram symbols. Seabird is a new open-source tool, derived from the technology behind diagrams.net, that offers graphical development of MP diagrams. Seabird learns from prior modeling languages and tools, presenting a horizontally focused activity layout, with symbols familiar to flowchart users. Modelers can use the root event, atomic event, alternate event selection, parallel unordered sets of actions, and both loop event types from MP. Additionally, with SysML input/output flow in Seabird, the MP coordinate statement can be automatically generated from graphical input. Seabird broadens the possible user base for scope-complete behavior modeling, widening the opportunity for increased understanding of emergent behavior across the U.S. defense industrial base.

References

- The American Society of Mechanical Engineers. 1947. *Operation and Flow Process Charts*. New York: ASME. <http://catalog.hathitrust.org/Record/005735891>.
- Aizier, Bruno, Stéphanie Lizy-Destrez, Charlotte Seidner, Chapurlat Vincent, Daniel Pru, and Jean-Luc Wippler. 2012. “xFFBD: Towards A Formal Yet Simple and Complete Functional Modeling Technique for System Designers.” In *22nd Annual International Council on Systems Engineering Symposium*. <https://hal-enac.archives-ouvertes.fr/hal-01022483>.
- The Gilbreth Network. n.d. “Biography.” Accessed December 21, 2021. <https://gilbrethnetwork.tripod.com/bio.html>

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND AND MOTIVATIONS

Model-based systems engineering (MBSE) has demonstrated success in reducing the introduction of silent errors across the engineering enterprise. U.S. Department of Defense (DOD) contractor Raytheon’s implementation of MBSE “reduced specification defects by 68%” (Cook and Wilson 2015, 41). Westinghouse demonstrated a 70% cost savings in the verification phase for railway switching systems through MBSE (Cook and Wilson 2015, 41). While these defect improvements and cost savings demonstrate progress, the “affordable transition to [a] model-based approach” is a key challenge of MBSE adoption (Madni and Purohit 2019, 17), lengthening transition time to realize DOD’s *Digital Engineering Strategy* (U.S. Department of Defense [DOD] 2018).

Throughout history several graphical modeling languages have been developed to capture behavior. The first familiar representation was designed in 1921 (The Gilbreth Network n.d.) as process charts (predecessor to flowcharts) and the most recent SysML 2.x representation is under draft design today in 2022. Many modifications have been made to behavioral graphical notations over time, but it is not clear if these graphical language modifications have increased adoption, usability, or education of behavior modeling.

The motivation of this thesis is to discover new capabilities to represent a scope-complete text language using graphical techniques. This research aims to provide more approachable formal modeling semantics to the systems engineering community to increase use and adoption of scope-complete modeling techniques. As various systems and sub-systems are combined to perform a mission, emergent behavior can occur, which can only be discovered when the system or system of systems is analyzed systematically as a whole. Scope complete modeling generates a set of traces that discover possible intended and unintended system behaviors. The use of scope complete modeling aids a user to discover scenarios which can happen and conversely will not happen in a scenario of interest (NPS Wiki n.d.b). A 1990 study by Warren Keuffel found system design issues discovered in the requirements phase are 4,000–100,000% less expensive than errors

discovered during systems operations (Keuffel 1990; Mekki, Ghazel, and Toguyéni 2011). Furthermore, each phase with an error left undiscovered increased costs “steep[ly]” (Boehm, Valerdi, and Honour 2008, 5). This increased adoption is expected to reduce complex system costs and schedule through defect detection earlier in the design process.

B. RESEARCH QUESTION

Which elements of the Monterey Phoenix (MP) language could be represented graphically to help the systems engineering community adopt scope-complete modeling methods and analysis?

To answer this question, five decomposed questions are below:

- How did we evolve to the state of graphical languages today? To date, no study has comprehensively analyzed the connections between languages since the 1920s.
- What usability heuristics can we learn from prior language specific research in the practice?
- What is the current state of systems engineering graphical language adoption? Is the market share of these languages growing, declining, or flat overtime? Are the languages by in large liked or disliked by the community?
- Is a whole-new graphical language needed to represent MP or can modifications be made to existing languages to support scope-complete modeling?
- How can a system translate from pictorial boxes and lines to correct MP code without significant compiler errors?

C. RESEARCH METHODOLOGY

This research develops and tests a graphical language of MP based on knowledge gained from a study of the evolution of graphical languages. This research utilizes a

comprehensive literature review, open-source (secondary) data analysis from public practitioner feedback, an integrated analysis of literature and open-source data, a graphical MP language derived from these sources, a mapping to Systems Modeling Language (SysML), and a prototype software implementation of the graphical language. Each phase will perform the following process:

- **Literature Review:** Collect prior research on behavioral systems engineering representations. This forms a comprehensive data compilation from early designs through current representations. The emphasis of this review is on design evolution of functional flow/behavior diagrams, state diagrams, and sequence diagrams.
- **Open-Source Data Analysis:** The secondary data on engineers' use of systems engineering languages is sourced from Y Combinator Hacker News comments and GitHub project information. Y Combinator data is analyzed using keyword extraction, natural language processing of adjectives, and sentiment analysis of user opinions through pretrained machine learning models. Social media analysis (Social Mention) and search analysis (Google Trends) are also used to discover data patterns.
- **Integrated Representation Analysis:** Data from the literature review is correlated to open-source data analysis information to process user feedback with each representation. This produces a quantitative map of representations, including a formula to score each representation utilizing sentiment data, popularity data, and executability/formalization data.
- **Derived MP Graphical Language:** The result of the integrated data analysis identifies the strongest candidates to derive a graphical language for MP. Each MP formal language semantic is mapped to the developed graphical language with a gap analysis.

- **MP Graphical Language / SysML Mapping:** The research develops a mapping to SysML version 2 representations to analyze differences between approaches.
- **Prototype Software Implementation:** The graphical language is instantiated in a prototype tool environment. Software code implements a graphical user interface to create diagrams in MP and a backend data layer to store the representations. The interface provides code generation to MP's formal language specifications.

D. BENEFITS AND PURPOSE OF RESEARCH

The complexity of aircraft in the U.S. defense enterprise has increased by ~1,000% through each generation. This pattern has been relatively consistent since the 1960s and is expected to continue with future aircraft (West 2017, 48). Additionally, in 2017, Congress directed the Secretary of Defense to “establish mission integration management activities for each mission area” (U.S. House of Representatives 2016, 300). This creates an increasing need for DOD to operate more efficiently and effectively in the future. This new MP graphical representation aims to reduce risk and increase adoption of MBSE techniques for complex system design and mission integration planning.

The systems engineering community primarily uses Unified Modeling Language (UML) and SysML graphical representations for behavioral analysis (Google Trends n.d.). These languages do not currently support scope-complete generation and analysis of behavioral models.

The goal of this research is to define, implement, and test a formal graphical language to represent MP event grammars. A new graphical representation bridges the existing modeling preferences with scope-complete analysis.

E. THESIS ORGANIZATION

This chapter introduces the basis for this thesis, methodology, and potential benefits of research in graphical modeling languages to support MP. The next chapter chronologically dives into each relevant functional modeling language, pointing out key

attributes of each while discussing possible omissions. The third chapter explores the current popularity and sentiment of each graphical language, to discover with data science principles the current landscape. The fourth chapter uses a case study to map a selection of functional graphic languages to MP code. The fifth chapter discusses a selection methodology, resulting graphical MP language, and results from testing this language. The sixth and final chapter discusses conclusions drawn from this research.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LITERATURE REVIEW

The development of a graphical language to support MP requires mapping of MP features to common graphical elements, understanding of existing and historical graphical languages, and the interaction between languages historically. This chapter begins with a discussion on MP's current code representation and follows with a historical analysis of early graphical notations, state charts, flow-based languages, and sequential representations. A chapter summary documents a graphical history of all languages discussed in this chapter.

A. MP EVENT GRAMMAR

The MP language is represented by a single type of object, events. MP includes several logical constructs to relate events. This section discusses decomposition, flow, parallelism, selection, loop iteration, and coordination.

1. Event Decomposition and Flow

MP includes two primary relations: inclusion and precedence. These two relations unlock many of MP's core functionality for event expression needed for almost all behavior modeling problems (NPS Wiki n.d.b).

Events may be decomposed into a hierarchy and "included" within other events. For example, given the MP code: "**ROOT** Hungry_Hippo: make_dinner; make_dinner: prepare_dinner cook_dinner," hungry hippo is decomposed by make_dinner which is in turn decomposed by both prepare and cook dinner as seen in Figure 1. Root events are top-level events indicated by the root keyword, while composite events are events beneath a root event or another composite event with associated child events. Root events often represent performers of events within MP but may also represent top-level processes, states or time phases as illustrated by the many example models available on MP-Firebird at <https://firebird.nps.edu>.



Figure 1. MP Composite Event Example

Precedence is a finish to start relation in which the prior event must complete prior to execution of the successor event. For example, given the MP code: “`ROOT Hungry_Hippo: make_dinner eat_dinner`,” dinner must be made prior to eating dinner as seen in Figure 2. This can be expanded from two events to as many events as needed using the relation.

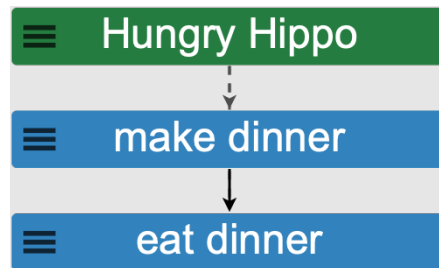


Figure 2. MP Sequential Example

2. Root Events and Parallel Logic

Each root performer event logically operates in parallel within its own thread. Root events allow for each performer to be modeled separately and is a better reflection of the real world. For example, consider two players of the game tic-tac-toe: Hungry Hippo and Mary. Modeling Hungry Hippo and Mary within the same root event, effectively as just a simple sequence of events, may not reflect a real-world model. What if Hungry Hippo asks Nancy for advice on her next move? What if Mary quits mid-game? The questions are more accurately modeled with each root event in its own separate thread, which clarifies who performs an event.

To model parallelism within a root or composite event, MP includes the means to express unordered sets. For example, given `“make_dinner: prepare_dinner { cook_dinner, exercise },”` prepare dinner must occur before the cook dinner and exercise; however, as seen in Figure 3, `cook_dinner` and `exercise` occur at the same time, even though they are performed by the same root event (NPS Wiki n.d.b).

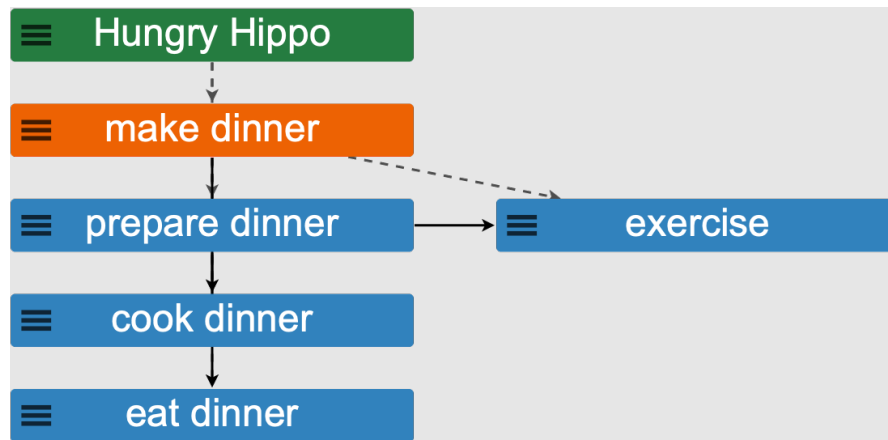


Figure 3. MP Parallel Example

3. Event Selection / OR Logic

Selection decisions within MP have two forms: alternatives and optional events. Alternatives are an exclusive “OR” and allow choosing a single event from a set of events. For example, given `“ROOT Hungry_Hippo: (make_dinner | order_takeout) eat_dinner,”` the hippo will either choose the make dinner or order takeout alternative. Notably, the parentheses in this example introduce a second concept, event order of operations. The alternative event is ordered together, while `eat_dinner`, not within parentheses occurs after this decision. A final concept of selection decisions is optional events. What if the hippo sometimes eats sushi and does not cook her meal? Given the MP code: `“make_dinner: prepare_dinner [ cook_dinner ],”` cook dinner is now an optional event and accounts for the hippo’s preference for sushi, sometimes, as seen in Figure 4 (Giammarco and Troncale 2018).

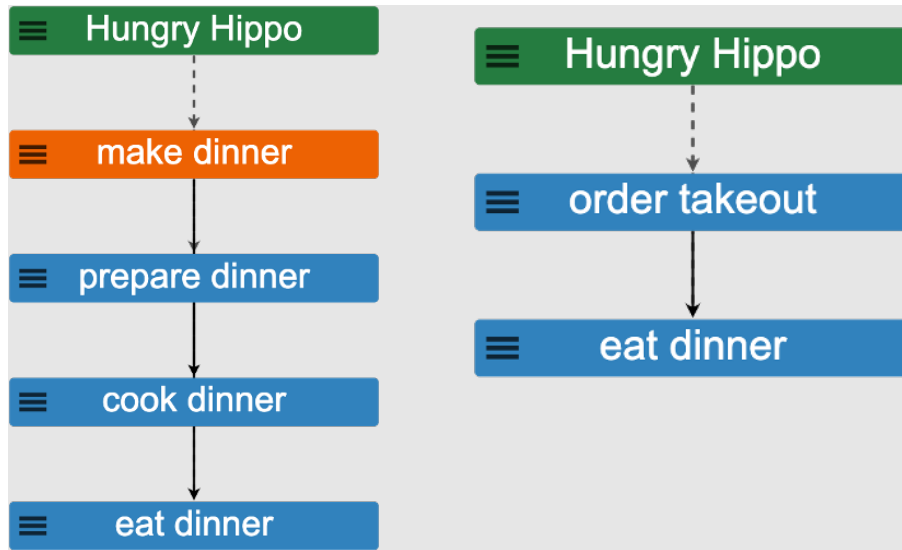


Figure 4. MP OR Logic Example

4. Event Iteration / Loop Logic

Events may occur more than once; for example, hungry hippo likely eats meals multiple times per day. Given the MP code, “`ROOT hungry_hippo: (* eat_meal *)`,” the Hungry Hippo can eat a meal zero or more times per day as seen in Figure 5. This syntax expresses that events within `(*)` occur zero or more times. What if Hungry Hippo always has at least one meal every day? The MP code “`ROOT hungry_hippo: (+ eat_meal +)`” allows for an event to occur one or more times; this way, the hippo always gets at least one meal (NPS Wiki n.d.a). Figure 5 demonstrates a single example trace of running this code at scope three. Scope controls the depth of model execution and limits scenario generation to a finite set of cases (Giammarco and Troncale 2018, 4–5).

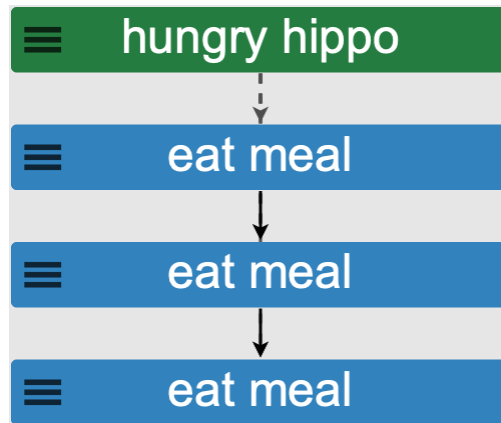


Figure 5. MP Loop Logic Example

5. Event Coordination

Recall the `cook_dinner` example from earlier. Hungry Hippo and the oven would be best modeled as two separate events as they are separate performers. Consider the MP code: `"ROOT Hungry_Hippo: set_oven_temperature put_food_in_oven; ROOT Oven: reach_temperature;"`; in this model, the hippo's steps never interact with the oven. The hippo must set the oven prior to the oven reaching temperature, and the food must wait for the oven to reach temperature prior to being placed in the oven as seen in Figure 6. To implement this logic, a coordination constraint is needed to allow for the hippo's setting of temperature to occur before the oven reaches temperature. Similarly, a second coordination statement can constrain the food to go in the oven after the oven has reached temperature (Giammarco and Troncale 2018).

```

SCHEMA loop_example

ROOT Hungry_Hippo: set_oven_temperature put_food_in_oven;

ROOT Oven: reach_temperature;

COORDINATE $x: set_oven_temperature FROM Hungry_Hippo,
            $y: reach_temperature FROM Oven

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: reach_temperature FROM Oven,
            $y: put_food_in_oven FROM Hungry_Hippo

DO ADD $x PRECEDES $y; OD;

```

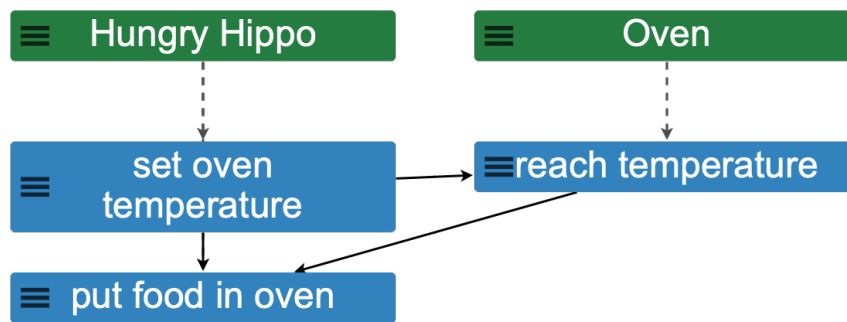


Figure 6. MP Coordination Example

Additional language elements are available within MP, including predicate calculus logic with “ENSURE” statements, BUILD blocks, and event sharing with the “SHARE ALL” command. To demonstrate a fully working interface, these language elements are out of scope for this thesis.

B. FUNCTIONAL GRAPHICAL LANGUAGES

The state of functional graphical languages has evolved over approximately 100 years of research. This section presents a historical analysis of each functional graphical representation, which obtained significant historical relevancy to the systems engineering discipline.

1. Early Graphical Notations

Early graphical notations inspired all modern graphical functional diagramming languages. This subsection reviews the most impactful early diagrams and points out primitive concepts that can be recognized in modern languages.

a. Process Charts

In 1921, the American Society of Mechanical Engineers (ASME) hosted an annual meeting to present “Process Charts” as a more efficient way to perform work. This is the first documented use of functional diagramming to capture a set of steps. The married inventors, Frank and Lillian Gilbreth, were interested in the “one best way” to perform work. Earlier in his career, Frank analyzed bricklaying methods to optimize bricklaying performance. Notable inventions were a new scaffolding system for optimal vertical adjustment to the current bricklayer, improved brick and mortar placement, and work performance with lower-priced labor stacking bricks (The Gilbreth Network n.d.). His work improving bricklaying inspired a commitment to furthering the engineering and management field.

Process charts were developed to visualize a process, improve the process through graphical representation, and present the process in its entirety. A key focus of process charts was on simplicity to promote their availability and understanding to the largest group of people as possible. For U.S. government defense applications, the Gilbreths noted that if government forms numbered each blank space and developed correlated instructions for filling in these forms, “the resulting savings would be astonishing (The Gilbreth Network n.d.).” Formalizing processes with process charts could improve readability, communication, and accuracy of complex government procedures.

Process charts included a set of standard symbology to express components, quality inspections, and operations as seen in Figure 7. Components of process charts were represented at the start of a process chart, in a vertical swimlane orientation, to serve as swimlanes to connect inspections and operations. Operations are identified by numbered circles.

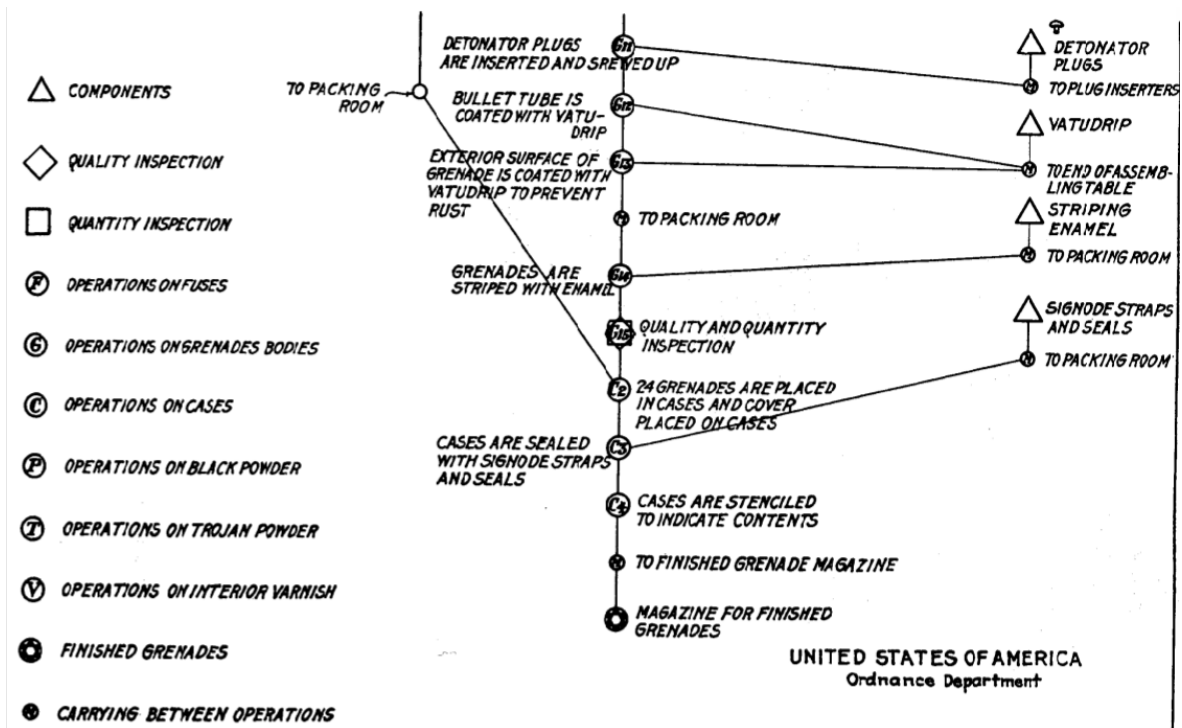


Figure 7. Process Chart Representation of Loading Rifle Grenade. Source: Gilbreth and Gilbreth (1921).

Parallel operations were presented implicitly with multiple lines forking from a predecessor operation and joining at a later successor operation as seen in Figure 8. Coordination between component swimlanes is accomplished through simple directly connected linear lines between operations. No arrows are utilized to indicate directionality as this is assumed implicitly from the process chart's text. There is no standard form to specify the input/output transferred between operations. The numbers on the diagram are placed to the left of textual labels. Each number refers to the performer of the action, while the label indicates the action performed. For example, 10 refers to the bidders, who perform "make [an] estimate," while both 8s refer buyers who "send a quotation" in the first occurrence and make a buying decision in the second occurrence (Gilbreth and Gilbreth 1921, 15).

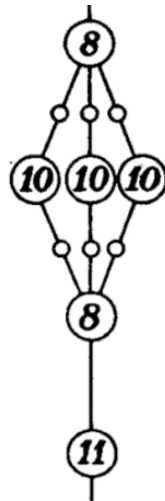


Figure 8. Process Chart Representation of Parallel Paths. Source: Gilbreth and Gilbreth (1921, 15).

Notably, in this first iteration, modern logical constructs are not represented. There exists no concept of selection or alternate paths nor a representation of looping/iteration flow. The decision diamond itself is, however, represented for the first time as a quality inspection gate, as seen earlier in Figure 7, “Quality Inspection.” Similarly, a quantity inspection gate is visualized by a square in the process chart standard symbology. The inspiration in modern flow representations from process charts is evident in this early form, although this first visual chart misses key modern behavioral logic elements.

b. Petri-Nets

Petri nets were first developed in 1939 by Carl Adam Petri when he was just 13 years old. Twenty-three years later, in 1962, Petri published his doctoral dissertation to document and mathematically formalize the notation (Petri and Reisig 2008). The original Petri net consists of a directed graph with nodes representing states (circles) and edges with rectangles representing transitions (Van der Aalst 1994, 46). A simple Petri net is demonstrated in Figure 9 with s1 representing a place and t1 a transition action.

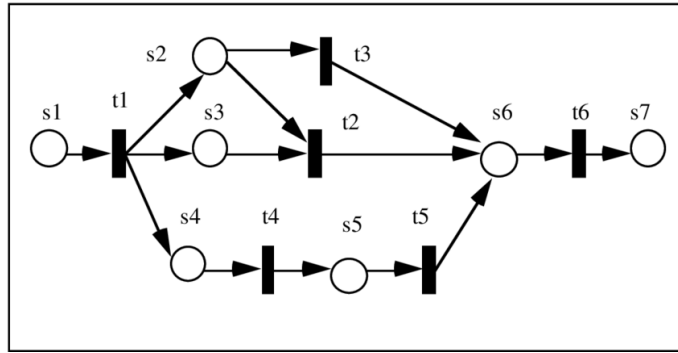


Figure 9. Example Petri Net Demonstrating Early Formal Logic. Source: Chen and Lee (1992, 44).

Several extensions have been made to the original Petri nets specification; for example, concurrency support was added which uses tokens that hold a strongly typed data value. Petri nets inspired modern UML Activity Diagrams with “token driven semantics” (Bedini, Maschotta, Wichmann, and Zimmermann 2019, 298).

c. *Flowchart Standards*

Over 25 years after the Gilbreths first process charts presentation, ASME published an “Operation and Flow Process Charts” standard in 1947 (The American Society of Mechanical Engineers 1947). This standard, while an inspiration for modern representations, still was distant from current symbology, as seen in Figure 10. A circle represented operations while a square represented an inspection. Interestingly, inspections can be thought of as a form of a decision gateway in modern representations; however, no ability to select branches nor loop existed in this early symbology. Special symbols for transportation (an arrow) and delays (a D) were also present in this early standard.

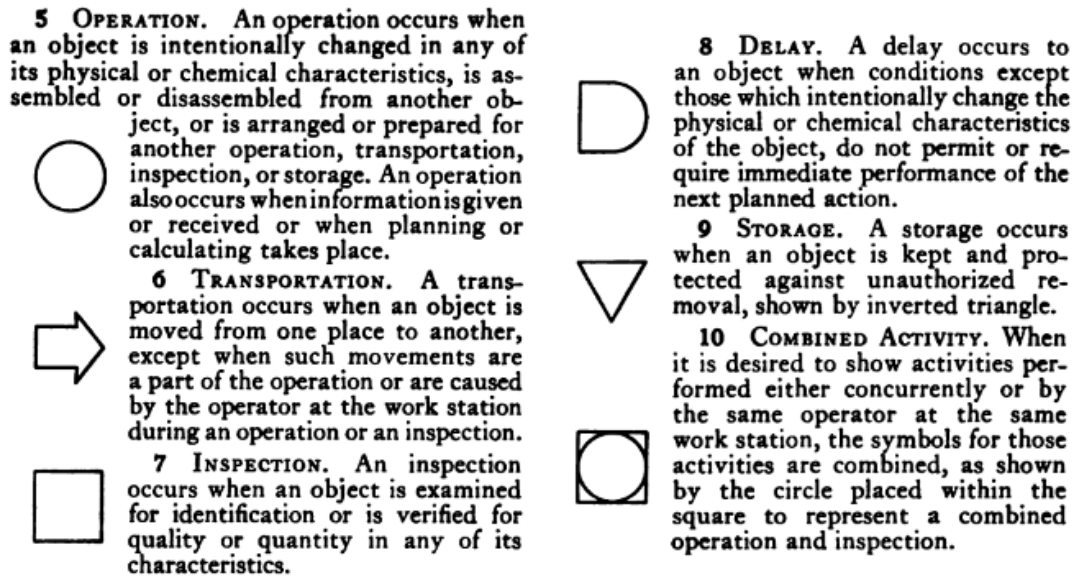


Figure 10. Original Standard Process Chart Symbols from 1947 with Little Resemblance Today. Source: The American Society of Mechanical Engineers (1947, 3).

John von Neumann developed the today-familiar flowchart symbology. Grace Hopper taught this representation to early programmers at Eckert-Mauchly Computer Corporation (Ensmenger 2016, 327). Interestingly, the flowchart symbology (Figure 11) is present in flowcharting templates developed after World War II (Smithsonian n.d.). These templates included a parallelogram to represent input/output, a square to depict a process, and a diamond to indicate a decision. These early flowcharts were utilized to program the first mainframe computers at the time. The 1970 American National Standard Institute published Federal Information Processing Standard 24, which officially standardized the symbology originally used in 1950s mainframe computer programming development (ANSI 1971). Remarkably, the flowcharting symbology present in popular diagramming tools today remains almost unchanged from the original 1950s derived notation (Microsoft n.d.).

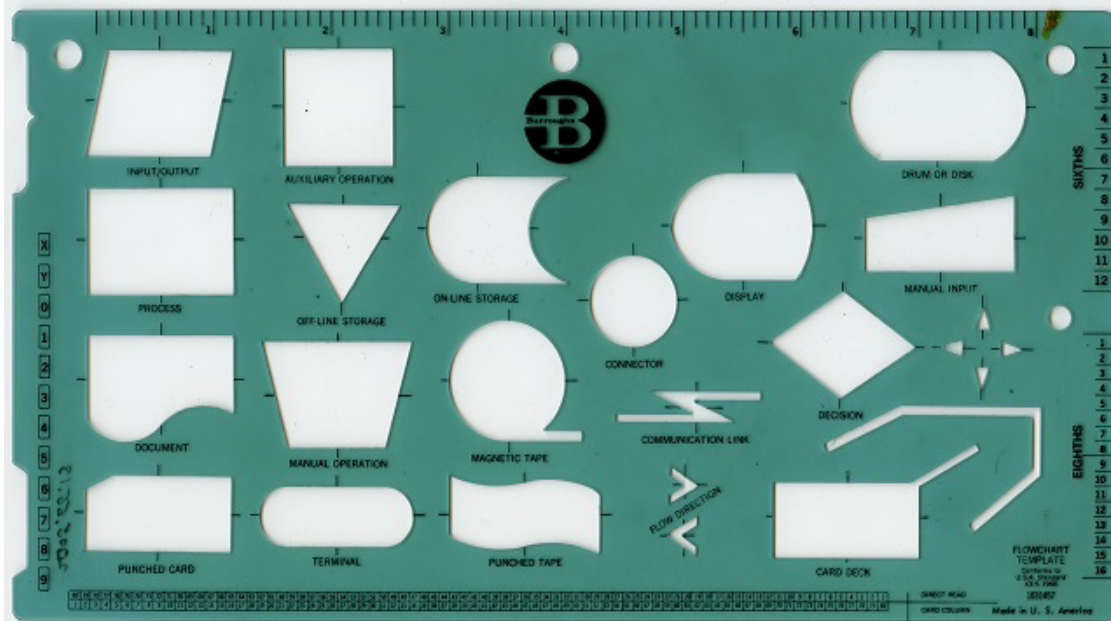


Figure 11. An Early Flowchart Template Demonstrating Common Symbols from the 1950s, Still Recognizable. Source: Smithsonian (n.d.).

d. *Functional Flow Block Diagrams (FFBD)*

In the 1950s, TRW, Inc. developed the FFBD notation, which became a frequently used tool in the systems engineering community (Aizier et al. 2012, 2). NASA utilized FFBDs to represent space systems and mission series of events and their corresponding control flow (Wikipedia 2023). Figure 12 represents an FFBD from Vitech CORE®, a commonly used systems modeling tool for drawing FFBDs (Vitech Corporation n.d.). This FFBD includes iteration (IT), loop (LP) parallel (AND), selection (OR), and multi-exit function control nodes. Iteration supports a finite number of executions of functions within the construct, while loops occur until an exit condition is met; this is seen with the customer and kitchen branches. Parallel (AND) is used to convey concurrent paths and can be used to act as swimlanes to represent performers of the functions. Parallel is visualized with the customer, service area, svc. customer, eating, and kitchen branches. A multi-exit function is represented by eat-in that allows eat-in to serve as a named OR node. FFBDs support both named multi-exit functions ORs and unnamed OR selection. Notably, FFBDs do not support representation of data. This data capability is discussed in Enhanced Functional Flow Block Diagrams (EFFBD).

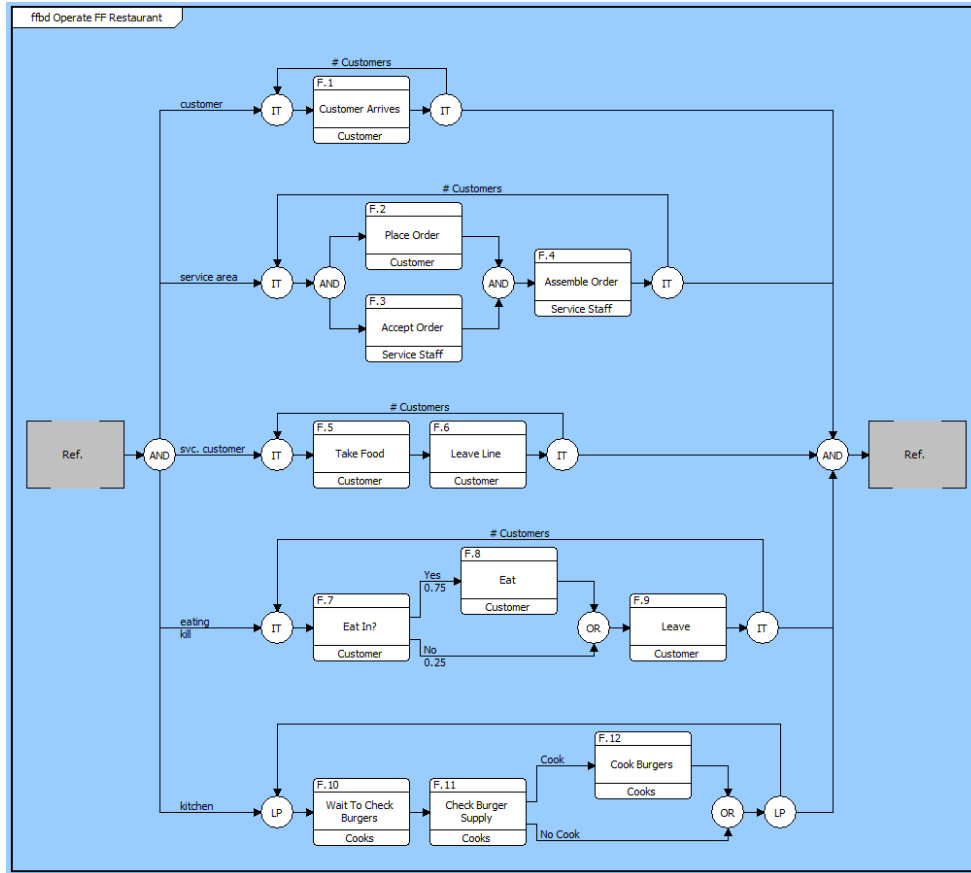


Figure 12. FFBD Example from Vitech's Well-Known Implementation of the Representation. Source: Vitech Corporation (n.d.).

e. N2 Chart

Literature varies on the origin of the N2 chart; however, the representation was first published by Robert Lano in an internal TRW report in 1977 (Kasser 2018). It is likely that Lano was “responsible for recognizing the true potential of the N2 chart” (Hitchins 1990, 9) rather than its sole creator. The N2 chart, seen in Figure 13, presents functions along the diagonal of a matrix with inputs/outputs inserted in all other matrix cell locations. Inputs flow into the top or bottom of a function box, while outputs flow from the left side or right side of the box. For illustrative purposes consider the origin point of the matrix with function 1 (F1) to be (0, 0). At point (3, 0) function 1 outputs an input to function 4. Similarly, at point (2, 3) function 4 outputs an input to function 3 (Federal Aviation Administration [FAA] 2006, 4.4-12).

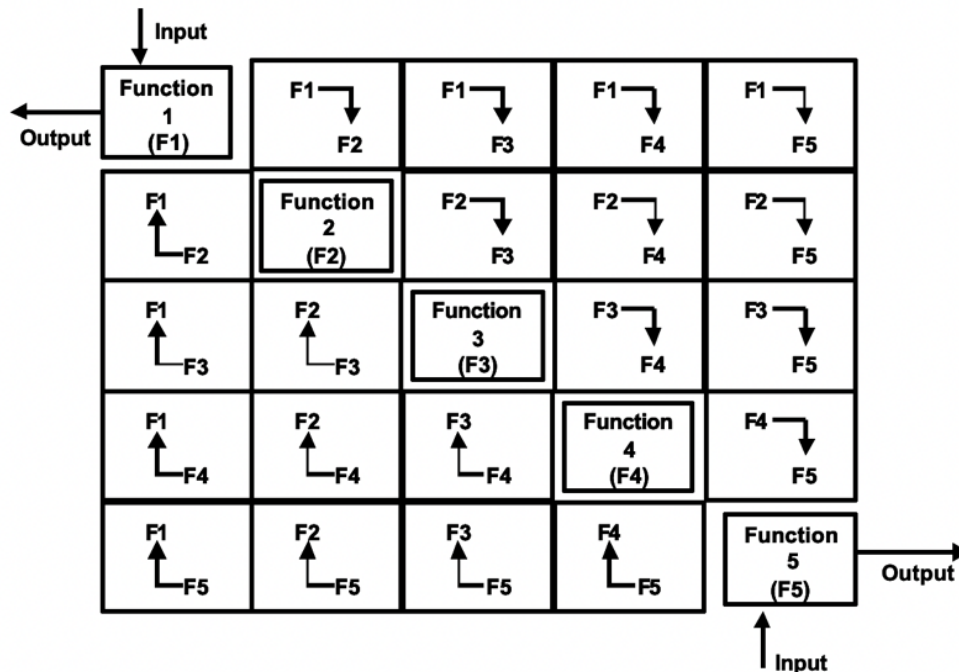


Figure 13. Example N2 Chart Demonstrating Item Flow between Functions.
Source: FAA (2006, 4.4-12).

The design structure matrix (DSM) can trace its origins to this N2 chart representation (de Weck 2012). The DSM allows further analysis of clusters of interfaces present within a system. This clustering of interfaces draws focus to critical areas of a system. The DSM presentation of interfaces within a grid is heavily inspired from the N2 chart, with the novelty of clustering representing DSM's novelty.

f. Structured Analysis and Design Technique (SADT)

SADT was designed to present complex functions of any system through a set of diagram representations of the system's activities. Softech published the graphical language in 1975 with a primary focus on data flow between activities. In its original publication, SADT was not provided with a computational toolset, and the diagrams were drawn by hand (Ludewig n.d., 183).

At its core, SADT boxes are specified in input, control, output, mechanism (ICOM) format (IBM Corp 2006). Data begins as an input represented by an arrow on the left of the box and is transformed into an output on the right side of the box. This data

transformation is governed by a control at the top of the box. The mechanism or asset performing the function is represented by an arrow at the bottom of the box. A box number is positioned in the bottom right corner inside each box, while a detailed page reference is included below the box in the right corner if decomposition is present. Parentheses around an arrowhead are used to represent omission of detail in a related parent or child diagram (U.S. Department of Commerce 1976, 5-20).

An example SADT is presented in Figure 14. “Policies/policy change recommendations” is a representative control, while “course material and descriptions” symbolizes an input. Outputs illustrated include “course units” and “unit description database.” Notably, “unit description database” is also an input to the “find course descriptions and units” function. Box numbers one through four include further detail and thus have page numbers 5-31 through 5-39 presented in the bottom right. The diagram contains no identified mechanisms (U.S. Department of Commerce 1976, 5-29).

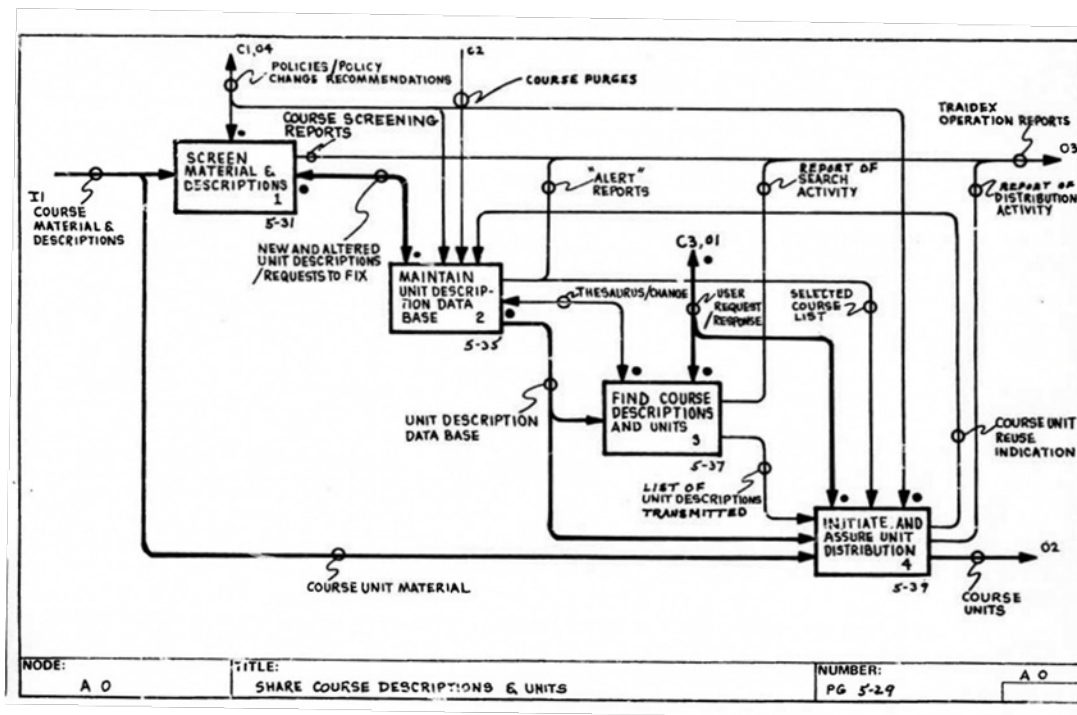


Figure 14. Example SADT Diagram Which Preceded the IDEF0. Source: U.S. Department of Commerce (1976, 5-29).

g. Systems Requirements Engineering Methodology (SREM)

SREM was designed to improve systems requirements capture for ballistic missile defense systems. Notably, SREM sought to be the first methodology to simulate the system design automatically, automate simulator construction, and provide modular building blocks for code development (TRW 1974, 142). A visual representation of requirements, called SREM R_Nets, is presented as a new way to capture functional behavior in requirements.

SREM R_Nets are graphically represented as triples, named Structured Data Trees. These trees are composed of functions with a set of inputs and outputs. Each leaf-level function is effectively a “black box” not expressing how inputs are transformed to outputs during execution. SREM R_Nets include the ability to decompose functions as a set of sub-directed graphs (Alford and Lawson 1979, 117). An example R_Net is depicted in Figure 15.

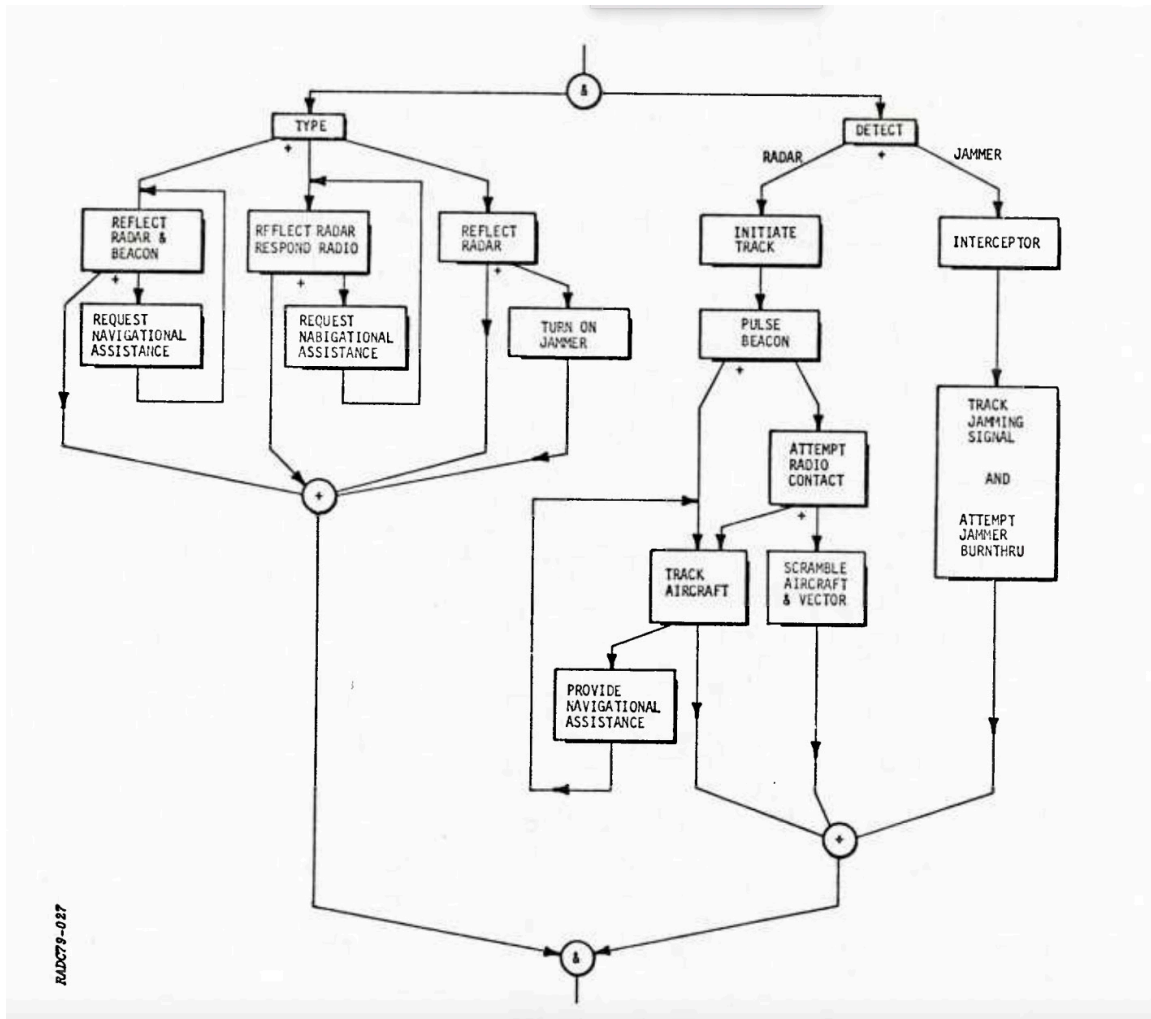


Figure 15. SREM R_Net Example Used in DOD's First MBSE Tool, DCDS.
Source: Alford and Lawson (1979).

SREM Structured Data Trees provide four logical operators: selection, parallel, replication, and left-to-right sequence. Selection (OR) is exclusive with only a single branch selected to execute. Parallel will execute all subsequent branches concurrently at the same time. Replication effectively copies a branch for a dynamically defined number of times. Execution occurs in parallel, but with replication the number of parallel branches is defined at execution. SREM heavily influenced future languages including the EFFBD representation (Alford and Lawson 1979, 116–117).

2. State Chart Graphical Notations

State charts provide representation of the interaction between a set of functions. Specifically, the input/output items transferred are emphasized within these representations. Decomposition of a function into sub-ordinate diagrams is typically supported by these notations to support complex system representation. This section presents both classical representations such as the IDEF0 in addition to actively developed representations such as the UML state machine diagram.

a. Integration Definition Function Modeling (IDEF0)

In 1993, National Institute of Standards and Technology (NIST) released Federal Information Processing Standards Publication (FIPS) 183 to describe the Integration Definition Function Modeling (IDEF0). FIPS 183 was derived from the Air Force Integrated Computer-Aided Manufacturing (ICAM) volumes from 1981. IDEF0 was designed to provide consistency to the modeling functions and their associated data elements. Similar to SADT, IDEF0 captures inputs, outputs, mechanisms, and controls for each function and their relationship to functions at the same level (Federal Information Processing Standards Publication 1993).

Of interest, IDEF0s include a tunneling feature, first introduced in SADT, whereas arrows which are not present in decomposed diagrams (or their parent diagram) are displayed within parentheses. This tunneling concept is present in Figure 16 with the designer and inspector mechanisms. Tunneling within IDEF0s is useful to portray incompleteness of a model and indicates the diagram's current maturity (Syque n.d.).

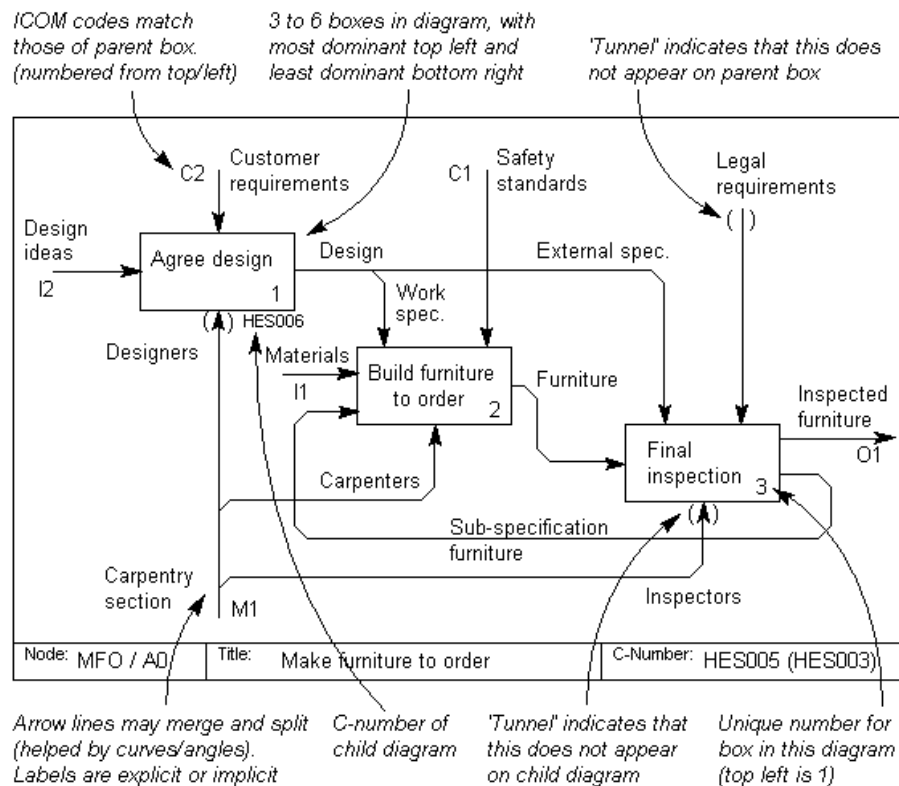


Figure 16. Example IDEF0 Formalized from SADT. Source: Syque (n.d.).

b. Harel Statecharts

In 1986, David Harel introduced a formal specification for state charts. Harel's motivation for the development of state charts is the significant prevalence of event-driven reactive systems in the real world, but that these event-driven semantics lacked a formal specification for computer simulation. At the time, simple finite state machines and state-transition diagrams, represented as direct graphs, could be utilized to represent a small set of state transitions; however, use for complex system representation was considered infeasible. Harel's first innovation was clustering of state levels in hierarchies and allowing the communication of events across state charts.

The finite state machine effectively executes in an exclusive OR fashion, whereas a single branch (event) is selected for execution from each state. Harel also introduced orthogonality, or parallel state machine diagrams, to support concurrency and parallel event paths. This parallel state is represented by a dashed line in Figure 17. Furthermore, display

and run represent decomposed states to manage complexity (Harel 1987, 233). The most popular object languages at the time, Object Modeling Technique (OMT), Booch, and Ivar Jacobson's objectory state diagrams all used Harel Statecharts as their state transition representation (Mattsson 1994, 6). Grady Booch (Booch Method) and James Rumbaugh (OMT) would later design UML with Ivar Jacobson (Booch, Rumbaugh, and Jacobson 2005).

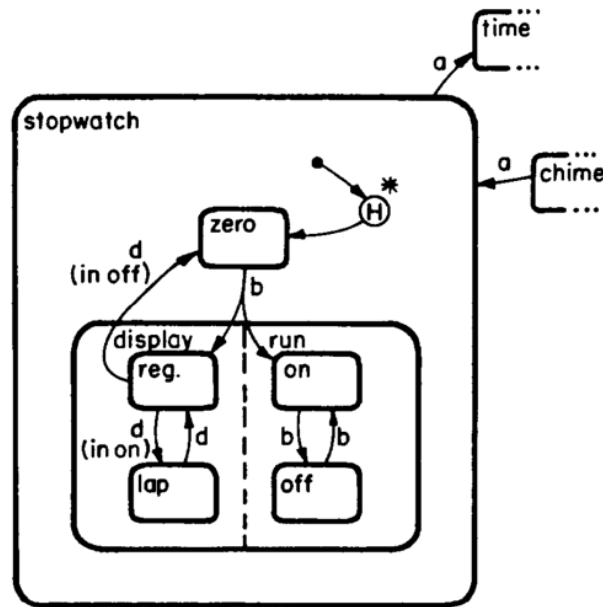


Figure 17. Example Harel State Chart Demonstrating State Decomposition.
Source: Harel (1987).

c. *Unified Modeling Language (UML) State Machine*

The UML standard presented state machines to model dynamic behavior of a system. UML represents states as rounded rectangles. The transitions between states are directed edges (lines) with names signifying the events on the diagram connecting states. This state machine technique was closely modeled from Harel's state charts "sustaining evolution [and] remained mostly stable, changing only to continue consistent with the rest of UML" (Chonoles 2018, 1). State charts, seen in Figure 18, added support for inheritance which allows for common functionality to be graphically shared among classes of states.

Hierarchical states from Harel are preserved in the form of sub-machines (Quantum Leaps 2015, 1).

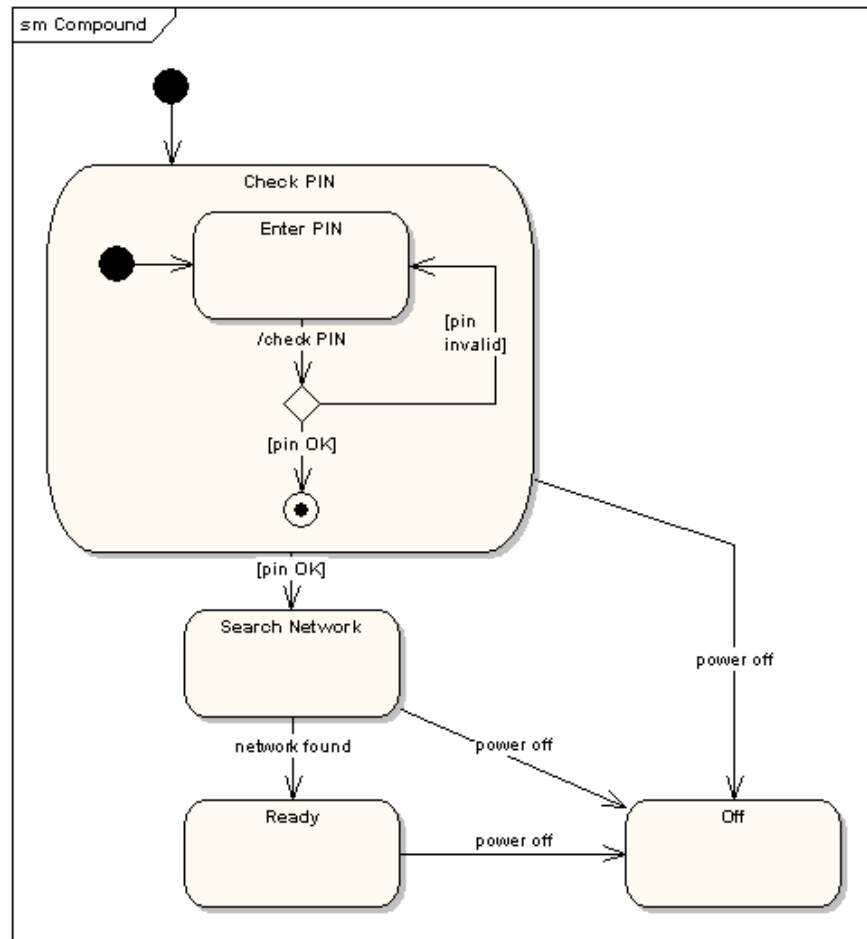


Figure 18. Example UML State Machine with Guarded ORs and Sub-Machines. Source: Sparx Systems (n.d.).

3. Flow-Based Graphical Notations

Functional behavior diagrams are each derived from the original flow chart and typically taught to every engineer early in their education. Each representation utilizes a box to represent an action and a line to represent flow between actions. This approach is typically used to specify and analyze the functional behavior of a system. Furthermore, execution of these behavioral representations differs greatly between ontological and UML foundation tools, with ontological tools supporting rich simulation and UML tools focused

more on step-through simulators. Capture of non-sequential flow and logical constructs differs significantly between representations and is discussed in this section.

a. Enhanced Functional Flow Block Diagrams (EFFBD)

Behavior diagrams commonly describe the process flow or steps to complete an activity. Generally, the input/outputs between functions are also represented in many classical and modern flow-derived languages. Notably, FFBDs do not have a supported facility to represent this data. In 1968, TRW Inc. developed the EFFBD notation internally (Long 2000). The EFFBD includes the display of an oval to represent input/output data connected between functions. These data outputs can trigger functions in the functional flow. Input triggers block execution of the receiving function until the dependent output function completes execution.

In Figure 19, a representative EFFBD is presented. “1 Serial Function” must wait to execute until external input is output from a prior functional flow. The “Multi-exit Function” outputs “Data 4,” which triggers “Output Function.” While it is still true that “Data” 4 must be received prior to output function’s execution, the placement of output function is after the full execution of all other functions in the diagram. Data one through three and five in gray represent non-triggering inputs. These inputs will not cause dependent functions to wait for execution to occur (Long 2000). Later, the EFFBD would influence development of other languages including BPMN, LML, SysML, and UML.

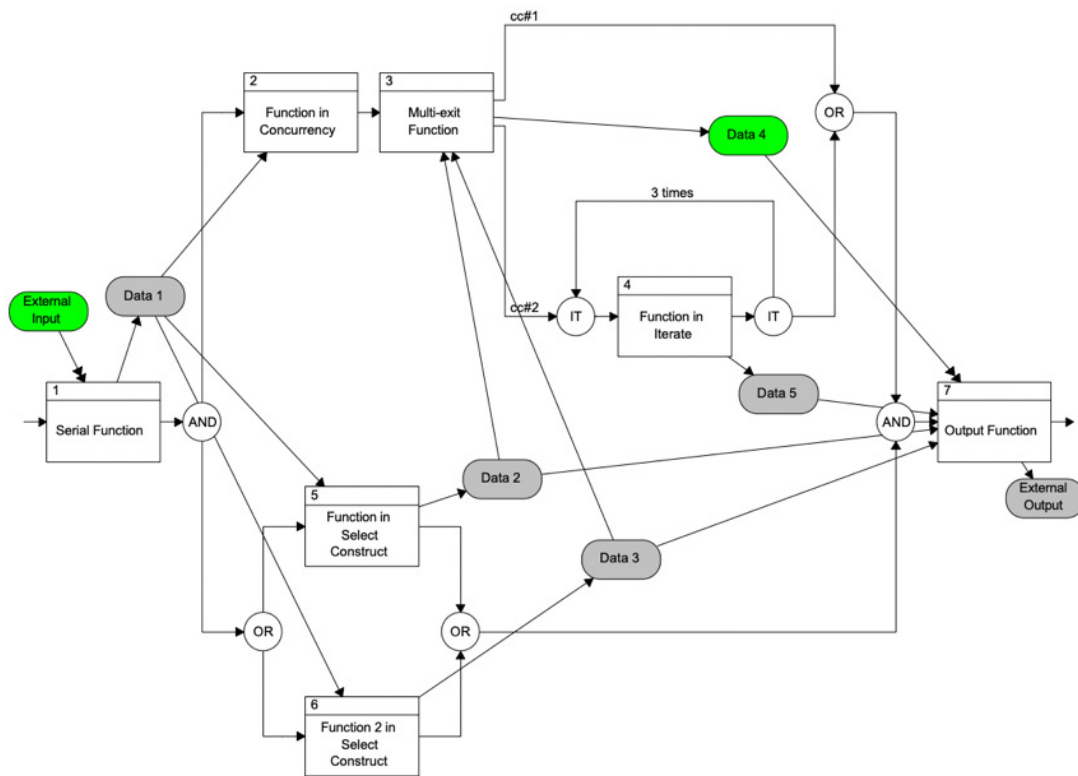


Figure 19. Example EFFBD Representation Adding Item Flow to FFBDs.
Source: Long (2000).

b. Business Process Modeling Notation (BPMN)

The Business Process Management Initiative (BPMI) was established in 2000 to standardize process modeling for business applications (Hanna 2021). BPMI developed two key technologies: Business Process Modeling Language (BPML) to describe process modeling in XML and Business Process Query Language (BPQL) for execution and deployment support. This standardization process led to the release of BPMN 1.0 in 2004 and an eventual merger into OMG in 2005.

BPMN includes several key elements including flows, event, activity, gateway, and pool. BPMN flows include sequential flow between activities, message flows between pools, and association flows between data and notes. Notably, in addition to standard start and end events, BPMN includes intermediate events between for incoming messages or time-based events (Lynch 2021).

BPMN includes a collaboration diagram, shown in Figure 20, to demonstrate interactions between processes. Additionally, BPMN 2.0 added choreography and conversation diagrams. Choreography diagrams only display processes which interact with each other, with a similar purpose to sequence diagrams. Visually, these diagrams use a bar at the top and bottom of an activity. The performer is placed at the bottom while the sourced interaction performer is placed at the top. A collaboration diagram is represented in Figure 21.

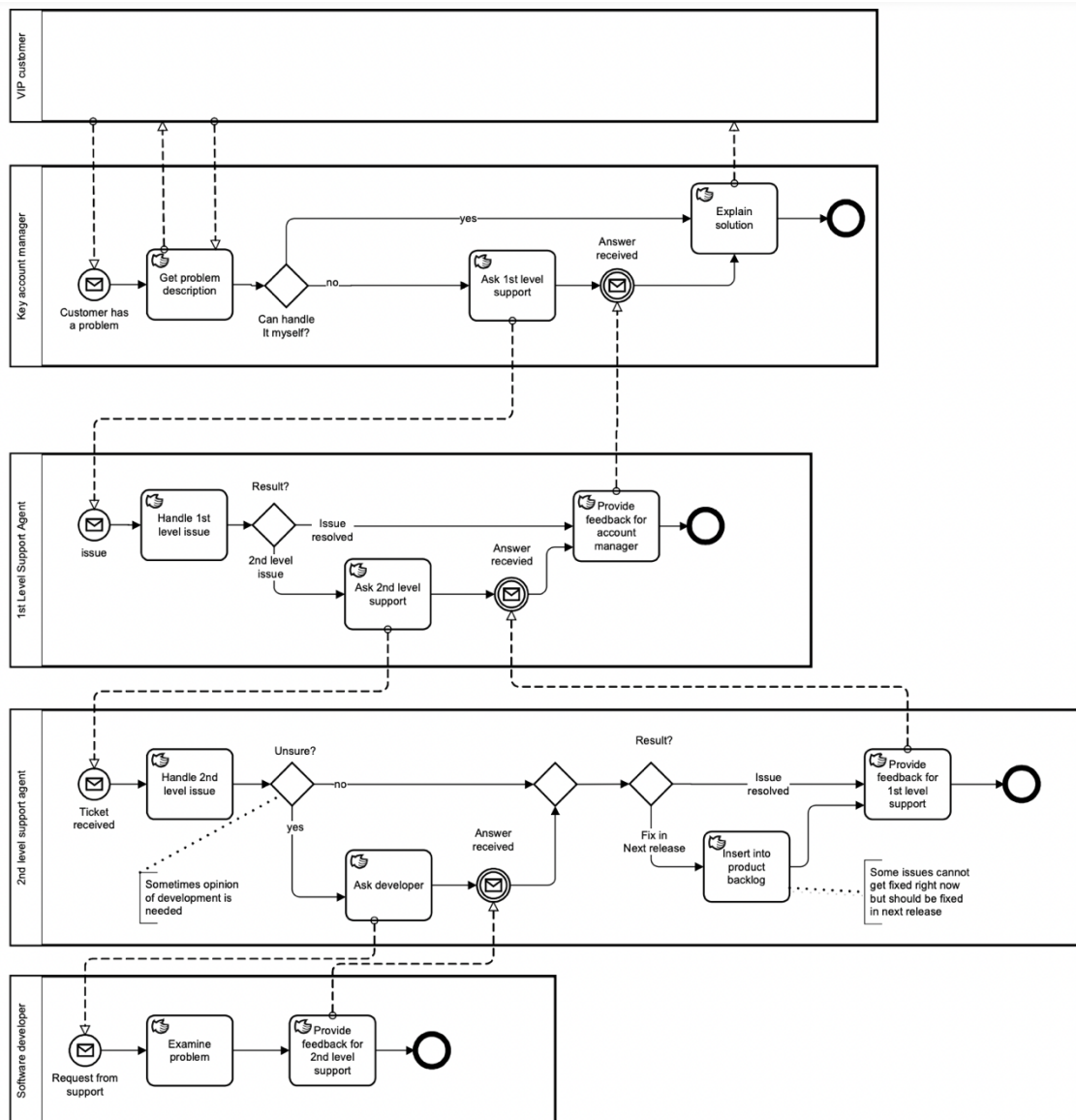


Figure 20. BPMN Collaboration Diagram. Source: OMG (2010).

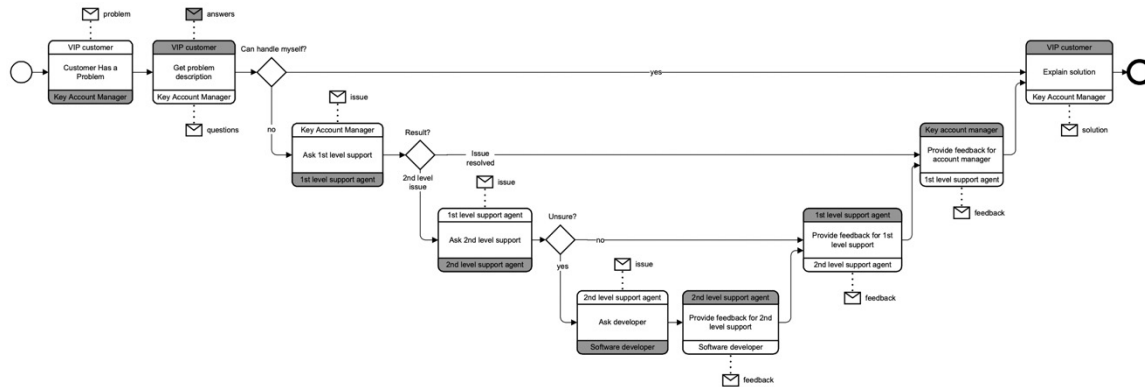


Figure 21. BPMN Choreography Diagram. Source: OMG (2010).

BPMN includes several extensions, including Decision Model and Notation (DMN) and Case Management Model and Notation (CMMN). DMN uses a Decision Requirements Diagram (DRD) to formally describe decisions, their input data, and applicable decision tables (OMG 2021s). CMMN supports case management and specific modeling notation to support exchange of case models. While BPMN is well suited to describe repeatable processes, it is not an optimal technique for ill-defined cases. CMMN allows for capture of ad hoc decisions which occur during an activity (OMG 2016). Usability studies, notably with only a limited pool of 21 developers, indicate that horizontal layouts are preferred and offer less eye strain. Importantly, users interact longer with diagrams that fit within a screen area and do not require scrolling (Lübke, Ahrens, and Schneider 2021, 127). A study on 30 business graduate students found BPMN to be no more usable than UML activity diagrams (Birkmeier, Klöckner, and Overhage 2010). A study demonstrating no usability improvement with BPMN is of interest as BPMN being superior to UML's usability is a common assumption among the business community.

c. *Lifecycle Modeling Language (LML)*

The LML action diagram, seen in Figure 22, derives from both EFFBD and SysML activity diagram representations. LML was developed by a group of practitioners led by the LML Steering Committee.

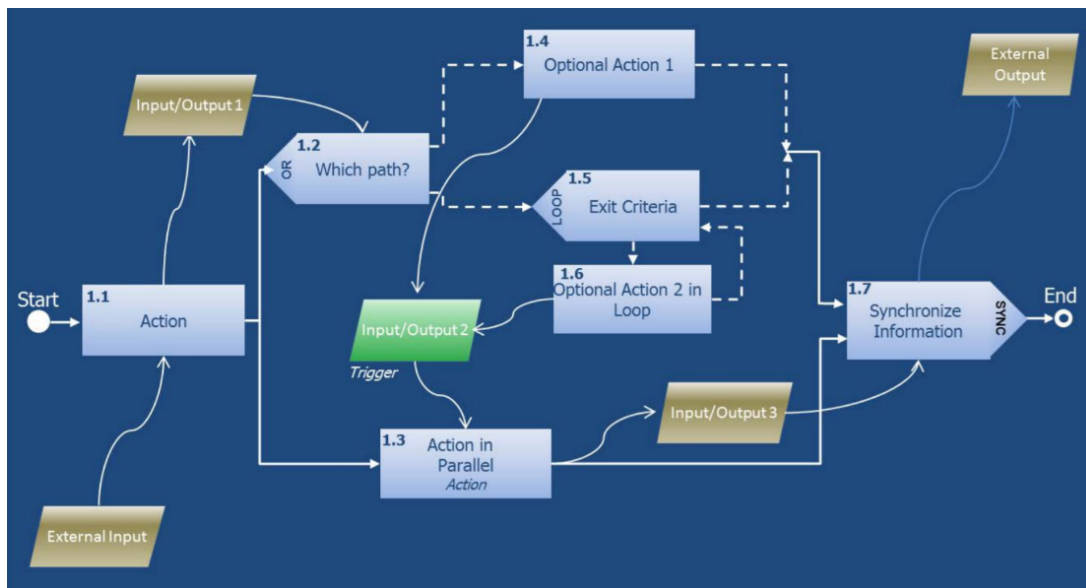


Figure 22. LML Action Diagram with All Constructs. Source: Lifecycle Modeling Language (2015).

LML represents all decisions as distinct entities with a name for each type of decision. LML provides the ability to trace these constructs to other modeling entities in an MBSE database; however, this also requires every construct to identify an action's name. LML separates OR, LOOP, and SYNC decision types through separate graphical labeling. Similarly, SysML2, currently in draft, will also use actions to represent control nodes. This evolution allows for fine-grained traceability to control gateways (Lifecycle Modeling Language 2015).

All ORs in LML are exclusive ORs with only a single branch executing following the OR gateway decision. In LML, a single branch must be selected at the OR gateway point at the completion of the ORs duration. If a branch is not selected, execution will fail with an error.

LOOPS within LML express both iteration (for...) and conditional continuation (while...) loop types. Each tool provider utilizes a proprietary programming language to express the type of loop logic. LML does not specify nor represent this logic graphically.

Notably, LML includes a unique control node, the SYNC point. A SYNC point action begins when any connected branch is activated. Once the SYNC point has finished

execution, it immediately kills all branches still running. The SYNC point allows for coordination of decisions on parallel paths between ORs. SYNC allows for graphical representation of parallel branch coordination.

The author of this report has participated in the original development of the action diagram.

d. UML Activity Diagram

UML introduced activity diagrams in UML 1.1 in 1997 (OMG 2017). UML activity diagrams describe an ordered sequential flow through activities. The use of solid lines to signify control flow has remained largely unchanged since the original 1.x UML specification. However, several additions have been included since its introduction, including a migration from a meta-model inherited from state machines to a new semantic meta-model specifically for activity modeling (Selic 2005, 9-10) based upon Petri Nets (Quatrani n.d., 31). UML 2.0 introduces data flow within control flow, attached data flow, and pin notations to signify input and output pins of activities. Of note, activity diagrams do not include explicit support for probabilistic nor duration modeling; however, probabilistic logic can be accomplished through guards on the control flow (Störrle 2005, 50).

Activity diagrams begin with initial nodes and include activity, action, object nodes (input/outputs), choice, and join nodes. Activity nodes represent a set of actions which perform tasks. These actions and activities are related to each other through control flow and represent the functional sequence of execution. Similarly, object nodes can be related via data flow and/or the control flow itself. Choice nodes represent decisions on the control flow and the selection of a branch to continue execution. The choice path is typically decided via guards, which block entry to a part of the control flow until they are evaluated as true. The end of a flow can be represented with a flow final node, which ends execution of that flow and resumes execution at a higher level of execution. A fork/join represents the start and end, respectively, of an explicit parallel execution path. An activity diagram ends with an activity final node (Quatrani n.d., 36).

A UML 2 activity diagram is represented in Figure 23. This diagram is directly comparable to an EFFBD diagram representation (Bock 2003, 169). Execution begins on the left with External Input and flows to the 2.1 Serial Function. This function outputs Item 1 and succeeds with a fork (parallel) node. That parallel node succeeds with 2.2 and 2.3, with 2.2 indicating a Multi-exit Function, symbolized with a parameter set in UML 2.0 semantics. A loop is represented with choice (decision) nodes and the iteration count controlled with a guard “before third time.” All object nodes, called Items in this example, serve as triggers to block execution of their input action until the output action has completed execution. Notably, this example includes streaming items, items which send a continuous stream of data, in Items 3 and 4.

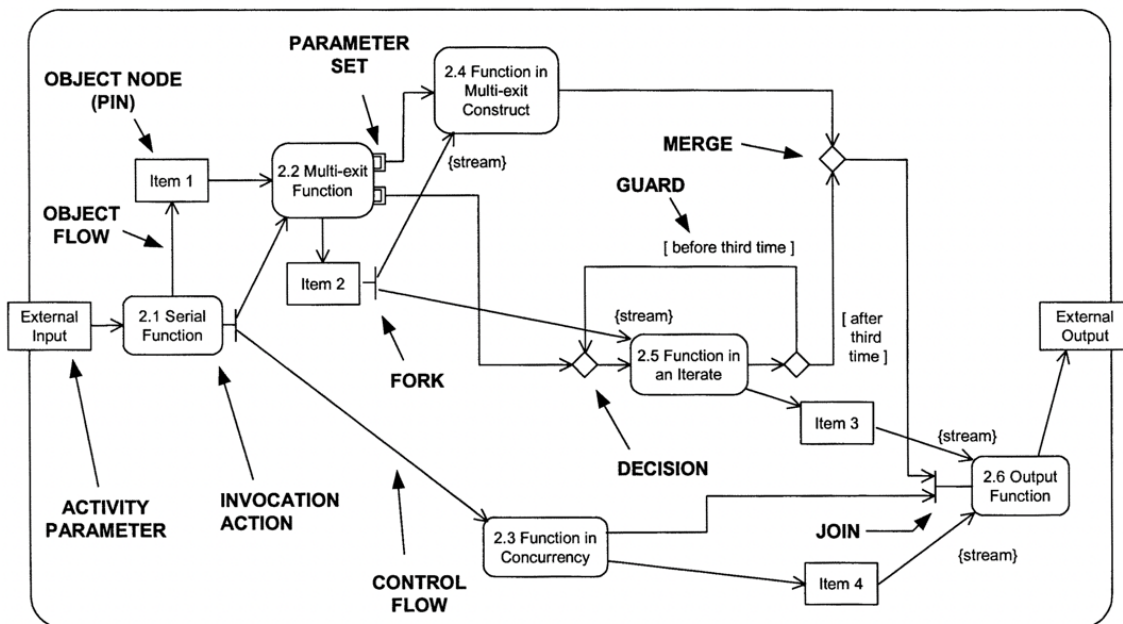


Figure 23. UML 2 Activity Diagram Closely Inspired from EFFBD Logic.
Source: Bock (2003, 169).

e. *Systems Modeling Language (SysML) Activity Diagram*

Systems Modeling Language (SysML) development was originally led by Cris Kobryn in collaboration with the International Council on Systems Engineering (INCOSE) and the OMG (SysML Forum n.d.). The activity diagram, derived from UML 2.0, can trace

its roots to the EFFBD. From a systems engineering perspective, EFFBDs were a familiar representation to capture behavior and proven to provide execution capabilities of defined logic (Bock 2003, 160). The origin of the UML 2.0 activity diagram is discussed in UML Activity Diagram.

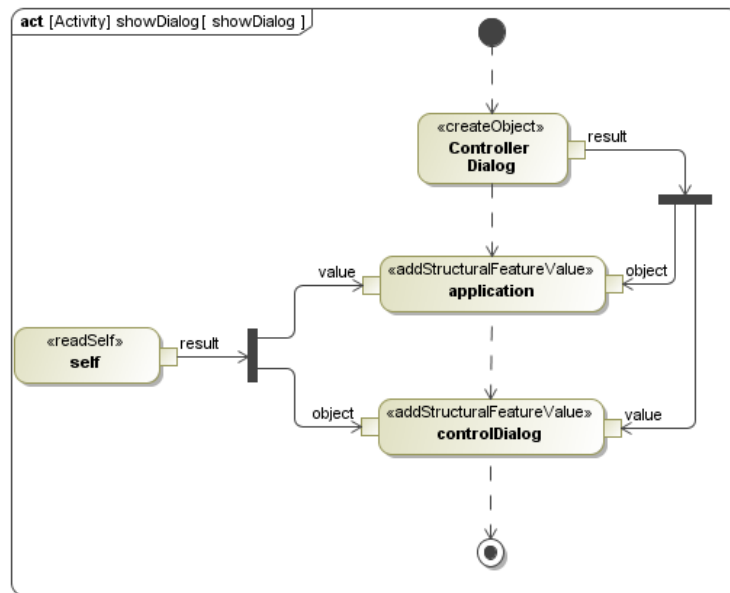


Figure 24. SysML 1.x Activity Diagram Demonstrating Commonality with UML 2's Representation. Source: No Magic (n.d.).

SysML included two notable additions: a parametric diagram to capture equations and a requirement diagram to specify requirement hierarchies and relationships. These two additional representations are out of scope for this behavior modeling analysis.

f. SysML2 Activity Diagram

The SysML2 activity diagram (under draft at the time of writing) includes several key changes, which differ significantly from its UML 2.x roots. All control nodes (fork, join, decision, etc.) are now named actions. This construct is like the LML Action Diagram approach to named control nodes with the addition of parallel control nodes. Currently, SysML2 does not support linkage of object nodes to control node actions, while would allow for the input of a decision to be readily apparent. Additionally, SysML2 defaults to

asynchronous input/output transfer in its graphical representation, as opposed to the traditional finish-to-start relationship first popularized in SREM R_Nets (Friedenthal 2021, 57). The author of this report has participated as a member of the SysML2 submission team.

4. Sequential Graphical Notations

Sequence diagrams describe interaction between a set of performers. These diagrams are commonly used to represent transitions between different modules or applications within software development. Additionally, sequence diagrams can be used to symbolize interaction between physical hardware systems. The well-known UML sequence diagram is presented in this section along with influencing representations.

a. Message Sequence Chart (MSC)

A MSC displays the communications between components of a system. The International Telecommunications Union (ITU) first standardized the notation in 1993 in the “Criteria for the use and applicability of formal description techniques: Message Sequence Chart” (International Telecommunication Union 1994, 1). Prior to this standardization of MSC, this technique was used extensively within the telecommunications community. Components communicate via messages with sending and receiving as distinct events. In an MSC, the components are represented by rectangles at the top of the diagram (ex. calling party). Messages are represented via arrowed lines between components (ex. off hook). Conditions (ex. idle) are represented by hexagrams, which display system states that block execution until their condition is true. Conditions can be global (shared across all components) or local to a specific component. Textual grammar to describe the in messages, out messages, conditions (shared all), and system components was also standardized through this process (International Telecommunication Union 1994, 22).

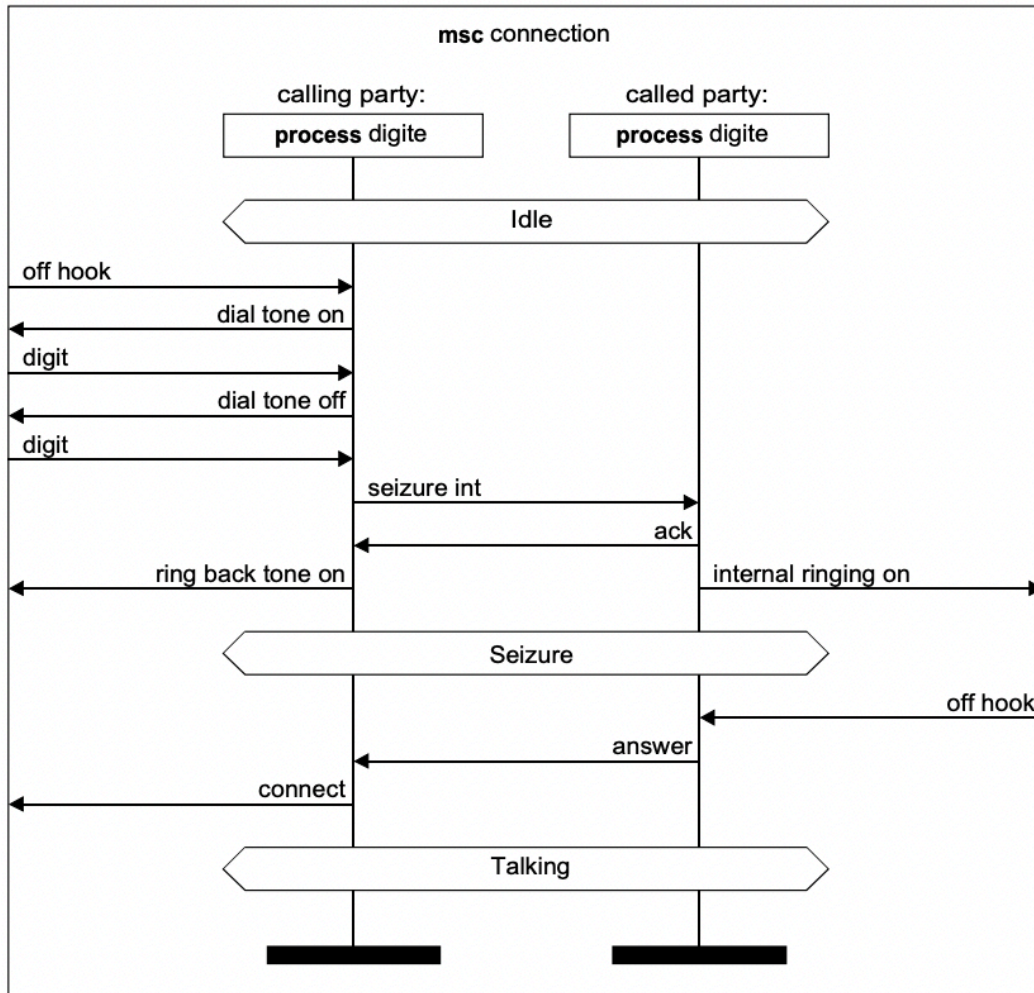


Figure 25. Message Sequence Chart Example Which Inspired the Modern Sequence Diagram. Source: International Telecommunication Union (1994, 22).

b. SysML and UML Sequence Diagram

The sequence diagram was present in the original UML 1.x specifications with a simple sequence diagram notation influenced by MSC. UML 2 included significant revisions versus the original specification. UML 2 added fragments to organize parts of a sequence diagram, decomposition of interactions, formal loop semantics, and object creation/destruction (IBM Corp 2006).

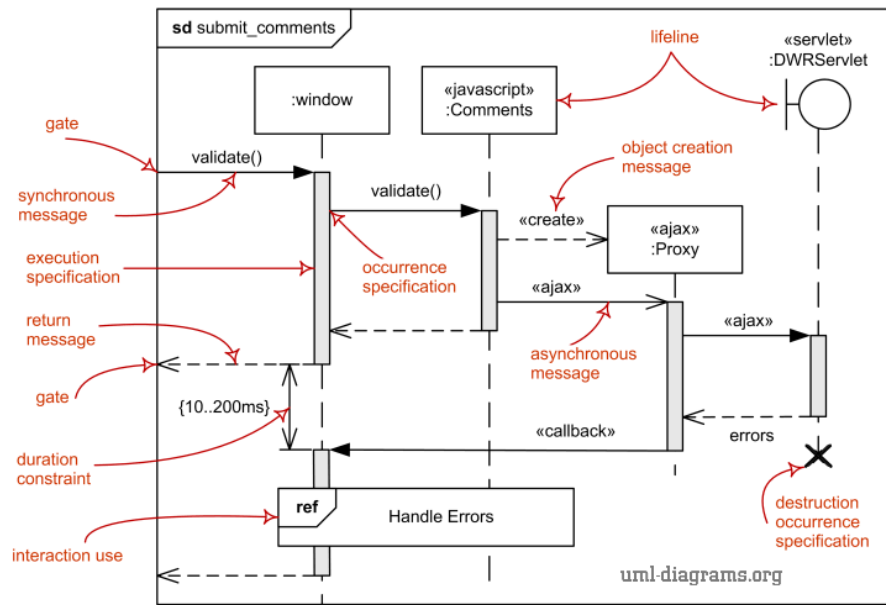


Figure 26. Example UML 2.x Sequence Diagram Demonstrating Interaction Flow Between Actors. Source: UML Diagrams (n.d.).

Given the historical influence from MSC, it is useful to compare with UML’s graphical representation. The two include minor naming differences: for example, action (MSC) versus ExecutionOccurrence (UML), method call (MSC) versus operation call (UML), etc. Both MSC-2000 and UML 2.x contain logical features to represent loops, opt (optional), par (parallel), and alt (alternative) paths. MSC includes timers within its instances; however, UML 2.x does not have an explicit way to model this behavior. UML 2.x requires separate lifelines to express this notation. Additionally, suspension areas—indication an event is postponed—have no UML 2 equivalent.

The sequence diagram remains unchanged from the original UML 2 specification in SysML 1.x. In applications of systems engineering, the sequence diagram represents the interaction between lifelines (actors) and can concisely represent these flows. Sequence diagrams can describe behavioral logic with parallel, loop, and selection notations layered over the interactions, but these notations add visual complexity to the diagram and can distract reviewers from the interactions represented.

UML 2.x adds a neg (negative) and asset for specification of absolutes in each scenario. These additions were inherited from Harel LSCs (Haugen 2004, 9).

c. *Harel Live Sequence Chart (LSC)*

A common criticism of MSCs is their lack of formal expressiveness. The original MSC standard and the UML 1.x derived sequence diagrams utilized comments to express functionality missing from the diagram's expressions. LSC is an extension of MSC, which includes live elements which must happen and forbidden elements which are not able to occur. Notably, LSCs introduce sub-charts, loops/iteration, and alternative flows to sequence chart notations.

LSC supports both mandatory (universal) and provisional (extensional) representations. Universal LSCs must be satisfied in every system run while existential LSCs must only be satisfied once. LSCs include hot semantics whereas the instance must continue to the next location, all messages sent must be received, and all conditions must be met or the entire LSC is aborted. Conversely, LSCs also include cold semantics where execution can end at a location, message receipt may not occur, and unmet conditions only require sub-chart exit. LSC sub-charts can utilize conditions to control branching (OR logic) and sub-chart multiplicity to specify iterations. An example LSC specifying both sub-charts and optional (OR charts) is in Figure 27. LSCs heavily influenced UML 2 semantics and future revisions of the MSC standard (Damm and Harel 2001).

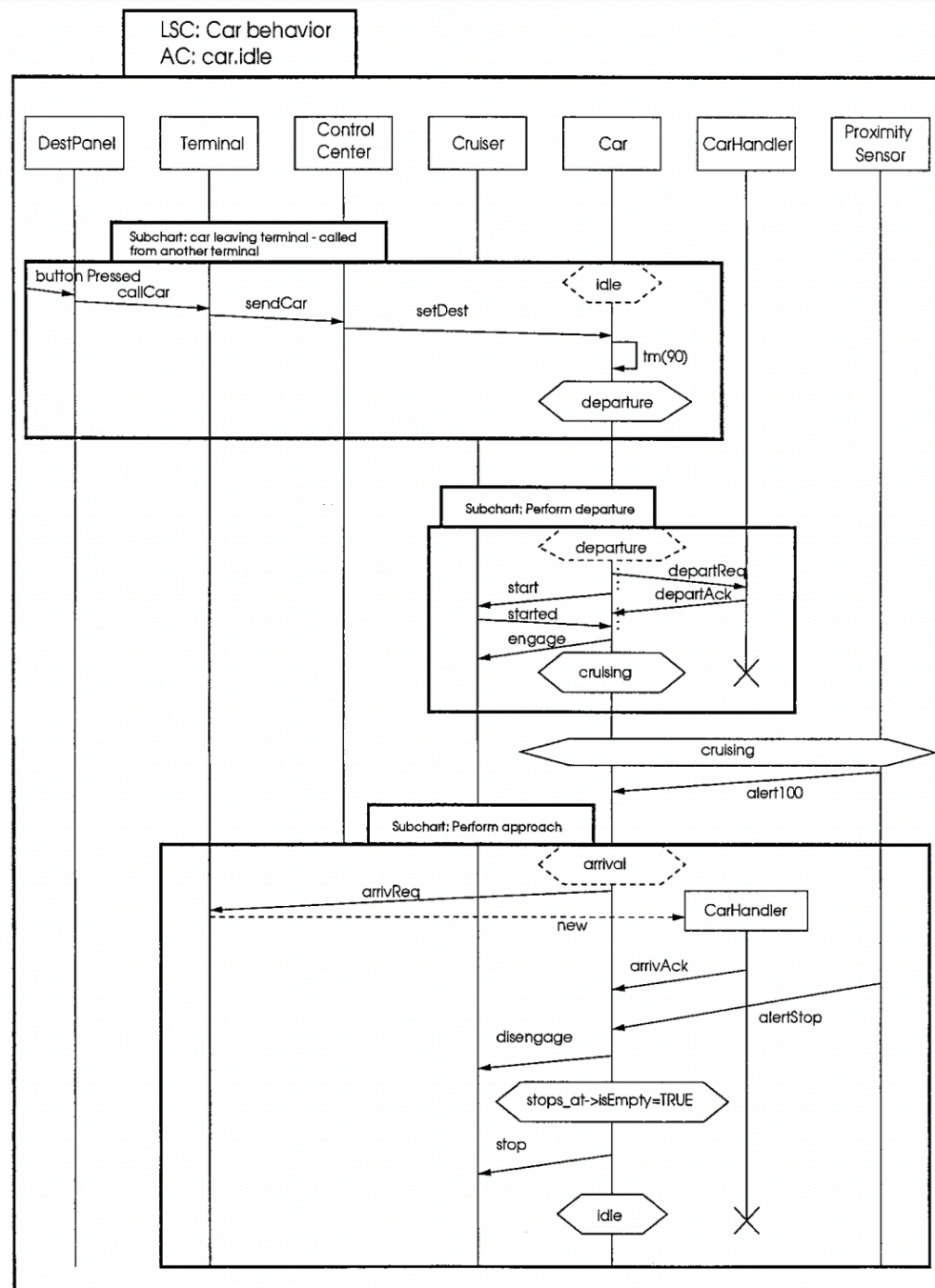


Figure 27. Example LSC. Source: Damm and Harel (2001).

C. CHAPTER SUMMARY

MP events support several logical operators including selection, iteration/looping, parallelism, and coordination. Similarly, functional flow graphical languages, originally

inherited from process charts, support selection, looping, parallelism, and input/output transfer are supported in MP user-defined relations. The graphical language community first formalized these languages with Petri Nets and SREM R_Nets for execution. All functional flow languages with execution semantics inherit concepts from these early representations. Conversely, it was found that data flow (ex. SADT, IDEF0) and interaction diagrams (MSCs, sequence) do not support logical operations or only support logic through coordination activities. A history of all applicable graphic languages is presented in an original graphic, Figure 28, overviewing chronologically the dates and flow of language development. Note this diagram presents first public use of a graphical language, and as such, the EFFBD is within the 1990s row when popularly used, and not present in 1960s when only used at TRW.

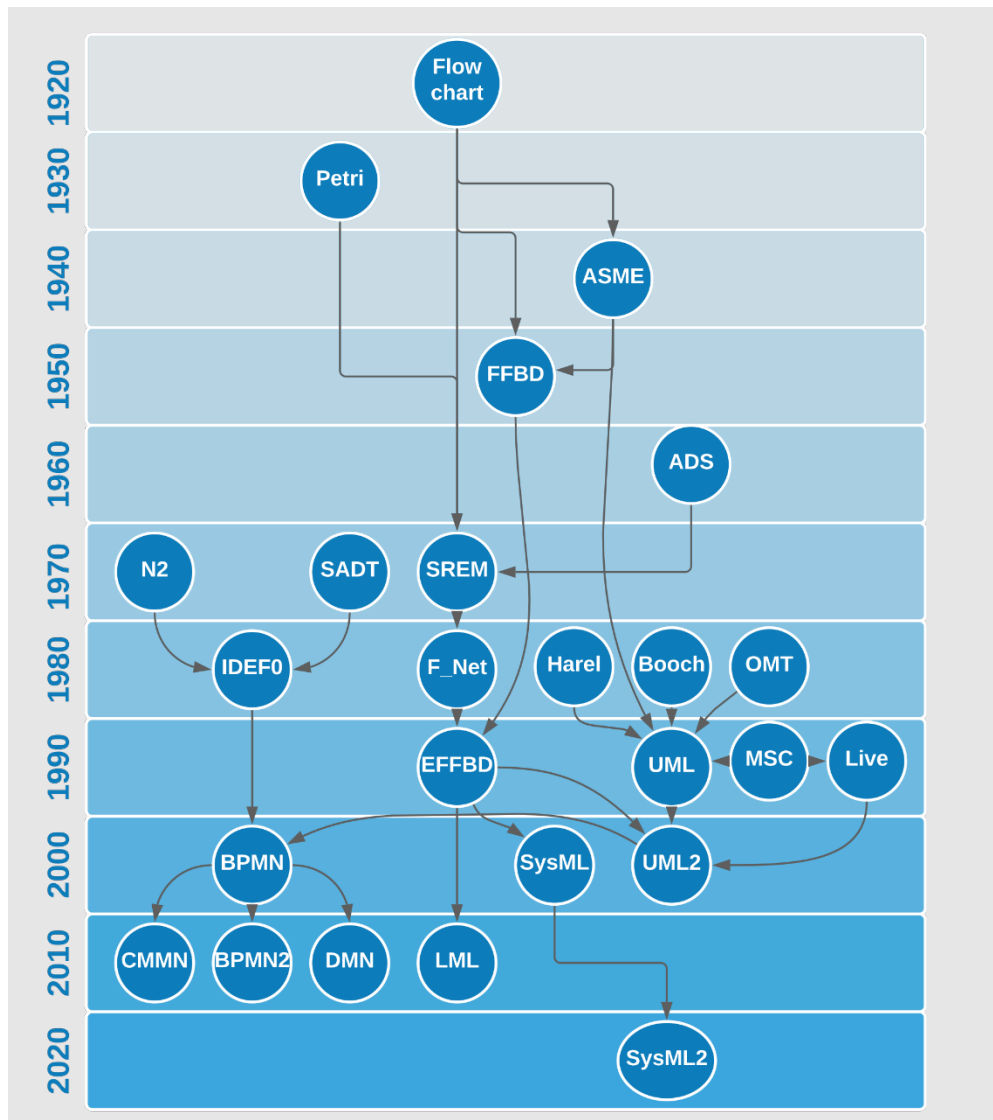


Figure 28. Functional Graphic Language History

III. OPEN-SOURCE DATA ANALYSIS

This open-source data analysis chapter informs selection of a graphical representation for the basic MP language elements through analysis of BPMN, EFFBD, flowchart, IDEF, LML, SysML, and UML regarding reach, popularity, and relative sentiment. These languages were chosen due to their current relevance historically and their ability to represent functional flow. As discussed in III.A Google Trends Analysis, only BPMN, flowchart, SysML, and UML had enough data to analyze in this chapter.

A. GOOGLE TRENDS ANALYSIS

Google Trends is an analysis website which tracks search traffic popularity overtime. Google has a database of search volumes from 2004 through the present. This database can be utilized to discover interest in relevant search terms. Utilizing the literature review as a guide, the search terms “bpmn,” “EFFBD,” “flowchart,” “IDEF0,” “lml diagrams,” “sysml,” and “uml diagrams” were analyzed for search traffic patterns in 2021. Only “bpmn,” “flowchart,” “sysml,” and “uml” generated non-zero average percent traffic when compared to the top search term “flowchart.” These four diagram types were analyzed further using Google Trends Topic Search. This feature bundles multiple keyword patterns to form topic categories of terms (Google Trends n.d.).

The topic analysis found the topic “flowchart” as the largest generation of search traffic by far, accounting for an average of 70–80% of all search volume compared, as seen in Figure 29. UML shows a downward trend with a 25–30% traffic share in 2011 falling to ~15% in 2020. BPMN and SysML demonstrated similar patterns as each other accounting for small single digit percentages consistently overtime, resulting neither in significant traffic drops nor increases since 2011.

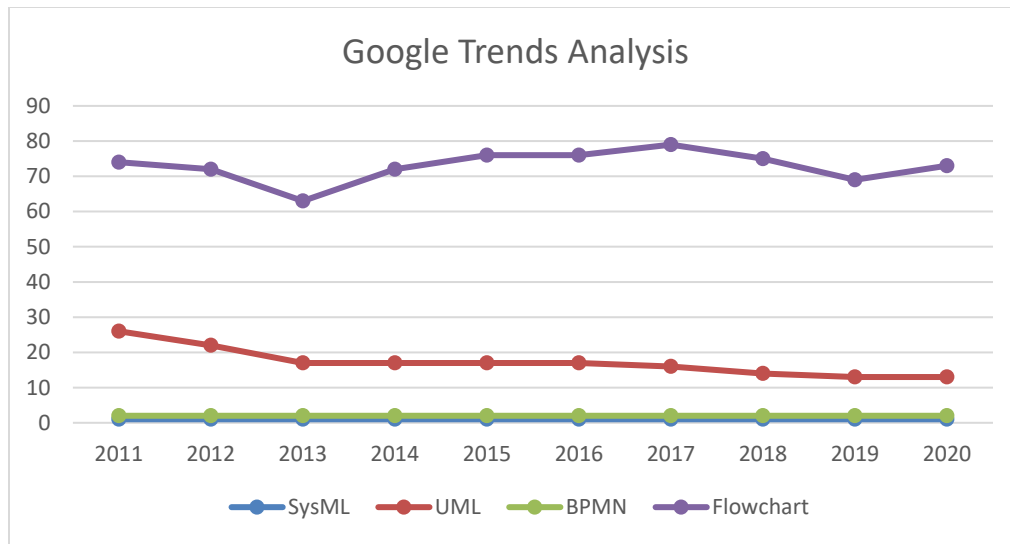


Figure 29. Google Trends Analysis Chart Demonstrating Flowchart’s Popularity Dominance

B. GOOGLE SCHOLAR ANALYSIS

Google Scholar is a web search engine which focuses on academic articles and information. Google Scholar includes “articles, theses, books, abstracts and court opinions, from academic publishers, professional societies, online repositories, universities” (Google Scholar. n.d.). A search was conducted on November 10, 2021, for each selected search term: SysML, UML, BPMN, and Flowchart. Table 1 captures Google Scholar search volume by year.

Table 1. Google Scholar Popularity Data (in hundreds)¹

	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
SysML	12.8	14.2	14.7	18.8	16.6	19	19	22.2	23	22.8
UML	266	246	274	258	259	250	252	248	221	227
BPMN	31.6	34.2	38.1	39.7	43.4	44.2	48.8	53	55	49.9
Flowchart	461	566	701	814	935	956	905	919	803	595

¹ Data in Table 1 was collected on November 10, 2021, through Google Scholar, searching for the terms: SysML, UNL, BPMN, and Flowchart. URL: <https://scholar.google.com>.

These results were graphed in a line graph by year to identify trends in publications for each diagram representation in Figure 30.

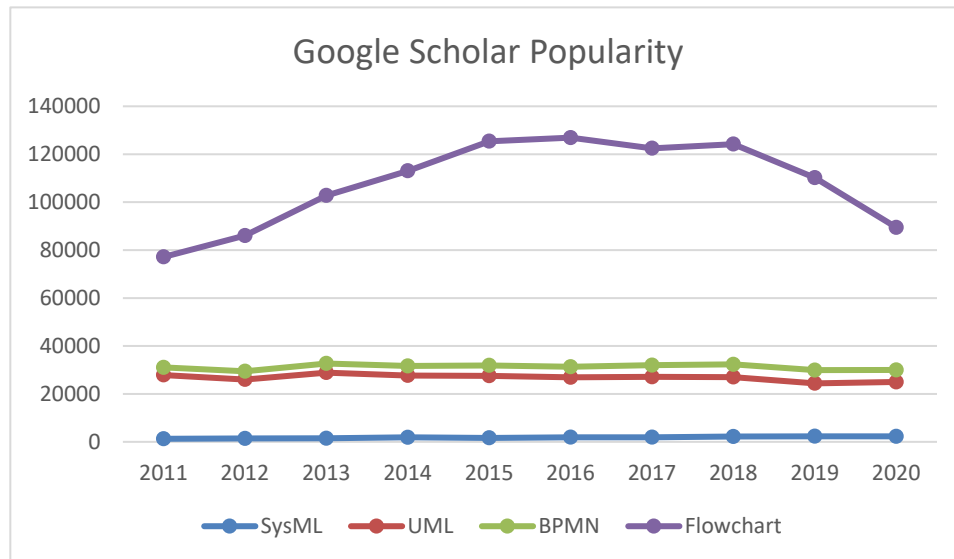


Figure 30. Google Scholar Popularity Chart Again Demonstrating Flowchart’s Overwhelming General Popularity

This analysis of academic publications found flowcharts to have the largest number of publications by a factor of three to five. However, flowchart demonstrated a decline in publications in 2019 and 2020, while all other representations were relatively flat over that period. It is unknown if the COVID-19 pandemic partially effected the results for flowchart in this area with the reduction of some conferences and journal publications, however this pattern was not observed for other languages, and is trending downward again in 2021. This information was discovered during a Google Scholar search for the term “flowchart” in years 2019, 2020, and 2021.

C. SOCIAL MEDIA POPULARITY ANALYSIS

The social media analysis tool, Brand24, was utilized to capture social media volume from October 15 to December 15, 2021 (Brand24 n.d.). Brand24 captures data from Twitter, news, Instagram, podcasts, blogs, news, videos, and web sources to track mentions of keywords in social media data. For UML the keywords “unified modeling

language,” “uml modeling,” and “uml diagram” were used for social data fetching and not “uml” itself. Most “uml” traffic was not related to the graphical programming language; for example, the University of Massachusetts at Lowell was a significant source of “uml” traffic. The keywords “bpmn,” “sysml,” and “flowchart” were each used to represent their languages. This analysis for 1 month found that flowcharts were four times more popular than the second most popular language BPMN. Notably the share of all other languages combined represents approximately half of flowcharts total social media mention share as seen in Figure 31. The inability to use simply “uml” itself could have partially effected UML’s represented market share; however, the Google Trends analysis demonstrated similar patterns to this social media capture.

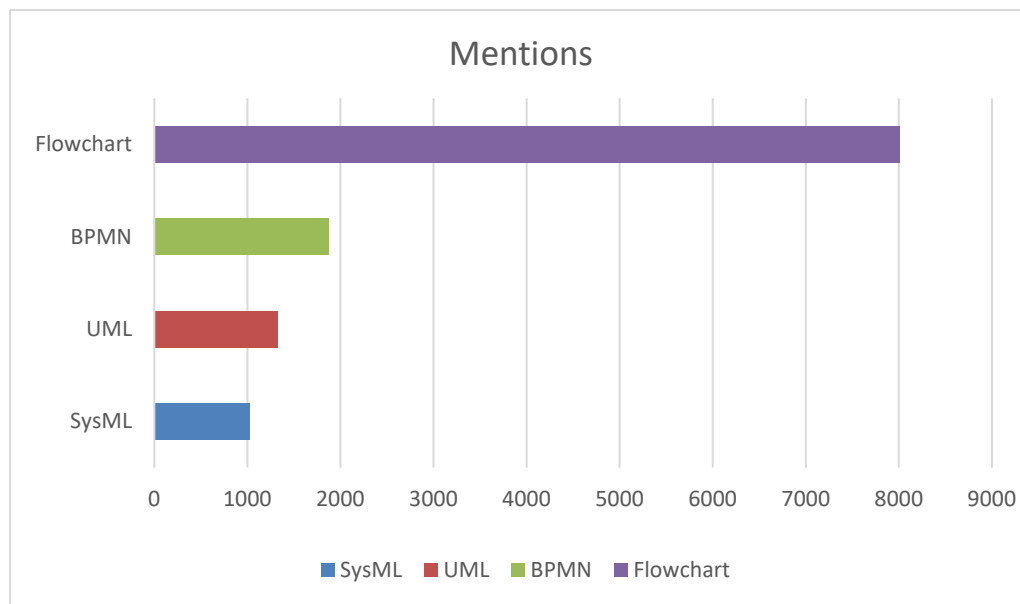


Figure 31. Social Media Mention Chart Demonstrating Flowchart’s General Popularity

In addition to tracking the number of mentions, Brand24 has a sentiment analysis capability, which uses artificial intelligence to find positive, negative, and neutral social mentions. This tool was used to categorize all mentions on each diagram type over this period as seen in Figure 32. Social media found SysML to be the most positive in sentiment (83%) versus flowchart at only 58.8%. Of note, this only analyzes user feeling on a post,

not necessarily feeling toward a specific representation itself. Spot human analysis of flowchart sentiment signifies the negative sentiment is not related to flowcharts as a diagram representation. Notably, traffic for SysML also grew over this period, perhaps demonstrating renewed interest in SysML with a new release. However, the flowchart negativity is likely due to its use as a general representation and not specifically towards rejection of flowcharts to visualize processes.

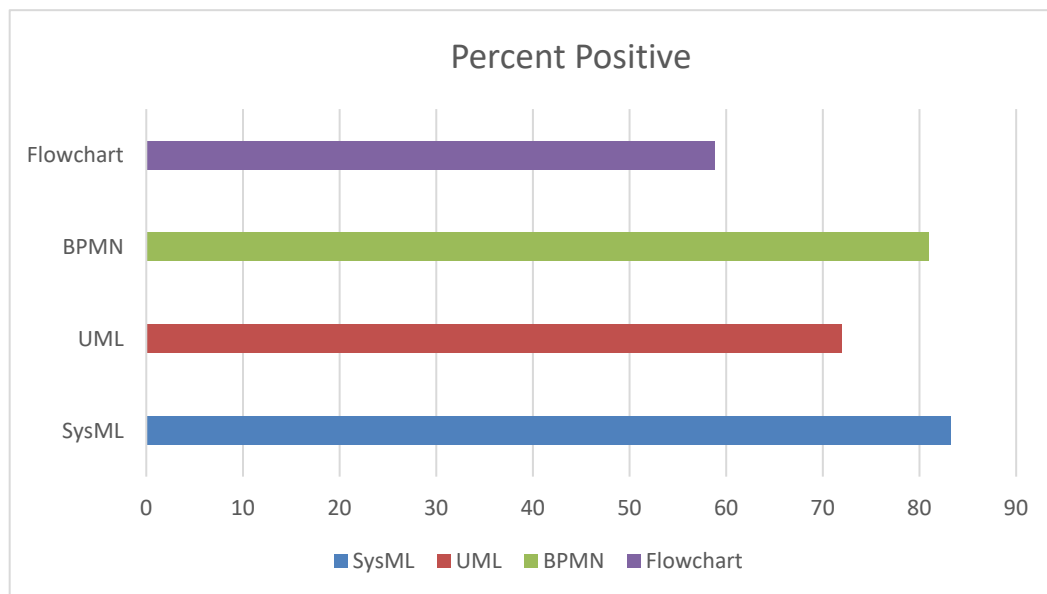


Figure 32. Social Media Sentiment Analysis Showing Positive Sentiment for All Representatives Except Flowcharts

D. GITHUB REPOSITORY ANALYSIS

GitHub is a Microsoft, Inc. cloud service, which hosts source code and issues tracking for over 50 million developers worldwide (Johnson 2020) community. GitHub is the largest online repository of source code and is continually growing by 10 million new developers annually. GitHub stores repositories of tools, language specifications, and utilities to interact with many popular graphical modeling languages. In November 2021, the count of repositories, lines of source code, and number of commits were captured via GitHub's search tools as seen in Table 2.

Table 2. GitHub Graphical Language Data²

	Repositories	Code	Commits
SysML	248	71000	5000
UML	13000	10000000	21000000
BPMN	2000	998000	62000
Flowchart	3000	1000000	173000

The number of repositories is heavily skewed towards UML in the software community. This is also true for the number of lines of related source code and commits. Notably, of the results captured from GitHub for UML only 1,000 results were last updated prior to 2015, meaning that the vast majority of results are from the past seven years. This demonstrates that there is still significant use of UML within the software community at large. BPMN and flowchart were statistically similar with SysML obtaining a very small market share of only 248 related repositories. An analysis of these trends is visualized in Figure 33. The outer ring compares commits, middle ring compares lines of code, and the inner ring compares the number of repositories between languages.

² Data from Table 2 and Figure 32 are from information collected on November 2021 during a GitHub search. URL: <https://github.com>.

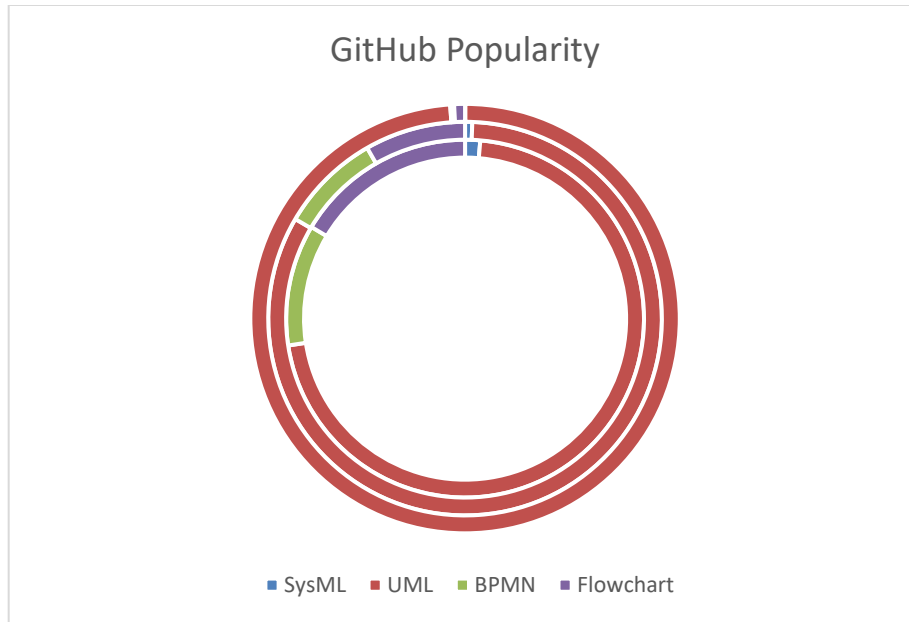


Figure 33. GitHub Graphical Language Chart Demonstrated Dominance by UML for Software Graphical Representation

E. YCOMBINATOR ANALYSIS

The analysis of YCombinator data employs unique software developed for this thesis and utilizes custom logic to perform both a natural language processing (NLP) analysis and a natural language understanding (NLU) analysis with sentiment scores. A methodology of operation is presented to fully capture custom logic utilized in this paper.

1. Methodology

a. Narrative

The YCombinator Data Analysis is performed through a series of primary phases: data wrangling, NLP analysis, and NLU analysis. In the data wrangling phase, the top 10 stories of all time from YCombinator Hacker News are fetched. Most comments on a keyword were found in the most popular stories with an almost exponential relationship among stories and few comments available in less popular stories to analyze. All comments for these top 10 stories are then parsed into a set of JSON files. In the NLP analysis, phase words are stemmed (reduced to their Latin-like root) and then ranked by top keyword. Additionally, the top adjectives are also organized and ranked accordingly. In the NLU

analysis phase, comment text is sent to IBM Watson® for sentiment analysis. This sentiment analysis is then scored on a scale of -1.0 to 1.0. The higher the score the higher the general sentiment of the analyzed text.

b. MP Validation

To validate the process and methodology for YCombinator Data Analysis, an MP schema was developed.

(1) Root Events

The MP textual schema is composed of three root events: an Analysis Script, YCombinator Cloud, and IBM Watson®. These three root events represent the components or actors of this approach. The analysis script is a custom Node.JS application to process and analyze YCombinator data. YCombinator, through Algolia, provides a RESTful JSON Application Programming Interface (API) to request YCombinator data. IBM Watson® performs natural language understanding on textual data provided to Watson.

```
SCHEMA YCombinator_Analysis
ROOT Analysis_Script: Request_First_Page_Of_Stories
( ( Receive_Stories Request_Comments_From_Top_Ten_Stories
    Analyze_Dataset_With_NLP Analyze_Dataset_With_NLU ) |
    Throw_Story_Error | Story_Fetch_Timeout );
ROOT YCombinator_Cloud: Attempt_Send_Stories
Attempt_Send_Comments_For_Stories;
ROOT IBM_Watson: (* Analyze_Text_With_NLU *);
```

(2) Compositive Events

A series of composite events were identified as decomposition of scenario root events. These composite events allowed for capture of failure states (ex. `Analyze_Text_With_NLU`) and iteration of datasets (ex. `Request_Comments_From_Top_Ten_Stories`). The capture of this logic allows for an explanation of the data science methods used to perform this social data analysis.

```

Analyze_Text_With_NLU:
    ( Provide_Text_Sentiment | Send_Text_Sentiment_Error );

Attempt_Send_Stories:
    ( Provide_First_Page_Of_Stories |
      Send_Story_Fetch_Error | Fail_To_Send_Stories );

Attempt_Send_Comments_For_Stories:
    ( Provide_Comments | Send_Comment_Fetch_Error |
      Fail_To_Send_Comments );

Request_Comments_From_Top_Ten_Stories:
    ( * ( + Request_Story_Comments ( Collect_Comments |
      Throw_Comment_Error | Comment_Fetch_Timeout ) + ) * );

Analyze_Dataset_With_NLP:
    ( * Extract_Keywords Remove_Digits Stem_Keywords * )
    Reverse_Sort_Top_Keywords ( * Extract_Adjectives * )
    Reverse_Sort_Top_Adjectives Visualize_NLP_Results;

Analyze_Dataset_With_NLU:
    ( * Request_NLU_Analysis (
      Collect_Text_Sentiment | Log_Error ) * );

```

(3) Coordination Logic

Coordinate logic allows for root event collaboration across swimlanes. Many events must wait for other swimlane events to complete before proceeding. For example, a request must be issued from the analysis script before IBM Watson® can analyze data received from that request. This coordination ensures events occur in the proper sequence and wait for necessary input data.

```

COORDINATE $x: Request_NLU_Analysis FROM Analysis_Script,
           $y: Analyze_Text_With_NLU FROM IBM_Watson

```

```

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Send_Text_Sentiment_Error FROM IBM_Watson,
           $y: Log_Error FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Provide_Text_Sentiment FROM IBM_Watson,
           $y: Collect_Text_Sentiment FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Request_Story_Comments FROM Analysis_Script,
           $y: Attempt_Send_Comments_For_Stories FROM
YCombinator_Cloud

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Provide_Comments FROM YCombinator_Cloud,
           $y: Collect_Comments FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Send_Comment_Fetch_Error FROM YCombinator_Cloud,
           $y: Throw_Comment_Error FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Fail_To_Send_Comments FROM YCombinator_Cloud,
           $y: Comment_Fetch_Timeout FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Request_First_Page_Of_Stories FROM Analysis_Script,
           $y: Attempt_Send_Stories FROM YCombinator_Cloud

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Provide_First_Page_Of_Stories FROM YCombinator_Cloud,
           $y: Receive_Stories FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Send_Story_Fetch_Error FROM YCombinator_Cloud,
           $y: Throw_Story_Error FROM Analysis_Script

```

```
DO ADD $x PRECEDES $y; OD;

COORDINATE $x: Fail_To_Send_Stories FROM YCombinator_Cloud,
           $y: Story_Fetch_Timeout FROM Analysis_Script

DO ADD $x PRECEDES $y; OD;

GLOBAL SHOW ACTIVITY DIAGRAM Analysis_Script,
           YCombinator_Cloud, IBM_Watson;
```

c. Activity Diagram and Sequence Diagrams

MP provides a SysML-like activity diagram representation to visualize MP input grammar. This facility allows for a traditional flowchart-like graphic to capture event logic before iterating over a set of scope-complete sequence diagrams. This activity diagram, seen in Figure 34, demonstrates the successful analysis script path as compared to two primary failure states with story timeouts and story fetch errors. This does not visualize later issues with IBM Watson® nor comment fetching in the process. These events are available in their output sequence diagram representations.

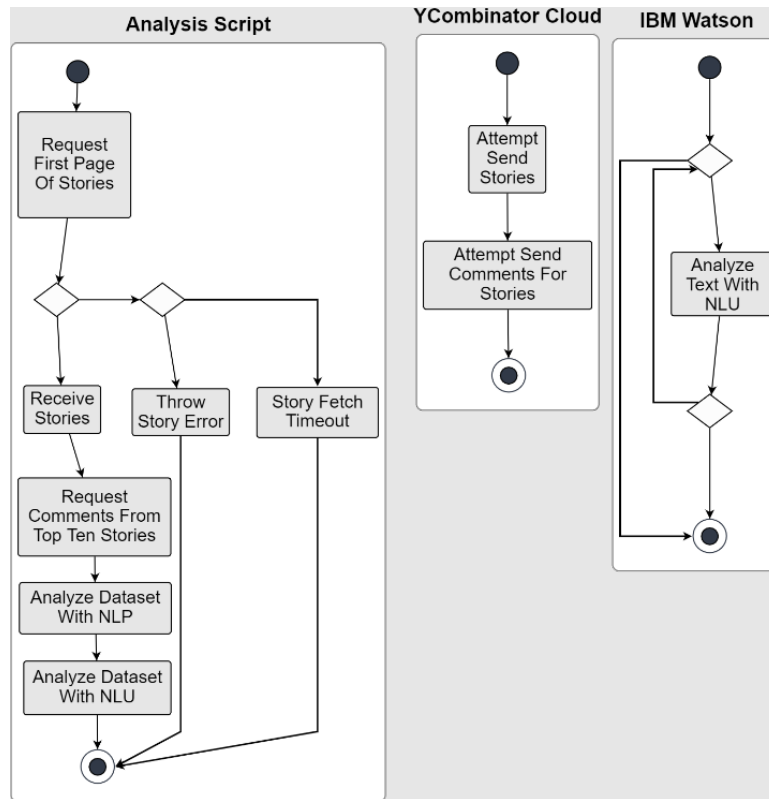


Figure 34. MP Activity Diagram of NLP and NLU Analysis Approach

2. Results

a. *Sentiment Analysis*

A total of 2,011 comments were captured from YCombinator for further analysis—1,388 comments were captured on UML, 611 comments on Flowchart, and only 17 comments were found on BPMN. These results, in the software community, are like those captured via GitHub Repository analysis. The low comment counts on BPMN were surprising to the author; however, these can be explained by the community’s software focus. The total number of comments by share is exhibited in Figure 35.

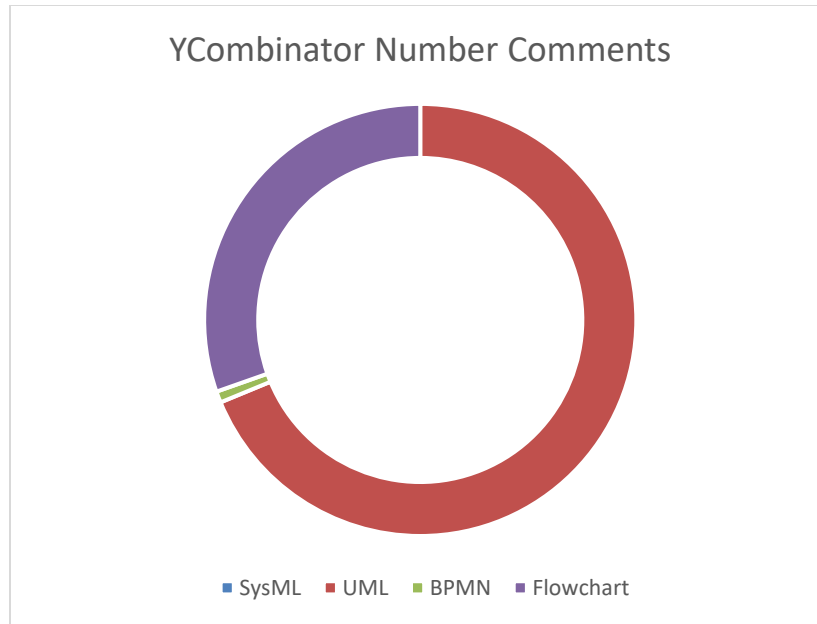


Figure 35. Number of Comments by Diagram Topic Type³

NLU uses pre-trained artificial intelligence models to discover meaning from textual input data. IBM Watson® provides a cloud platform and API to send sentences of text for analysis through Watson’s deep learning models (IBM n.d.). A feature of Watson is sentiment analysis which can predict the feeling of textual statements. A 1.0 would represent a very strong positive opinion, -1.0 conversely a strong negative opinion, while a 0.0 represents neutral feeling text. BPMN had the most positive score of 0.160, however, over a very small sample size. UML’s sentiment was positive at 0.067, but flowcharts had almost twice the positive sentiment at 0.127 as seen in Figure 36. This can be explained with a couple of top UML stories being “Has UML died without anyone noticing?” and “Why UML “Really” Died.” Interestingly, these headlines are consistent with the drop in search volume of the UML search volume seen earlier. Looking deeper into related all time search traffic volume, UML has reduced popularity by ~88% since it’s April 2004 peak (Dec. 5, 2021: Google Trends n.d.).

³ Data from Figures 34 and 35 come from Algolia, Search Hacker News. These data are a collection of the comments of the top ten stories in November 2021.

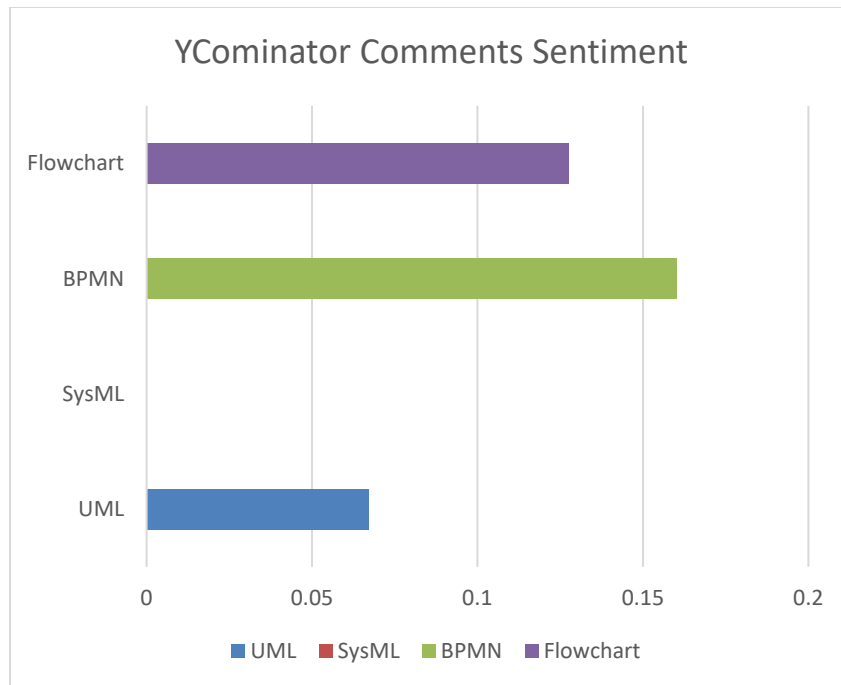


Figure 36. YCominator Sentiment Analysis Demonstrating Low Favor for UML Among Software Developers

b. Word Cloud Analysis

In addition to a sentiment analysis, NLP was graphed into word clouds to attempt to visually represent common terms used to describe popular terms. There was not enough data to develop word clouds of BPMN or SysML; however, flowcharts and UML were both graphed. Few negative terms were used to describe UML with “hard” being the only negative term associated with UML. No negative terms were identified in the flowchart word cloud. A more diverse set of terms were seen with flowcharts with top terms of “working,” “make,” and “code” standing out as seen in Figure 37.



Figure 37. Flowchart Word Cloud with Positive Words Commonly Appearing

Many terms were of near equal weight with UML (as seen in Figure 38); however, notable terms were “architects,” “business,” and “standard,” which are anticipated given UML’s purpose.



Figure 38. UML Word Cloud Demonstrating Only One Negative Word, “Hard”

F. SYSTEMS ENGINEERING PROFESSIONAL ARTICLES ANALYSIS

The International Council on Systems Engineering (INCOSE) publishes both a journal, *Systems Engineering*, and a magazine, *INSIGHT*. *Systems Engineering* has an impact factor of 1.449, while *INSIGHT* reaches over 10,000 INCOSE members.⁴ While the impact factor and readership may seem small, this is a significant fraction of the systems engineering community and the largest publications dedicated to the discipline.

An analysis, by keyword, with the Wiley keyword tool found SysML with the largest reach consistently across both publications. As seen in Figure 39, BPMN, primarily a language for business users, had the smallest reach across users. Interestingly, UML saw a large reach even within the systems engineering community. Flowcharts statistically were like BPMN, demonstrating a rather small reach of the systems engineering community.

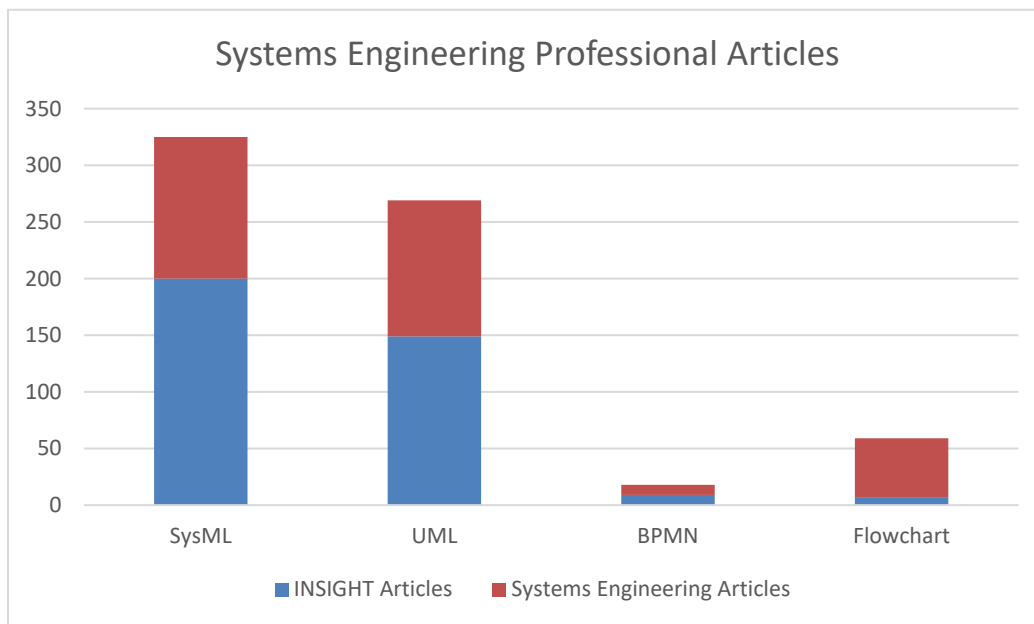


Figure 39. Systems Engineering Professional Articles Analysis⁵

⁴ These data were collected (on November 2021) using Wiley's search feature for journals: *Systems Engineering* and *INSIGHT*. URLs: <https://onlinelibrary.wiley.com/journal/15206858>; <https://onlinelibrary.wiley.com/journal/21564868>.

⁵ These data in Figure 38 were collected (on November 2021) using Wiley's search feature looking for systems engineering professional articles on SysML, UNL, BPMN, and Flowchart.

G. CHAPTER SUMMARY

The data capture of graphical modeling language data found that only BPMN, flowchart, SysML, and UML had enough data for analysis. On social networks (via Brand24), Google Analytics and Google Scholar, representing the general population, flowcharts were found to be the top term by a factor of four to five times the next most popular term. Via domain specific programming sources on GitHub, YCombinator, and HackerNews, UML was found to have the most relevant traffic. Similarly, SysML had the largest share via systems engineering professional publications. The sentiment towards each language found BPMN and flowcharts with a significantly higher score, approximately double, compared to UML.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CASE STUDY REPRESENTATIONS AND DISCUSSION

This chapter presents a case study, Wreck Interior Exploration Vehicle (WIEVLE), to analyze the application of MP and behavior graphical languages to full representation of the scenario. The prior popularity and sentiment analyzed BPMN, flowchart, SysML, and UML graphical notations are presented, in addition to EFFBD, IDEF0, and LML for historical comparison.

A. WIEVLE PROBLEM OVERVIEW

An unmanned underwater vehicle (UUV), the WIEVLE prototype is used as the basis for this MP graphical language scenario research. In this example scenario, the WIEVLE's primary mission is to discover objects of interest while maneuvering through compartments in an underwater wrecked submarine. The WIEVLE communicates for start-up and object decisions to an operator. Conversely, the WIEVLE will be capable of navigating its environment autonomously through onboard sensing (cameras) and an autonomous control computation facility (Eldred, Lussier, and Pollman 2021). The scenario is formally described with MP event grammar in the next section.

B. MP GRAMMAR REPRESENTATION

The WIEVLE MP grammar representation consists of three root events: Environment, WIEVLE AUV, and Operator, each of which performs the associated activities. The environment presents environment conditions to the WIEVLE within a loop. The operator starts the WIEVLE and commences the system's search pattern. Within a loop, the WIEVLE explores each compartment and then within another loop the WIEVLE navigates within a compartment. The environment's presented conditions are coordinated to the WIEVLE to decide on necessary action. The WIEVLE then manages the obstacle, manages a compartment entrance, manages a corner, proceeds forward (default), or manages an object. When objects of interest are found, the operator coordinates with the WIEVLE to collect information from the WIEVLE about the object and decides a course of action. These series of events continue until all objects are found within the submarine. This scenario presents loops, selection (OR), swimlanes (root events), and coordination/

synchronization of event features of MP. Notably, coordination requires communication across parallel root events and the ability to effect execution from other events in the series.

```
SCHEMA WIEVLE_Scenario;
```

```
ROOT Environment: (* Present_Environment_Conditions *);
```

```
ROOT WIEVLE_AUV: Commence_Search_Pattern
```

```
(* Explore_Compartments (* Navigate_Compartment  
Detect_Environment_Conditions ( Manage_Obstacle |  
Manage_Compartment_Entrance |Manage_Corner | Manage_Default |  
Manage_Object) *) *);
```

```
ROOT Operator: Start_WIEVLE_System (* Collect_Object_Information  
Decide_on_Object_Action *);
```

```
Manage_Obstacle: Avoid_Obstacle;
```

```
Manage_Compartment_Entrance: Sense_Compartment_Entrance_Detected
```

```
(Record_Location_of_New_Entrance |  
Determine_Compartment_Search_Complete);
```

```
Manage_Corner: {Sense_Corner_Boundary, Execute_Turn};
```

```
Manage_Object: Report_Object_Information Act_on_Object;
```

```
Manage_Default: Move_forward;
```

```
COORDINATE $x: Present_Environment_Conditions FROM Environment,
```

```
$y: Detect_Environment_Conditions FROM WIEVLE_AUV
```

```
DO
```

```
ADD $x PRECEDES $y;
```

```
OD;
```

```

COORDINATE $x: Start_WIEVLE_System FROM Operator,
    $y: Commence_Search_Pattern FROM WIEVLE_AUV
DO
    ADD $x PRECEDES $y;
OD;

COORDINATE $x: Report_Object_Information FROM WIEVLE_AUV,
    $y: Collect_Object_Information FROM Operator
DO
    ADD $x PRECEDES $y;
OD;

COORDINATE $x: Decide_on_Object_Action FROM Operator,
    $y: Act_on_Object FROM WIEVLE_AUV
DO
    ADD $x PRECEDES $y;
OD;

GLOBAL

SHOW ACTIVITY DIAGRAM Environment, WIEVLE_AUV, Operator;

```

The MP language strives for simplicity, yet its text language can be a barrier to non-coders. Furthermore, even coders experienced in popular programming languages have a learning curve as its pseudocode is designed for reasoning about behavior at a level above the implementation code. Users who understand its application space benefit from its formality that enables sets of scope-complete execution traces to be generated from the model. Of the models examined in this chapter, MP is the only language that “generates ...

scope-complete sets of traces...up to a scope limit decided by the user” (NPS Wiki n.d.d.). The schema structure also permits relaxation and restriction of interaction constraints to probe for unexpected but plausible event traces.

C. FLOWCHART REPRESENTATION

The first visual flow presented is the common flowchart, seen in Figure 40. In this example, three swimlanes, Environment, WIEVLE, and Operator, represent the performers of all processes. The start of execution is clearly indicated by the “Start” flowchart symbol.

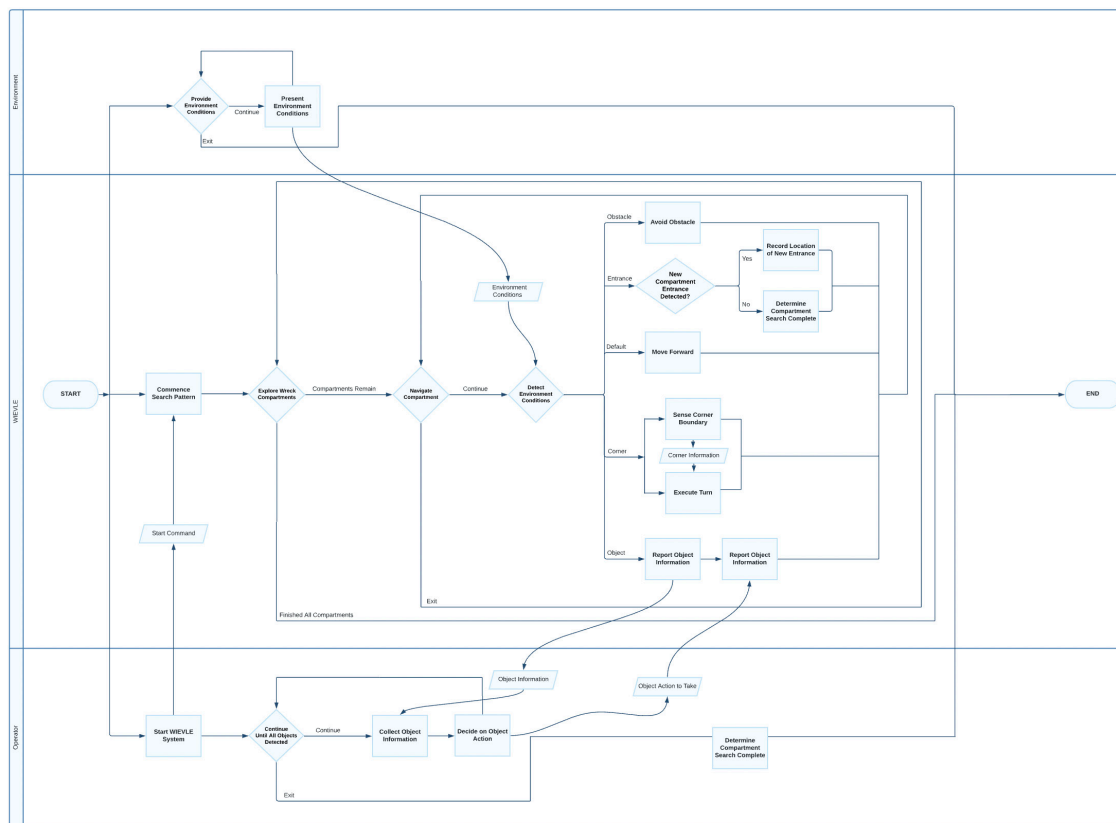


Figure 40. WIEVLE Scenario Flowchart Diagram

Each process is intended to be represented as a parallel thread of execution, and the use of a single line, in the vertical orientation, is used to represent this feature. Interestingly, the representation of parallel/concurrent flow is rather ambiguous with flowcharts amplified by community questions (Dabloons 2012). This likely can be traced to flowchart

history, with early mainframes only supporting execution of a single process at a time. However, the 1970 flowchart ANSI standard does contain a vertical orientation parallel-mode symbol with lines forking from a line to represent a fork and joining to a line to represent a join. Since the flowchart presented here is horizontally oriented, the junction of lines symbology is used for parallel paths.

Each parallel path is coordinated with input/output symbols that are output from generating processes and input by received processes. The flowchart symbology includes the parallelogram as shown in this example for input/outputs in addition to document, magnetic tape, punched tape, and many other symbols to represent specialized input/output symbols. Unfortunately, the execution of input/output is ambiguous. Are the inputs received after the generated process is completed, streamed duration execution, or at some point in mid-execution? The ANSI standard does not clarify execution semantics, leaving the assumption of a finish to start relationship with input/outputs.

Decisions are represented by a diamond and indicate the selection of a single path. These are inclusive ORs and can be used for both OR and LOOP control flow. An example LOOP is “Explore Wreck Compartments” which continues until no more compartments remain, indicated by labels on the control flow branches. An example OR is “New Compartment Entrance Detected?” with a distinctive “yes/no” branch for selection.

The flow of parallel paths is rejoined using a line junction, unfortunately this junction will never occur. The environment and operator are effectively in infinite loops with no ability to terminate or kill these processes. An alternative representation, dropping separate processes for each performer, is presented in Figure 41.

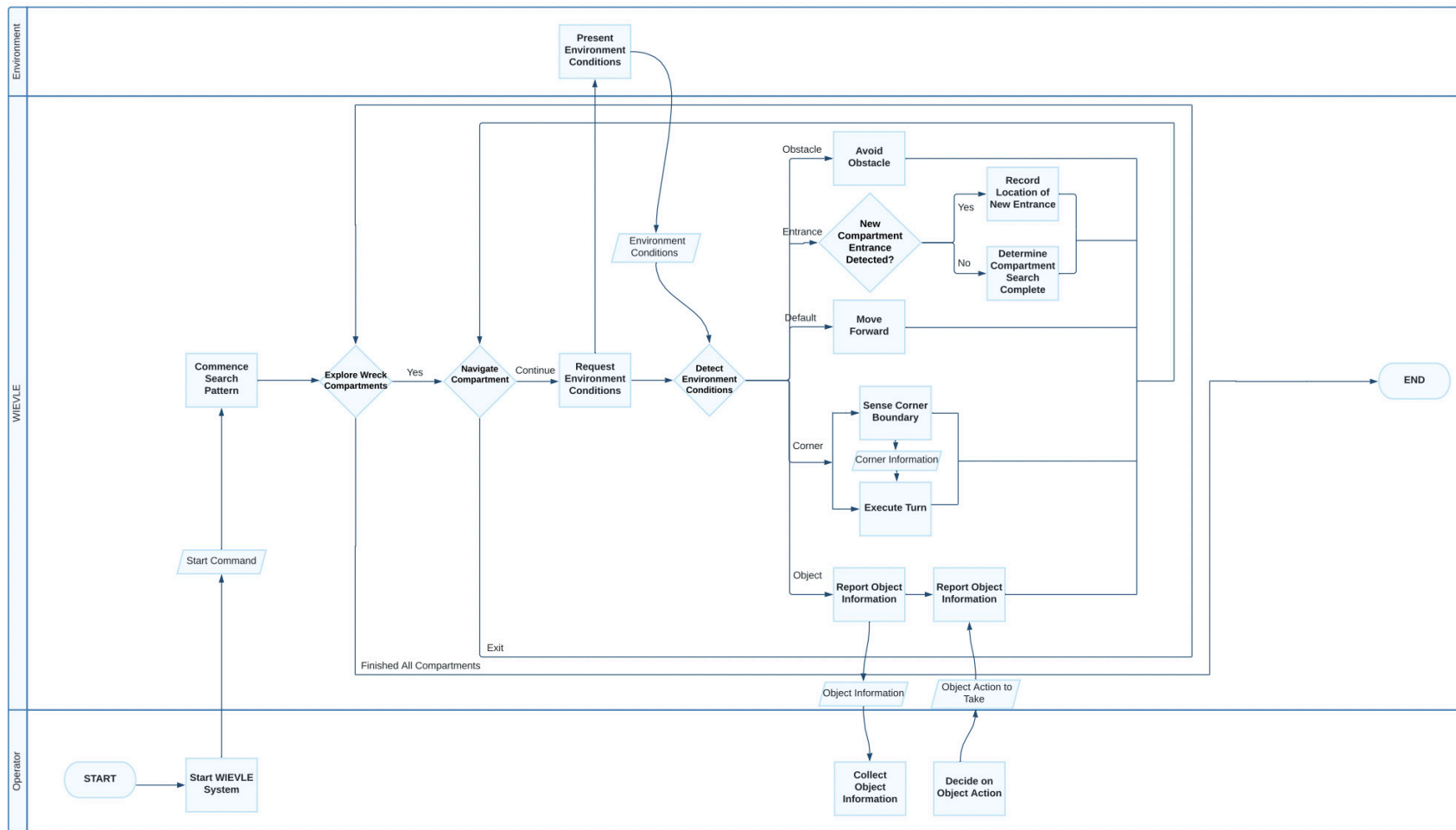


Figure 41. WIEVLE Scenario Sequential Flowchart Diagram

This approach uses purely sequential flow instead to represent the WIEVLE scenario. This solves the coordination issue and negates the need to represent controlled parallel branch interruption. Unfortunately, this removes the ability for the modeler to explicitly identify parallelism and forces a more simplistic view of the problem.

D. IDEF0 REPRESENTATION

In this WIEVLE example, seen in Figure 42, the IDEF0 is used to represent inputs, outputs, and mechanisms (performers) of the scenario. IDEF0 diagrams do not have a facility to represent control flow, decision nodes, nor control nodes within its symbology. This IDEF0, however, represents a condensed picture of key flow interactions between primary performers. Unfortunately, without the ability to represent predecessor relationships, IDEF0 cannot represent MP predecessor logic, and inclusion can only be represented through sets of IDEF0 diagrams.

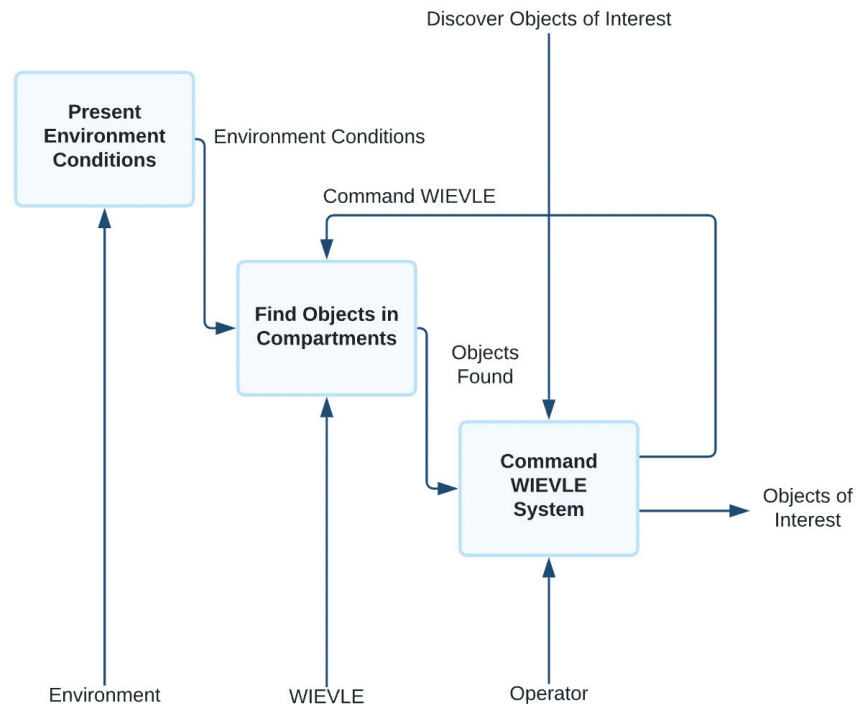


Figure 42. WIEVLE Scenario IDEF0 Diagram

E. EFFBD REPRESENTATION

As discussed in Enhanced Functional Flow Block Diagrams (EFFBD), the EFFBD diagram includes both control flow from the FFBD and items to represent input/output. Unlike the flowchart and IDEF0 diagram discussed earlier, the EFFBD is fully executable in tools like Vitech CORE® and Vitech GENESYS®. This allows for model logic to be simulated for correctness, evaluate the system performance, and ensure the modeling notation is computationally correct.

The WIEVEL EFFBD, seen in Figure43, begins and ends with an AND control node. Neither the FFBD nor EFFBD notation includes an explicit start or end node, which could cause confusion to readers inexperienced with this type of model. The parallel forks into environment, WIEVLE, and operator branches that execute concurrently. A fourth branch, kill, is included. This branch is an important EFFBD feature to allow termination of all other branches within an AND when the kill branch completes (Vitech Corporation 2015). This allows for the environment and operator to occur in endless loops until the WIEVLE mission is complete.

Within the WIEVLE, this EFFBD uses a combination of loop “LP” and multi-exit functions to represent loop. The loop executes continuously until the multi-exit function branch “Finished all compartments” is selected. This branch issues a loop exit “LE” command to stop loop execution. Conversely, the environment and operator loops do not contain a multi-exit function nor loop exit and will loop forever until terminated by the kill branch within their enclosing AND.

EFFBD has two facilities for selection: an OR control node and a multi-exit function. The OR node is not typically labeled itself and cannot receive or generate items. In this example, ORs are all represented by multi-exit functions, which are a special function that can choose between multiple branches for successor execution. The “Detect Environment Conditions” multi-exit function receives input from the environment and selects a branch for further execution. The advantage of multi-exit function in this scenario is clear labeling of decisions, receipt of items, and further traceability of decisions to requirements.

While traced to 1950s symbology, this EFFBD represents a surprisingly modern view in terms of the ability to represent logically elements, data items, and ability to execute computationally. This is likely true due to historical inheritance of modern viewpoints (BPMN, LML, SysML, and UML) from EFFBDs. Usability-wise, the “LP” and “LE” control nodes are unlikely to be clear to readers without EFFBD experience or a clearly marked legend. Kill is a functional action while the performers are their executing components; this makes the kill branch notation also awkward. Notably, an implementation deficiency represents the performers on a branch as simply textual labels and not components, missing the ability to strongly trace components to their functions automatically.

F. UML SEQUENCE DIAGRAM REPRESENTATION

The sequence diagram, represented in Figure 44, demonstrates interactions between lifelines (performers). The parallel paths are implicit between each lifeline, and the UML sequence diagram notation includes an explicit “par” interaction fragment to use where necessary. The start of the diagram is indicated clearly with an initial node. The first

message, “Start Command,” is sent by the operator to the WIEVLE. UML sequence diagrams use a verb/action format to represent the sending of messages, while noun names for the message input/output itself are omitted. The next element is the loop frame surrounding the rest of the diagram which continues until all compartments are searched. The interaction with the environment occurs on each loop of the scenario. If any object is detected, the WIEVLE and operator exchange object information and a decision for the WIEVLE to act upon. This “opt” fragment is equivalent to the optional event within MP.

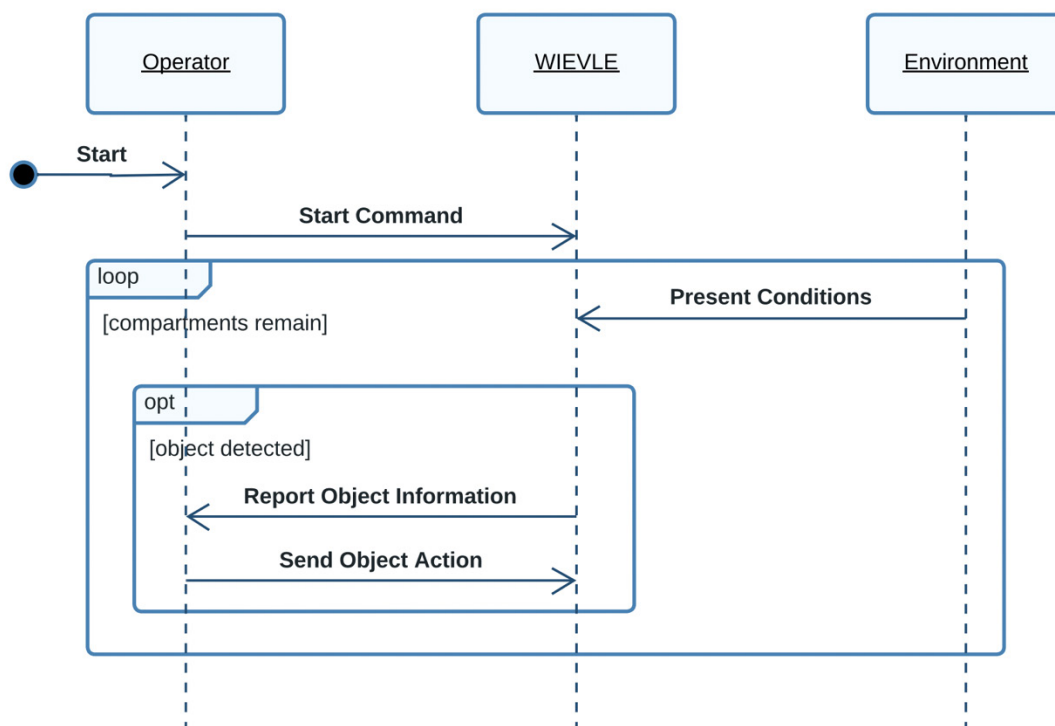


Figure 44. WIEVLE Scenario Sequence Diagram

The sequence diagram does not specify control flow with activities, as with the activity diagram, and therefore is more analogous to SysML item flow. While it is possible to send a message to yourself (a message on a lifeline), logical control node elements cannot be represented this way. A relative sequence diagram is however utilized as an output format of MP event generate to represent the path of event execution for a given scenario.

G. SYSML/UML REPRESENTATION

As discussed earlier, UML has undergone a major revision to its symbology in UML2 and a major revision is ongoing at the time of writing for SysML (SysML 2). For simplicity, the UML2 activity diagram notation is used in this WIEVLE scenario seen in Figure 45. Three swimlanes represent the performing blocks within this diagram. Notably, tool support of block association to actions/activities varies (SysML Forum 2012).

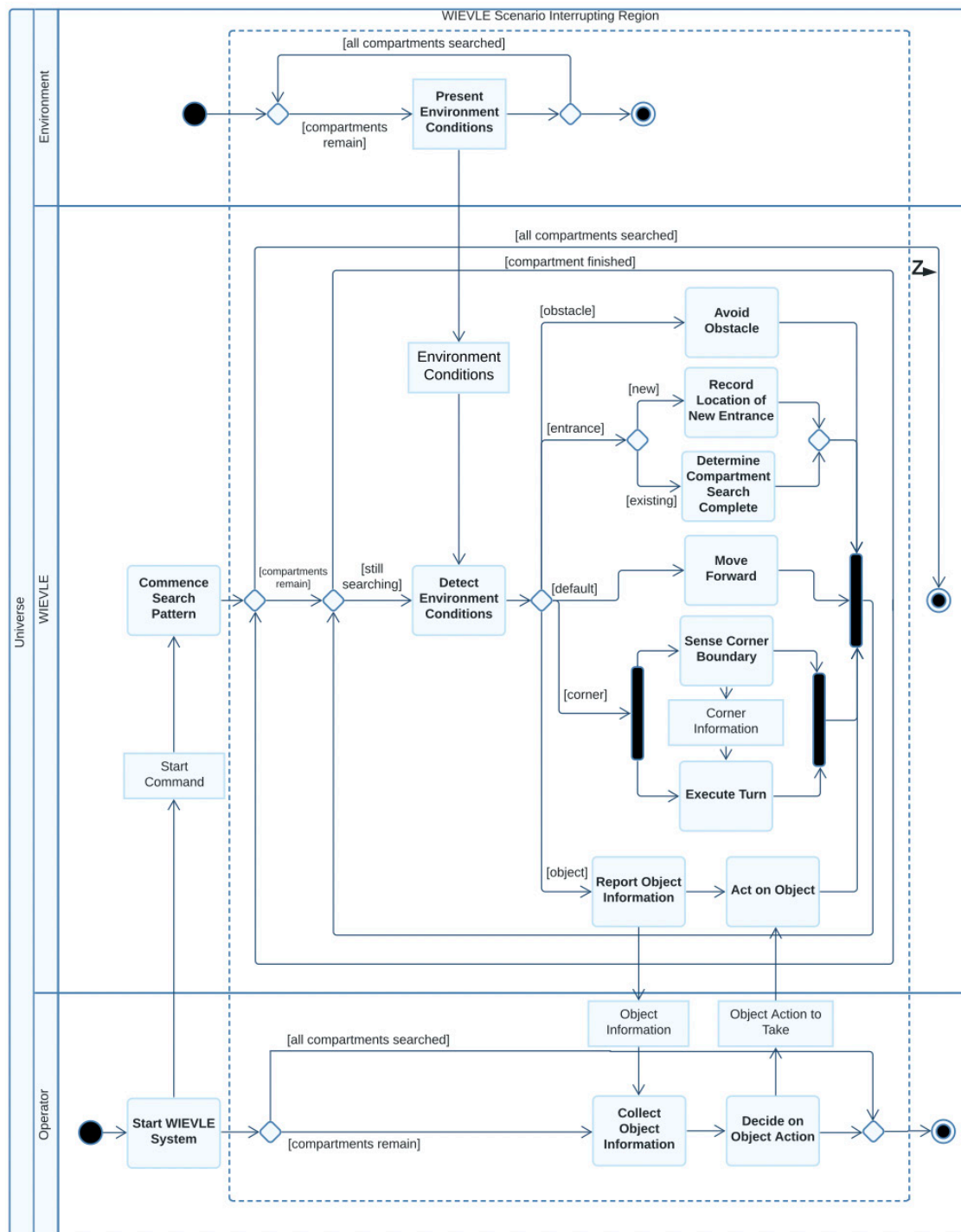


Figure 45. WIEVLE Scenario SysML/UML Interrupting Region Diagram

This diagram begins with a clear initial node on the left side of the diagram. The rest of the diagram (except for the final node on the right) is surrounded by an interrupting region. The interrupting region allows for an exiting edge (from the region) to kill all flow within the interrupting region (Friedenthal, Moore, and Steiner 2015, 223). This kill instruction is represented in the scenario via a lightning bolt annotation on the edge exiting the region. Unfortunately, this annotation is not supported within the fUML, UML subset, (OMG 2021b) nor the popular SysML/UML tool MagicDraw (MBE Execution 2020). The fUML authors state this representation is “considered to be more appropriate for higher level process modeling and outside the scope of fUML” (OMG 2021b, 37). This is unfortunate, as a similar feature, kill branches is present in the classical EFFBD representation. In general, the ability to model diagrammatic features without execution support generates risk of non-sensible diagram development.

To support executability, an alternative representation is demonstrated in Figure 47. This approach removes the interrupting region and models the scenario with parallel forks used to coordinate logic between diagrams. This secondary representation succeeds from the initial node to an action that starts the WIEVLE system. This outputs an object node (input/output) start command which is received by the commence start command action on the WIEVLE. The WIEVLE enters a branch node to represent a decision if compartments remain or if all compartments have been searched. A second branch node controls searching within the compartment itself. A parallel branch, like the style seen on the flowchart, is used to fork execution to the environment swimlane. Execution continues through each compartment and ends once all compartments have been searched.

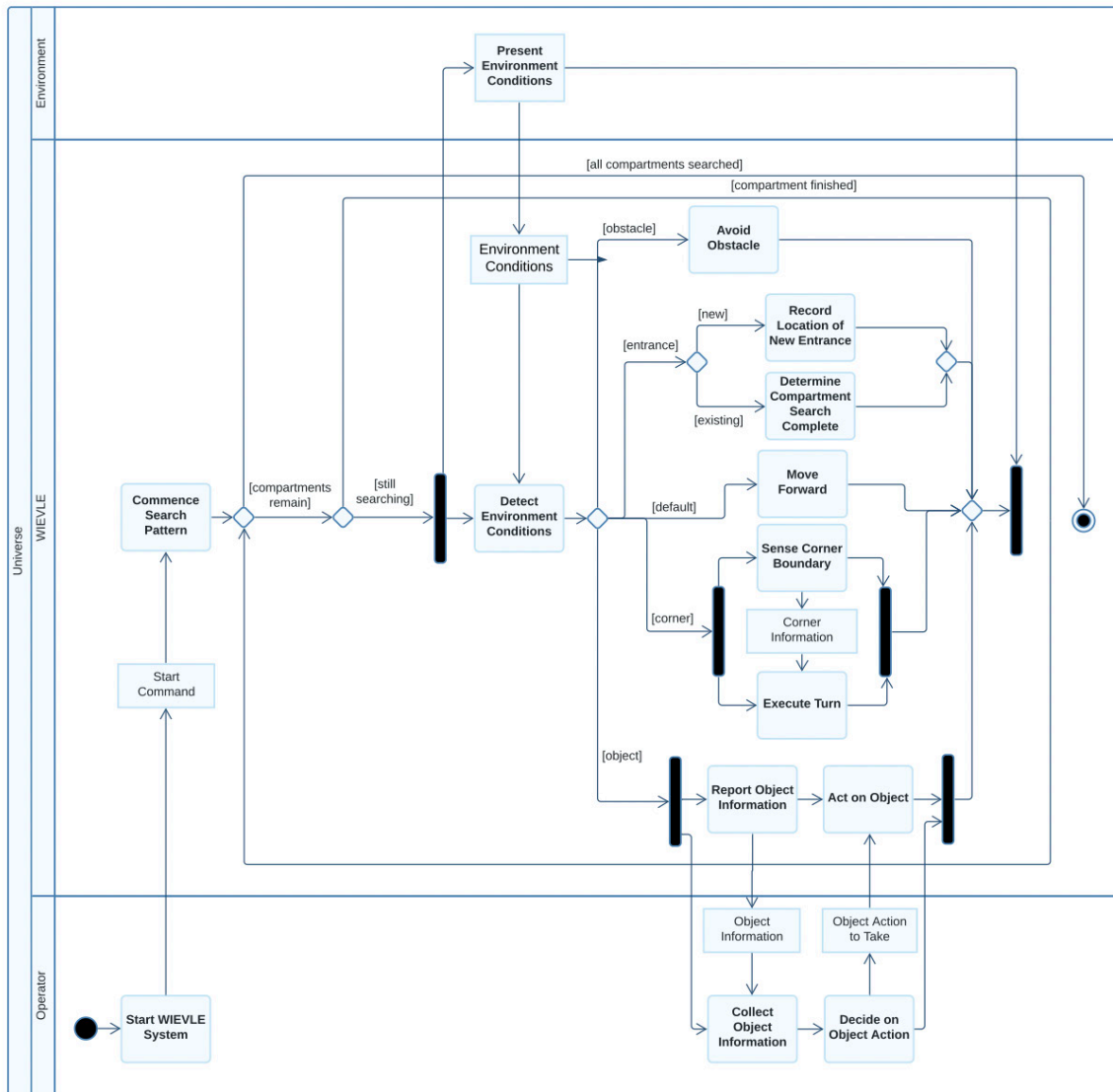


Figure 46. WIEVLE Scenario fUML-Compatible SysML/UML Diagram

The use of diamonds to represent decisions, rectangles to represent input/outputs, swimlanes, and vertical bars to represent parallel branches are similar to flowcharts and likely intuitive to most readers. The fUML compatible notation is supported by tool execution engines, but requires crossing swimlanes with control flow, which is not ideal as even though each swimlane is a separate performer, the entire diagram is now effectively modeled together as single performer. Encouragingly, the notation is compact and easy to fit within a small space graphically.

H. BPMN REPRESENTATION

The BPMN WIEVLE representation, seen in Figure 47, begins with a start event control node on the left of the diagram. As there is no kill branch, synchronization, nor interrupting region feature in BPMN, execution occurs sequentially in this example. A modeler could also decide to control decisions across process swimlanes with code; however, this is a clear, immediate limitation of BPMN.

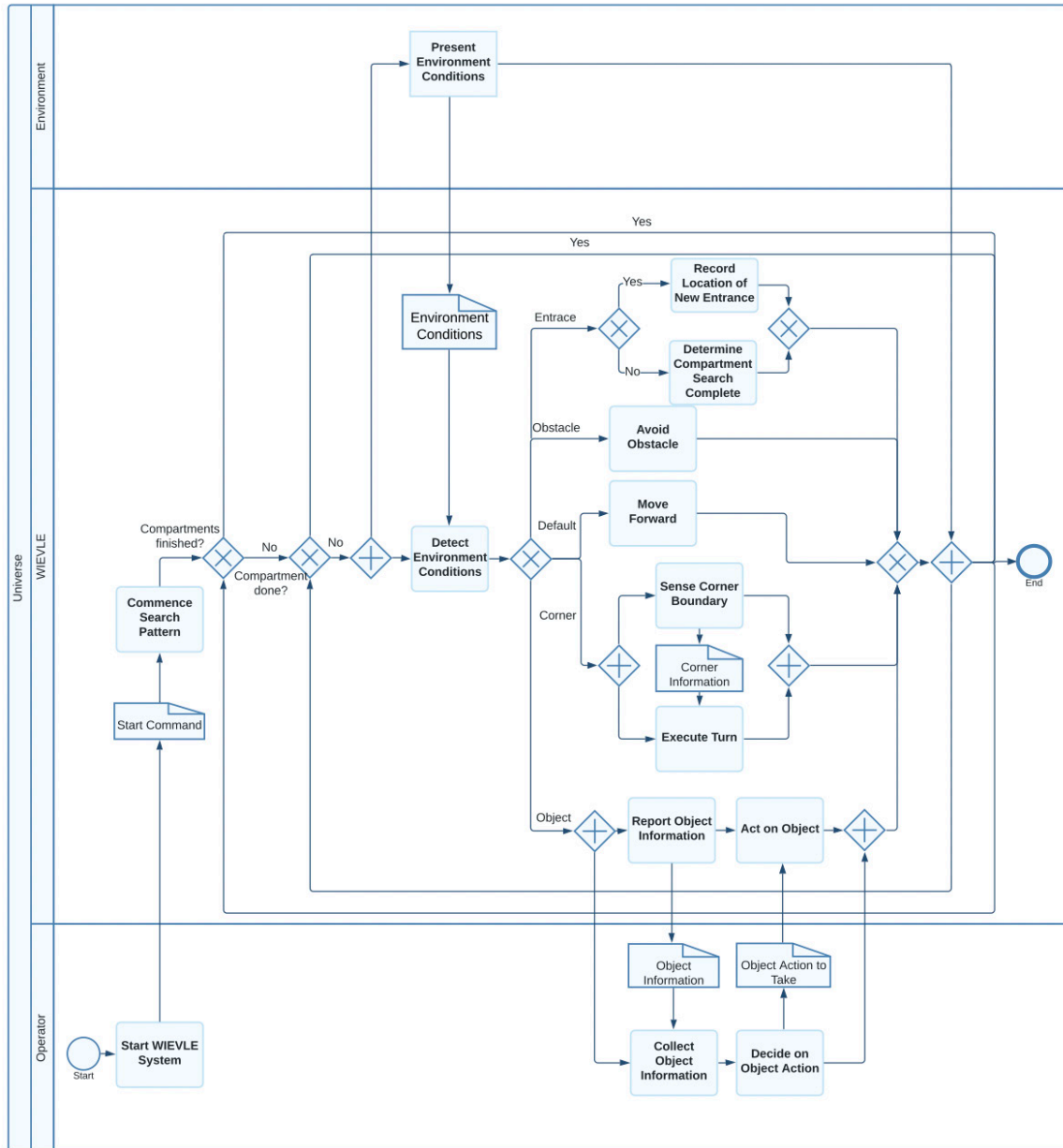


Figure 47. WIEVLE Scenario BPMN Diagram

The first process starts the WIEVLE system from the operator, sends a start command data object to the WIEVLE. This is received by commence search pattern. Receiving processes must wait for a generated process to complete prior to execution. BPMN diagrams are fully executable with a few execution engines available via open source and commercial channels (Predictive Analytics Today n.d.).

The first decision point / control node is a loop to explore all compartments. This loop is controlled by an exclusive OR represented by a BPMN gateway with an “X” symbol.” To allow for environment conditions to “kick off” on each loop iteration, a parallel “+” gateway is used. The use of both a “+” and “X” to represent gateways can cause visual confusion, as at a quick glance both gateways look similar. Additionally, it is unlikely a reader (without BPMN experience) would intuitively know what a “+” or “X” means.

I. LML ACTION DIAGRAM REPRESENTATION

The LML action diagram representation is visually like prior flow-based representations and, as discussed earlier, inherits its look and feel from prior diagram symbology. LML requires actions to represent all control nodes (except for parallel branches) to allow for traceability to other diagram elements.

The WIEVLE action diagram, seen in Figure 48 begins with a clear start control node on the left. This enters a parallel represented by a single edge entering a vertical line succeeding into multiple paths of execution. This is efficiently an implicit parallel as the only indication of parallelism is line routing on the diagram, which could cause confusion to readers without clear labeling. Many languages (BPMN, flowcharts, SYSML, and UML) use swimlanes to represent these ROOT event paths.

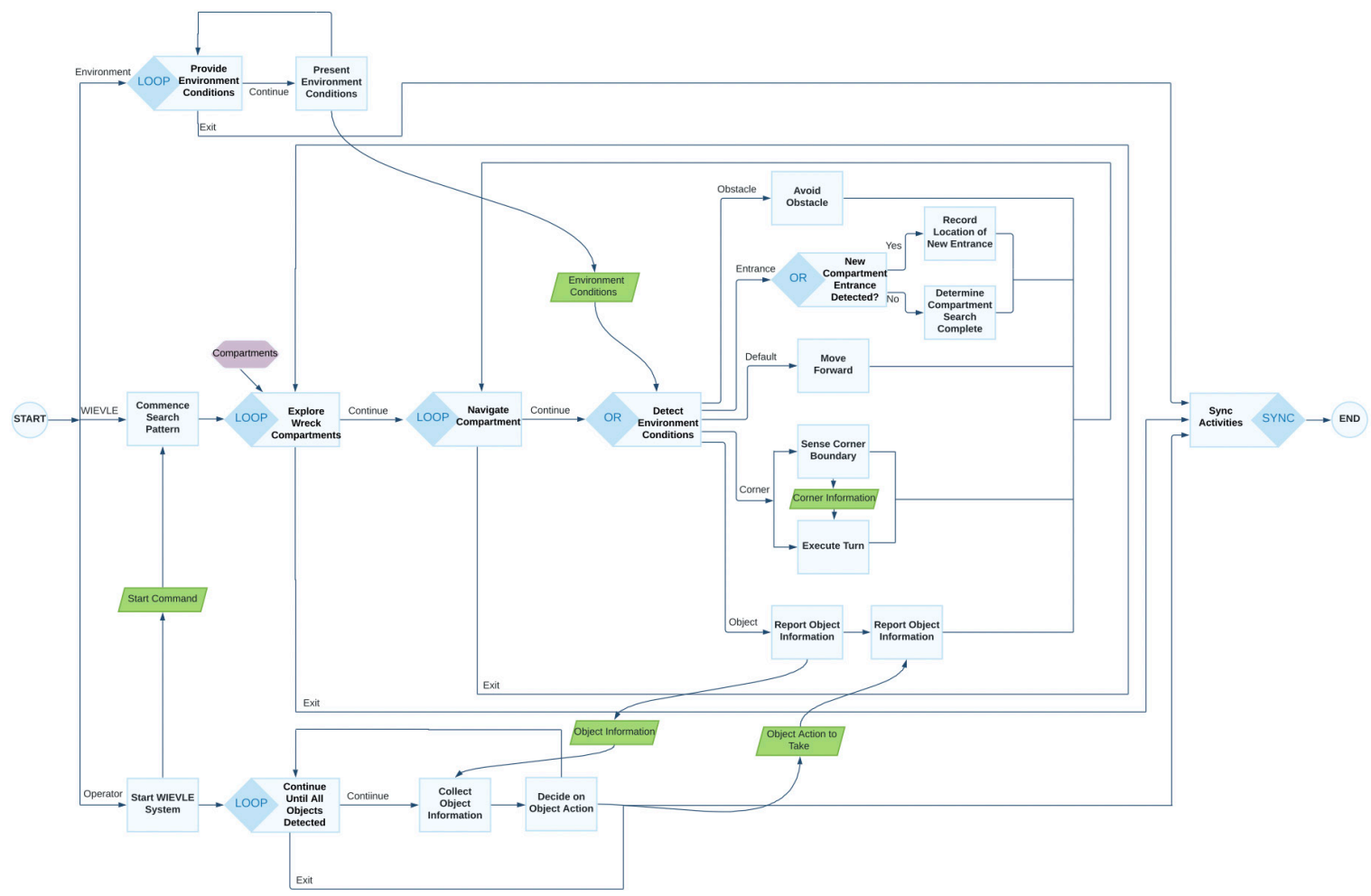


Figure 48. WIEVLE Scenario LML Action Diagram

The branches are labeled environment, WIEVLE, and operator to represent the performers of the associated actions. Importantly, LML concisely uses its asset class entities for these branches, using entities rather than uncontrolled text for labeling. This also allows for automatic traceability of entities within these control flows.

Input/outputs, represented by parallelograms, are the same as used by flowcharts mentioned earlier. The LML specification does not specify execution semantics of input/outputs. Tool providers have chosen a finish to start meaning (Innoslate Help Center 2021).

The kill branch feature, discussed earlier, is represented by a sync point in this scenario. Like the kill branch the sync point kills execution of all paths once it completes execution. Unlike the earlier kill branch, the sync kills all branches once one branch completes execution. This reduces the visual complexity and awkward kill branch from the EFFBD representation.

J. CHAPTER SUMMARY

The WIEVLE scenarios depicted demonstrate graphical features of each diagramming language. IDEF0 and sequence notations did not demonstrate precedence features within a performer (MP inclusion) to model control flow. Flowcharts, BPMN, EFFBD, LML, SysML, and UML were all capable of representing the scenario. Flowcharts can represent the problem but are forced into a purely sequential representation. Additionally, unlike flowchart derived notations, flowcharts lack formalism which could cause misinterpretation by diagram readers. SysML and UML lacked interrupting region support/kill branch functionality similar to EFFBDs and LML execution engines. The SysML and UML representations were the most compact visually.

V. MP GRAPHICAL LANGUAGE

This chapter combines data from open-source data analysis and the WIEVLE case study to select a graphical representation for MP. Furthermore, this chapter introduces both the software architecture and functionality of a new MP GUI. MP GUI's support for the six of the eight primary MP event grammar rule patterns is presented, along with a discussion of functionality not yet implemented. Lastly, the chapter presents the use of the MP-Firebird integration to generate exhaustive sets of traces up to the user-defined scope.

A. LANGUAGE SELECTION

Mikhail Auguston, the original creator of MP, developed the Visual Data Flow Language, to formalize graphical notations. Auguston's language supported common control operations in addition to graphical addition, subtraction, and synchronization of data across diagram elements (Auguston and Delgado 1997). Furthermore, the language included a specific element to buffer data to support streaming, a feature just now moved as a default within SysML v2. Visual Data Flow Language differed greatly from SysML and other languages previously discussed. The Visual Data Flow Language supported addition, multiplication, matrices, and mathematical expressions, which are currently not directly possible in any other common graphical language without scripting. This language did not prove popular with modelers and therefore it was not included in the set considered as a language target for the MP graphical interface.

Each language presents its own unique set of advantages and disadvantages as seen in Table 3. Note that Table 3 and Table 4 were developed with the Pugh matrix template tool (CIToolkit n.d.). Flowcharts are still extremely popular with the public, compared to formal languages, but bring ambiguity, lack expressiveness, and cannot be executed. BPMN is popular within the business community, adds executability over flowcharts; however, is generally not popular with the systems engineering community. Similarly, LML brings rich executability, formal functional expressiveness, and continuous improvements, but as with BPMN is not as popular in the systems engineering community. Conversely, SysML is widely used in the systems engineering community, but

uncommonly used by software developers and general engineers. UML is the most widely used language by software developers; however, UML also receives rather poor sentiment from the development community. While EFFBD and IDEF0 continue to inspire many languages, each is no longer under development and their popularity appears to be fading.

Table 3. Initial Language Selection Table

Criteria	Flowchart	BPMN	SysML	UML	EFFBD	LML	IDEF0
Expressiveness	-	0	+	+	0	+	-
Executability	-	+	0	0	+	+	-
Academic Popularity	+	0	-	+	-	-	-
General Popularity	+	-	-	0	-	-	-
General Sentiment	-	+	0	-	0	0	0
Programmer Popularity	0	-	-	+	-	-	-
Programmer Sentiment	+	+	0	-	0	0	0
SE Popularity	-	-	+	0	-	0	-
Recent Development	-	0	+	0	-	+	-

Using the data from studies above, LML, SysML, and UML are statistically in a three-way tie, with a slight one-point edge for UML. This is unsurprising, as like programming languages, different languages are better suited for different modeling paradigms. Combining the score for both SysML and UML into a combined category SysML/UML, bringing SysML's more relatable syntax, while maintaining compatibility with UML, delivers the most compelling choice for an initial MP user interface. Notably, no language is necessarily "superior" for behavior modeling, and no language dominates across all categories. Importantly, future researchers should consider BPMN and LML

compatibility. BPMN is popular within the business community and similar to flowcharts, while LML is fully executable and a popular language for functional analysis.

Table 4. Final Language Selection Table

Criteria	SysML	UML	LML	SysML/UML
Expressiveness	+	+	+	+
Executability	0	0	+	0
Academic Popularity	-	+	-	+
General Popularity	-	0	-	0
General Sentiment	0	-	0	+
Programmer Popularity	-	+	-	+
Programmer Sentiment	0	-	0	-
SE Popularity	+	0	0	+
Recent Development	+	0	+	+

Given the selection of SysML for this study, the new KerML language that SysML v2 is built upon may provide an additional advantage. KerML's textual notation is formal, validatable by a standard OMG reference implementation, and automatically generatable. An MP graphical interface could jointly generate both KerML code and MP code. Furthermore, similar capabilities may also be possible with BPMN and BPEL to bring MP capabilities to a wider community.

B. SOFTWARE FUNCTIONAL ARCHITECTURE

The ecosystem created to test SysML as a graphical language for MP includes four primary performers. The first is the user who utilizes Seabird to draw SysML diagrams and

generate MP code. The second is MP-Firebird, which compiles the MP code and graphically displays MP compilation results to the end user. Furthermore, MP-Firebird includes live intelligent suggestions for users like a modern development environment. Thirdly, a server-side backend, hosted on a cloud computing service, provides backend controller services and serves the front-end application to the user. Finally, Seabird is a new frontend application developed by the author, which allows a user to draw SysML activity diagrams, transpile MP code, and import diagrams from prior development sessions.

The inputs and outputs between these performers are shown in Figure 49. Most interactions occur directly between the user and Seabird. Notably, to reduce complexity and increase security, no user data is persisted to the cloud service. Similarly, for public users, MP-Firebird also does not permanently persist user data.

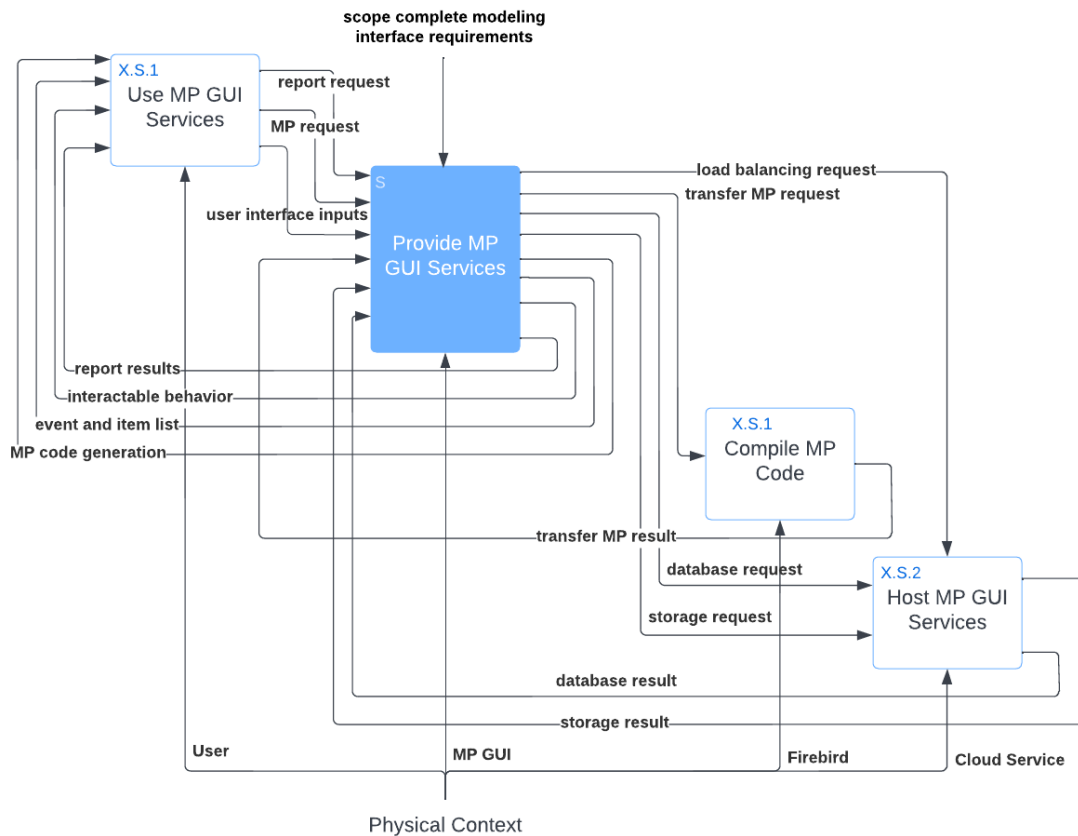


Figure 49. Seabird Functional Context IDEF0

Seabird itself is composed of four primary performers: a server-side backed module, frontend module, MP code generator, and a reports module as seen in Figure 50. The backend serves the Seabird application upon user request and handles automatic load balancing with a serverless implementation. The front-end module interprets user drag/drop commands and displays the result to the user. The MP Code Generator or transpiler, is a separate frontend module that supports transpilation of generic control flows into MP code. Interaction between Seabird and the transpiler is performed with JSON payload, which is separate from Seabird's interface logic. This separation is intentional and non-SysML specific, to allow for systems engineering tool vendors to use this source code as a reference implementation for their own applications. Lastly, a reports module provides reporting services to generate lossless PNGs, lossy JPEGs, and vector graphics SVGs.

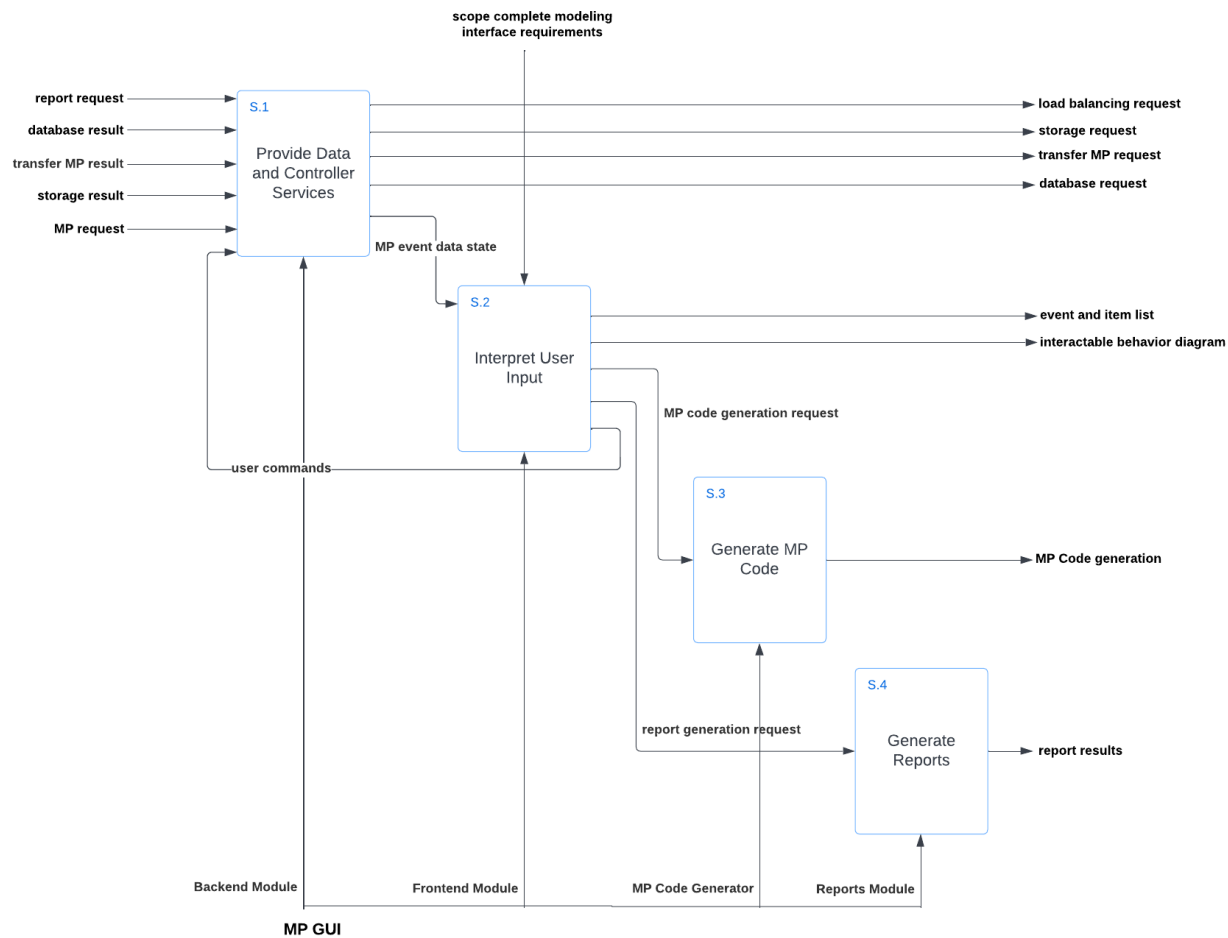


Figure 50. Open Seabird Functional Architecture IDEF0

C. SYSML IMPLEMENTATION TO MP MAPPING

1. Root Events

Seabird uses SysML swimlanes to visualize roots of an MP schema. Each root event is represented by exactly one swimlane within a pool construct. The pool's name specifies the name of the schema containing the MP code. Figure 51 demonstrates this capability. The NPS wiki demonstrates event interactions with a simple message flow coding example. Seabird is capable of automatically regenerating this example from a diagram (NPS Wiki n.d.c). In this example `simple_message_flow` is the name of the MP schema while `sender` and `receiver` represent root events. Seabird will automatically transpile this diagram into the following: `SCHEMA simple_message_flow ROOT Sender;; ROOT Receiver;;`.

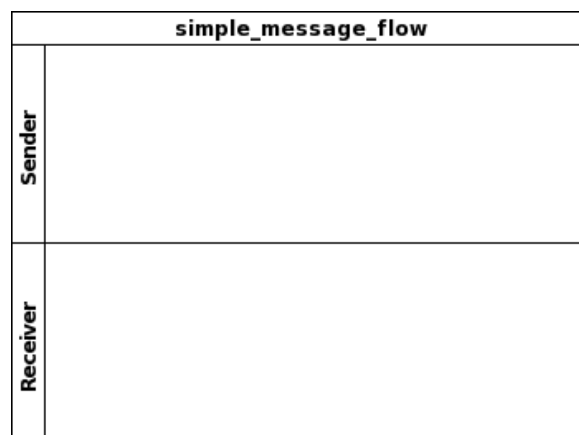


Figure 51. Root Event Example in Seabird

2. Atomic Events

SysML uses actions to represent the performance of a process. Seabird uses SysML actions to represent MP events as seen in Figure 52 for the action `send`. Users should include actions within a swimlane so that the generated MP code includes a named root event. If an action is not included within a swimlane, a default root event is automatically created to attach these dangling actions. Importantly, while not present as a separate concept from an event in MP, it is highly recommended each swimlane include an initial (start) and final (end) node. These nodes are optional for simple diagrams but are needed

by the transpiler for more complex logic, such as loops. Figure 52 will generate the following MP code “`SCHEMA simple_message_flow ROOT Sender: send;`”

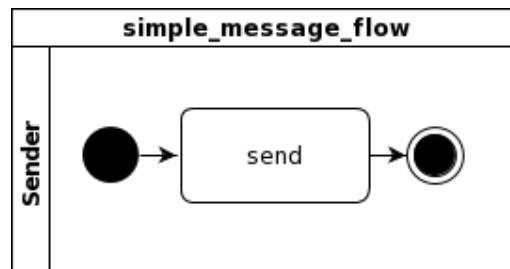


Figure 52. Atom Action Example in Seabird

3. Decomposition and Flow

Seabird’s transpiler engine supports both precedence and inclusion relations. Precedence relations use the SysML succession flow lines to capture the proper ordering of actions. This flow is restricted for use within the same swimlane and root event. Input/output flow and MP coordination are required to communicate between swimlanes and root events. respectively. A simple succession flow is seen in Figure 53 with initialize completing before the send action can start. Seabird does not support decomposition of diagrams (inclusion), and therefore, this feature is not currently available to user interface users. Figure 53 will generate the following MP code “`SCHEMA simple_message_flow ROOT Sender: initialize send;`”

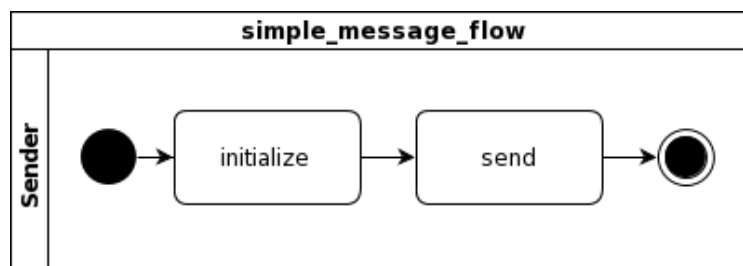


Figure 53. Sequential Action Flow in Seabird

4. Parallel Logic

Parallel logic is effectively represented by each separate swimlane itself, however, at times it is necessary to express parallelism within a performer. SysML uses the fork and join construct to represent this behavior. The send and queue events operate in parallel and are independent of each other's execution as seen in Figure 54. MP expresses parallelism within an ordered set. Accordingly, the generated MP code from Seabird is “`SCHEMA simple_message_flow ROOT Sender: { send , queue };`” Replication, a feature introduced by the EFFBD diagram, is not supported directly by Seabird, but can be easily added with “`{+ +}`” within MP-Firebird.

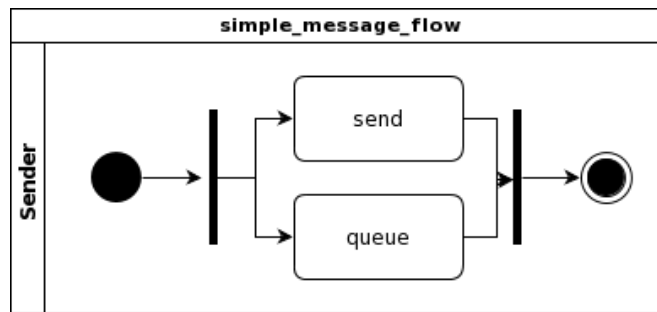


Figure 54. Parallel Logic in Seabird

5. Event Selection / OR Logic

SysML uses a decision node to represent selection of a path within the diagram. MP uses the alternate event syntax “(send | store)” to allow for choice events. Figure 55 demonstrates a Seabird diagram with a simple alternate logic decision. The send action is selected if it is needed now, otherwise if the sender can wait, the message is stored in a buffer. Interestingly, SysML uses the needed/wait guards on each path from a decision node to signify the condition which must be met to enter a path. While MP does not use guards as a separate concept from events, each guard is compiled as an additional event, in this example prior to send and store in buffer respectively. Seabird's transpilation of this diagram is “`SCHEMA simple_message_flow ROOT Sender: ( needed_now send | can_wait store_in_buffer ) ;`” Seabird can represent optional events through a diagram such as Figure 56.

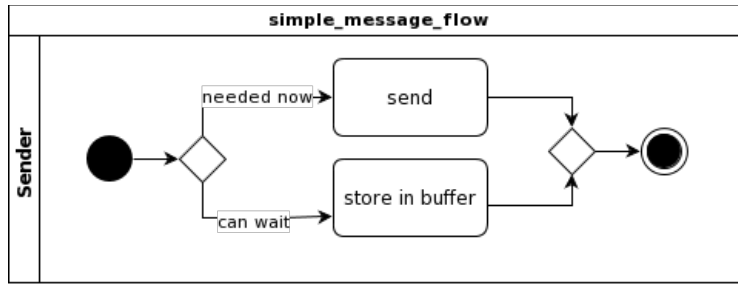


Figure 55. Event Selection Example in Seabird

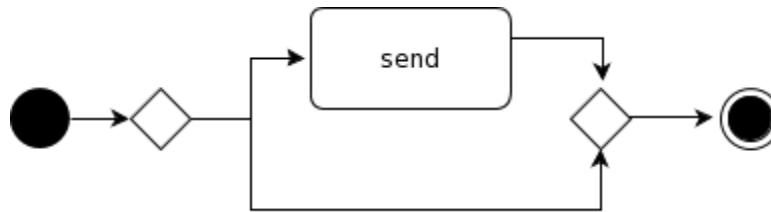


Figure 56. Optional Event Example in Seabird

6. Coordination

Each swimlane and parallel unordered event cannot use succession flow relationships. SysML uses item flow to express connections outside of succession, while MP utilizes a coordination feature. Seabird allows symbolizing this functionality with an object node and item flow. Item flows are not supported between control actions within SysML and must be between atomic actions.

Figure 57 represents message flow between a sender and receiver root event swimlane. The sender sends a message to the receiver through its receive action. The receive action must wait for the send action to complete before it can start execution. The resultant code follows:

```

“SCHEMA simple_message_flow

ROOT Sender: send;

ROOT Receiver: receive;

COORDINATE $x: send FROM Sender ,

    $y: receive FROM Receiver

DO ADD $x PRECEDES $y; OD; “

```

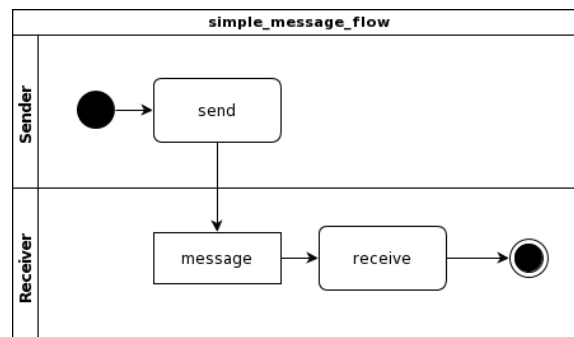


Figure 57. Coordination Example Using Input/Outputs in Seabird

7. Loop and Iteration

MP supports both a zero or more iteration “(* *)” and a one or more iteration “(+ +)” construct for loops. SysML’s decision construct can be placed before or after the contents of a loop. This difference in decision node placement allows for symbolization of both MP concepts. Figure 58 demonstrates a loop immediately following the initial (start) node on the sender. This loop can either continue sending and execute the send action or stop sending and terminate the loop. The send action is optional and is executed zero or more times. Similarly, the receiver’s receive action may also be executed zero or more times. A message can be passed through each loop to represent continuous sending and receiving of messages. The resultant MP code for this diagram is:

```

“SCHEMA simple_message_flow

ROOT Sender: (* continue_sending send *) stop_sending;

ROOT Receiver: (* continue_receiving receive *) stop_receiving;

COORDINATE $x: send FROM Sender ,

           $y: receive FROM Receiver

DO ADD $x PRECEDES $y; OD;”

```

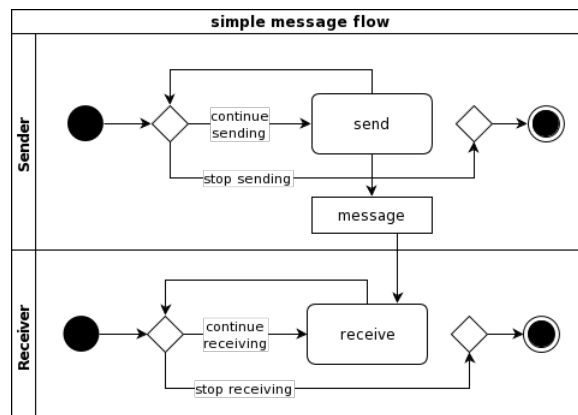


Figure 58. Zero or More Loop Example in Seabird

A modification of this diagram can be made to transpose the decision and merge nodes. In Figure 59, the sender will always send one message first. After sending this message a decision is presented to either continue sending the message or stop sending messages altogether. The receiver in the second swimlane follows the same logic pattern. Seabird generates the following MP code to represent one or more loops:

```

“SCHEMA simple_message_flowspan

ROOT Sender: (+ send continue_sending +) stop_sending;

ROOT Receiver: (+ receive continue_receiving +) stop_receiving;

COORDINATE $x: send FROM Sender ,

           $y: receive FROM Receiver

DO ADD $x PRECEDES $y; OD;”

```

8. MP Compiler Integration

Seabird offers a live integration with MP Firebird to compile source code generated by Seabird's transpiler. To access this capability a modeler selects "Generate" and then "Generate MP." MP modelers can then modify code generated from Seabird. After the modeler reviews and makes any necessary changes, selecting "OK" will transfer this code to MP Firebird. This integration allows for a complete ecosystem to model emergent behavior.

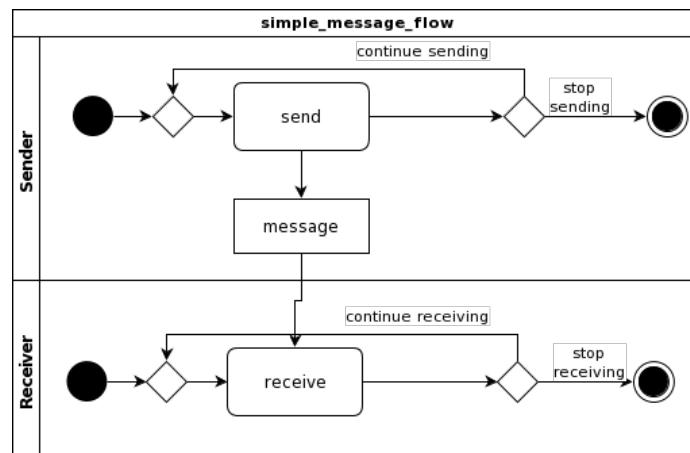


Figure 59. One or More Loop Example in Seabird

D. CHAPTER SUMMARY

This chapter presented Seabird, a new tool that offers a set of graphical symbols which can be transpiled to MP code. Seabird includes the root, atomic, parallel, selection, loop, and coordination events. Notable exclusions from Seabird include the "SHARE ALL" to share events between roots, the "ENSURE" construct to support predicate calculus expressions, the print message "SAY" command, set iteration, and the ability to "MARK" specific traces. At this time, automatic generation of advanced logic is left to human coders and written within the MP-Firebird tool. Seabird implements a subset of SysML, as SysML v2 is an emerging standard, for example interrupting regions and other advanced SysML features are not yet supported. Future iterations of Seabird and Firebird could explore a user interface wizard to assist with these constructs.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

The use of graphical languages to automatically generate formal modeling semantics provides significant promise to increasing the usability and broadening the user base for emergent behavior technology. MP enables modelers to unlock both desired and undesired emergent behaviors, typically unforeseen by human intuition alone. Prior to MP's development, a rich set of functional flow diagram languages was developed, with each new representation building on the next, to provide greater expressiveness while increasing formalism. Similarly, a growing suite of tooling is building around this ecosystem of diagrams, with a still variable ability to formally verify and execute these diagrams. Today, few tools support formalism of all drawing palette symbols, and, furthermore, only LML and SDL tools have complete executability. However, while the common flowchart remains the most popular graphical representation in use today, new representations, such as SysML v2, offer promise. SysML v2 activity diagrams are graphically familiar to flowchart users but offer the ability to generate formal textual code semantics.

Seabird can be used by Navy and academic users alike to discover scenario side effects previously undiscoverable to the modeling community. This research demonstrated, using Seabird, the inception of automatic MP code generation from SysML activity diagram symbols. Seabird is a new open-source tool, derived from the technology behind diagrams.net, that offers graphical development of MP diagrams. Seabird learns from prior modeling languages and tools, presenting a horizontally focused activity layout, with symbols familiar to flowchart users. Modelers can use the root event, atomic event, alternate event selection, parallel unordered sets of actions, and both loop event types from MP. Additionally, with SysML input/output flow in Seabird, the MP coordinate statement can be automatically generated from graphical input.

B. FUTURE WORK

While this research proved the principle of automatic MP code generation, Seabird usability is under active development and is expected to continually improve over time. For example, MP's set iteration "replicate" capability is not yet implemented. Furthermore, additional enhancements could increase model quality and enrich Seabird's expressiveness.

1. Activity Diagram Decomposition

The Seabird transpiler supports the generation of composite events through a simple JSON payload. However, the Seabird user interface does not yet support a set of decomposed diagrams. Future research should develop interface elements and database persistence technologies to allow for action decomposition. Hierarchical decomposition of actions will allow for more complex models to be generated from Seabird.

2. Seabird Usability Study

There is an opportunity to study usability heuristics to reach the widest audience possible. Seabird was developed by a single author with no formal usability testing. A user study should be performed to discover if the graphical interface truly increases usability to new MP users and if the interface elements could be further improved. This study should be made available publicly to improve the broader suite of modeling tools.

3. Seabird Modeling Validation Assistant

Seabird currently offers a free drawing environment like the user interface paradigm of many UML tools such as Sparx Enterprise Architect® and Dassault NoMagic MagicDraw®. Seabird expands upon this free drawing concept with the addition of scope-complete generation, however there is an opportunity to assist modelers further. Future research should reference SAIC's validation tool to provide live assistance to users developing activity diagrams to encourage proper logic development (SAIC. n.d.). The use of intelligent development assistance is already common in the software engineering community and could provide similar benefits to the systems engineering community.

4. Natural Language MP

Currently Seabird will only generate formal semantics from graphical language elements. Definition of predicate calculus conditions can be derived directly from human textual expressions. The next generation of machine-learning NLP technology, such as Generative Pre-trained Transformer 3 (GPT3) or GitHub Code Pilot, could be combined with Seabird's graphical interface to enable automatic predicate calculus generation. The use of GPT3 and related technologies can at times generate incorrect code, thus a new set of verification technologies will need to be developed to check output from these tools. The integration of natural language with graphical semantics could be the next evolution of modeling capabilities.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- Aizier, Bruno, Stéphanie Lizy-Destrez, Charlotte Seidner, Chapurlat Vincent, Daniel Pru, and Jean-Luc Wippler. 2012. “xFFBD: Towards A Formal Yet Simple and Complete Functional Modeling Technique for System Designers.” In *22nd Annual International Council on Systems Engineering Symposium*. <https://hal-enac.archives-ouvertes.fr/hal-01022483>.
- Alford, Mack W. and James T. Lawson. 1979. *Software Requirements Engineering Methodology (Development)*. Report No. RADC-TR-79-I68. Rome, NY: Rome Air Development Center. <https://apps.dtic.mil/sti/pdfs/ADA073132.pdf>.
- American National Standards Institute, Inc (ANSI). 1971. *Flowchart Symbols and Their Usage in Information Processing*. Report No. ANSI X3.5-1970. <https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub24.pdf>.
- The American Society of Mechanical Engineers. 1947. *Operation and Flow Process Charts*. New York: ASME. <http://catalog.hathitrust.org/Record/005735891>.
- Auguston, Mikhail and Alberto Delgado. 1997. “Iterative Constructs in the Visual Data Flow Language.” In *IEEE Symposium on Visual Languages* 152–159. <https://doi.org/10.1109/VL.1997.626573>.
- Bedini, Francesco, Ralph Maschotta, Alexander Wichmann, and Armin Zimmermann. 2019. “Towards Automated fUML Model Verification with Petri Nets.” In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development* 298–306. <https://doi.org/10.5220/0007371402980306>.
- Birkmeier, Dominik Q., Sebastian Klöckner, and Sven Overhage. 2010. “An Empirical Comparison of the Usability of BPMN and UML Activity Diagrams for Business Users.” In *ECIS 2010 Proceedings* 51. <http://aisel.aisnet.org/ecis2010/51>.
- Bock, Conrad. 2003. “UML 2 Activity Model Support for Systems Engineering Functional Flow Diagrams.” *Systems Engineering* 6 (4) (October): 249–265. <http://doi.org/10.1002/sys.10053>.
- Boehm, Barry, Ricardo Valerdi, and Eric Honour. 2008. “The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems.” *Systems Engineering* 11 (3) (April): 221–234. <https://doi.org/10.1002/sys.20096>.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. 2005. *Unified Modeling Language User Guide*. 2nd ed. London: Pearson.

- Brand24. n.d. "Project: Flowchart, UML, SysML, and BPMN." Accessed December 15, 2021. <https://app.brand24.com/panel/compare/?sid=752612568#d1=2021-10-15&d2=2021-12-15&dr=4&va=1&cdt=days&pc=752540006,752613065,752914744>.
- Chen, Kuo-Tay and Ronald Lee. 1992. *Schematic Evaluation of Internal Accounting Control Systems*. Report No. RM-1992-08-1. Rotterdam, Netherlands: Erasmus University Rotterdam, https://www.researchgate.net/publication/228713375_Schematic_Evaluation_of_Internal_Accounting_Control_Systems/figures?lo=1.
- Chonoles, Michael Jesse. 2018. "Chapter 18 – Behavior: State Machine Diagrams." In *OCUP Certification Guide*, edited by Michael Jesse Chonoles, 313–342. Amsterdam: Morgan Kaufmann. <https://doi.org/10.1016/B978-0-12-809640-6.00019-2>.
- CIToolkit. n.d. "Pugh Matrix Template." Accessed January 7, 2023. <https://citoolkit.com/templates/pugh-matrix-template>.
- Cook, Stephen and Shaun Wilson. 2015. "Determining Return on Investment for MBSE, SE & SoSE." Paper presented at Australian Systems Engineering Workshop, Sydney, Australia. https://www.incosewiki.info/Model_Based_Systems_Engineering/Files/4/47/07_-_Cook_%26_Wilson_-_Pres.pdf.
- Dabloons. 2012. "How does one represent multiple threads in a flow chart." Stack Overflow. Last modified October 25, 2012. <https://stackoverflow.com/questions/13074620/how-does-one-represent-multiple-threads-in-a-flow-chart>.
- Damm, Werner and David Harel. 2001. "LSCs: Breathing Life into Message Sequence Charts." *Formal Methods in System Design* 19 (July) 45–80. <https://doi.org/10.1023/A:1011227529550>.
- de Weck, Olivier. 2012. "Design Structure Matrix." Lecture at Massachusetts Institute of Technology. https://ocw.mit.edu/courses/esd-36-system-project-management-fall-2012/resources/mitesd_36f12_lec04.
- Eldred, Ross, Jonathan Lussier, and Anthony Pollman. 2021. "Design and Testing of a Spherical Autonomous Underwater Vehicle for Shipwreck Interior Exploration." *Journal of Marine Science and Engineering* 9 (3) (February): 320. <https://doi.org/10.3390/jmse9030320>.
- Ensmenger, Nathan. 2016. "The Multiple Meaning of a Flowchart." *Information & Culture: A Journal of History* 51 (3): 321–351. <https://doi.org/10.1353/lac.2016.0013>.

- Federal Aviation Administration. 2006. “Functional Analysis (Satisfies EIA 731 FA 1.2 and iCMM PA 4).” Last modified January 14, 2009. https://web.archive.org/web/20090114031713/http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/operations/sysengsaf/seman/SEM3.1/Section%204.4.pdf.
- Federal Information Processing Standards Publication. 1993. “Integration Definition for Function Modeling (IDEF0): Draft Federal Information Processing Standards Publication 183.” Last modified February 27, 2009. <https://web.archive.org/web/20090227122140/http://www.itl.nist.gov/fipspubs/idef02.doc>.
- Friedenthal, Sanford, Alan Moore, and Rick Steiner. 2015. “Chapter 9 – Modeling Flow-Based Behavior with Activities.” In *A Practical Guide to SysML*. 3rd ed. Amsterdam: The MK/OMG Press. <http://dx.doi.org/10.1016/B978-0-12-800202-5.00009-6>.
- Friedenthal, Sanford. 2021. “Introduction to the SysML V2 Language Graphical Notation.” GitHub. Accessed December 28, 2021. <https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/Intro%20to%20the%20SysML%20v2%20Language-Graphical%20Notation.pdf>.
- Giammarco, Kristen and Len Troncale. 2018. “Modeling Isomorphic Systems Processes Using Monterey Phoenix.” *Systems* 6 (2) (May): 18. <http://dx.doi.org/10.3390/systems6020018>.
- Gilbreth, Frank B. and L. M. Gilbreth. 1921. *Process Charts*. New York: The American Society of Mechanical Engineers. https://web.archive.org/web/20150509222833id_/https://engineering.purdue.edu/IE/GilbrethLibrary/gilbrethproject/processcharts.pdf.
- The Gilbreth Network. n.d. “Biography.” Accessed December 21, 2021. <https://gilbrethnetwork.tripod.com/bio.html>.
- GitHub. 2021. “Introduction to the SysML v2 Language Graphical Notation.” SysML-v2-Release. Accessed October 8, 2021. <https://github.com/Systems-Modeling/SysML-v2-Release/blob/master/doc/Intro%20to%20the%20SysML%20v2%20Language-Graphical%20Notation.pdf>.
- Google Scholar. n.d. “About.” Accessed December 11, 2021. <https://scholar.google.com/intl/en/scholar/about.html>.
- Google Trends. n.d. “Compare: ‘Monterey phoenix’ ‘sysml’ ‘uml.’” Accessed October 8, 2021. <https://trends.google.com/trends/explore?geo=US&q=%22Monterey%20phoenix%22,%22sysml%22,%22uml%22>.

- Google Trends. n.d. “Interest over time: Systems Modeling Language, Unified Modeling Language, Business Process Model, and Notation Flowchart.” Accessed December 21, 2021. <https://trends.google.com/trends/explore?date=2011-01-01%202020-12-31&geo=US&q=%2Fm%2F0csg1c,%2Fm%2F07x3g,%2Fm%2F08kq3d,%2Fm%2F02ltk5>.
- Google Trends. n.d. “Programming Language: Unified Modeling Language.” Accessed December 5, 2021. <https://trends.google.com/trends/explore?date=all&geo=US&q=%2Fm%2F07x3g>.
- Hanna, Katie Terrell. 2021. “Business Process Management Initiative (BPMI).” TechTarget. Last modified October 2021. <https://searchcio.techtarget.com/definition/Business-Process-Management-Initiative-BPMI>.
- Harel, David. 1987. “Statecharts: A Visual Formalism for Complex System.” *Science of Computer Programming* 8 (3) (June) 231–274. http://www.inf.ed.ac.uk/teaching/courses/seoc/2005_2006/resources/statecharts.pdf.
- Haugen, Øystein. 2004. “Comparing UML 2.0 Interactions and MSC-2000.” In *SAM 2004: System Analysis and Modeling*, 65-79. https://doi.org/10.1007/978-3-540-31810-1_5.
- Hitchins, Derek. 1990. “Conceiving Systems.” PhD thesis, City University London. https://openaccess.city.ac.uk/id/eprint/7668/1/Conceiving_systems.pdf.
- IBM. n.d. “Watson Natural Language Understanding.” Accessed December 11, 2021. <https://www.ibm.com/cloud/watson-natural-language-understanding>.
- IBM Corp. 2006. “Differences Between UML 1.x and UML 2.0.” Accessed December 21, 2021. http://home.iscte-iul.pt/~hro/RUPSmallProjects/core.base_rup/guidances/supportingmaterials/differences_between_uml_1_x_and_uml_2_0_CA70F2E6.html.
- Innoslate Help Center. 2021. “Level Five: Triggers.” Last modified September 10, 2021. <https://help.innoslate.com/users-guide/simulators/discrete-event/triggers>.
- International Telecommunication Union. 1994. *Criteria for the Use and Applicability of Formal Description Techniques*. Report No. Z.120, ITU-T. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-Z.120-199303-S!!PDF-E&type=items.
- Johnson, Khari. 2020. “GitHub: Python and TypeScript gain popularity among programming languages.” Venture Beat. Last modified December 2, 2020. <https://venturebeat.com/2020/12/02/github-python-and-typescript-gain-popularity-among-programming-languages>.
- Kasser, Joseph E. 2018. *Systems Thinker’s Toolbox: Tools for Managing Complexity*. 1st ed. Boca Raton: CRC Press.

- Keuffel, Warren. 1990. "Extra time saves money." *Computer Language*, November 1990.
- Lifecycle Modeling Language. 2015. "Lifecycle Modeling Language (LML) Specification." Accessed December 21, 2021. https://lifecyclemodeling.org/wp-content/uploads/2021/01/LML_Specification_1_1.pdf.
- Long, James E. 2000. "Relationships between Common Graphical Representations Used in System Engineering." *Insight* 21 (1) (April): 8-11. <https://doi.org/10.1002/inst.12183>.
- Lübke, Daniel, Maike Ahrens, and Kurt Schneider. 2021. "Influence of Diagram Layout and Scrolling on Understandability of BPMN Processes: An Eye Tracking Experiment with BPMN Diagrams." *Information Technology and Management* 22 (April): 99–131. <https://doi.org/10.1007/s10799-021-00327-7>.
- Ludewig, Julia. 2005. "Practical Methods and Tools for Specification." *Embedded Systems* (January): 174-207. <https://doi.org/10.1007/BFb0016352>.
- Lynch, Allison. 2021. "What is BPMN - Definition, Elements and Purpose." Accessed January 1, 2021. Edraw. <https://www.edrawsoft.com/what-is-bpmn.html>.
- Madni, Azad M. and Shatad Purohit. 2019. "Economic Analysis of Model-Based Systems Engineering." *Systems* 7 (1) (February): 12. <https://doi.org/10.3390/systems7010012>.
- Mattsson, Michael. 1994. "A Comparative Study of Three New OO Methods." Thesis, University of Karlskrona/Ronneby. <https://www.diva-portal.org/smash/get/diva2:833703/FULLTEXT01.pdf>.
- MBSE Execution. 2020. "An Interruptible Activity Region Modeling and Simulation," YouTube video, September 8, 2020, 3:47. <https://youtu.be/OXBdT9KI4dc>.
- Mekki, Ahmed, Mohamed Ghazel, and Armand Toguyéni. 2011. "Patterns-Based Assistance for Temporal Requirement Specification." Paper presented at Software Engineering Research and Practice, Las Vegas, Nevada.
- Microsoft. n.d. "Create a Basic Flowchart in Visio." Accessed December 21, 2021. <https://support.microsoft.com/en-us/office/create-a-basic-flowchart-in-visio-e207d975-4a51-4bfa-a356-eeec314bd276>.
- No Magic. n.d. "SysML Activity Diagram." Accessed December 21, 2021. <https://docs.nomagic.com/display/SYSMLP190/SysML+Activity+Diagram>.
- NPS Wiki. n.d.a. "Iterating with MP (Repetitive Event Patterns)." Accessed December 21, 2021. <https://wiki.nps.edu/pages/viewpage.action?pageId=921665540>.

- NPS Wiki. n.d.b. “MP Event Basics and Vocabulary.” Accessed December 21, 2021.
<https://wiki.nps.edu/display/MP/4.+MP+Event+Basics+and+Vocabulary>.
- NPS Wiki. n.d.c. “Event Interactions.” Accessed May 6, 2023.
<https://wiki.nps.edu/display/MP/6.+Event+interactions>.
- NPS Wiki. n.d.d. “About.” Accessed August 6, 2023.
<https://wiki.nps.edu/display/MP/About>.
- OMG. 2007. *OMG Systems Modeling Language (OMG SysML™)*. Report No. formal/2007-09-01. <https://www.omg.org/spec/SysML/1.0/PDF>.
- OMG. 2010. *BPMN 2.0 by Example*. Report No. dtc/2010-06-02. <https://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf>.
- OMG. 2016. *Case Management Model and Notation (CMMN)*. Report No. formal/2016-12-01. <https://www.omg.org/spec/CMMN/1.1/PDF>.
- OMG. 2017. *Unified Modeling Language*. Report No. formal/2017-12-05.
<https://www.omg.org/spec/UML/2.5.1/About-UML>.
- OMG. 2021a. *Decision Model and Notation*. Report No. formal/2021-01-01.
<https://www.omg.org/spec/DMN/1.3/PDF>.
- OMG. 2021b. *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Report No. formal/2021-03-01. <https://www.omg.org/spec/FUML/1.5/PDF>.
- OMG. n.d. “About the Unified Modeling Language Specification Version 1.1.” Accessed October 8, 2021. <https://www.omg.org/spec/UML/1.1/About-UML>.
- Petri, Carl A., and Wolfgang Reisig. 2008. “Petri Net.” *Scholarpedia* 3 (4): 6477.
<https://doi.org/10.4249/scholarpedia.6477>.
- Predictive Analytics Today. n.d. “70 Top Open Source and Free BPM Tools: The Best of Business Process Management Software.” Accessed May 21, 2023.
<https://www.predictiveanalyticstoday.com/open-source-free-bpm-tools>.
- Quantum Leaps. 2015. *Application Note: A Crash Course in UML State Machines*. Oxford: Newnes. http://www.state-machine.com/doc/AN_Crash_Course_in_UML_State_Machines.pdf.
- Quatrani, Terry. n.d. “Introduction to UML 2.0.” Accessed December 21, 2021.
http://www.cs.unb.ca/~wdu/cs3043/00-T4_Matthews.pdf.
- SAIC. n.d. “Digital Engineering Validation Tool.” Accessed February 2, 2023.
<https://www.saic.com/digital-engineering-validation-tool>.

- Selic, Bran. 2005. "What's New in UML™ 2.0?" Accessed December 21, 2021.
https://www.researchgate.net/publication/228376554_What's_new_in_UML_20.
- Smithsonian. n.d. "Flowcharting Templates." Accessed May 21, 2023.
<https://americanhistory.si.edu/collections/object-groups/flowcharting-templates>.
- Sparx Systems. n.d. "UML 2 Tutorial – State Machine Diagram." Accessed on December 27, 2021. <https://sparxsystems.com/resources/tutorials/uml2/state-diagram.html>.
- Störrle, Harald. 2005. "Semantics and Verification of Data Flow in UML 2.0 Activities." *Electronic Notes in Theoretical Computer Science* 127 (4) (April): 35–52.
<https://doi.org/10.1016/j.entcs.2004.08.046>.
- Syque. n.d. "IDEF0 - Part 1 (understanding it)." Accessed December 26, 2021.
http://www.syque.com/quality_tools/tools/Tools19.htm.
- SysML Forum. 2012. "Generating BDD from swim lanes in an activity diagram." Accessed on December 22, 2021. <https://groups.google.com/g/sysmlforum/c/J52wUmvSm0g>.
- SysML Forum. n.d. "SysML FAQ: Who created SysML and how did it get its name?" Accessed on December 22, 2021. <https://sysmlforum.com/sysml-faq/who-created-sysml.html>.
- TRW. 1974. *Software Requirements Engineering Methodology Notebook*. Report No. 2294-6921-02G. <https://apps.dtic.mil/sti/pdfs/AD0923470.pdf>.
- U.S. Department of Commerce. 1976. *TRAIDEX Needs and Implementation Study*. Report No. AD-A024-861. <https://apps.dtic.mil/sti/pdfs/ADA024861.pdf>.
- U.S. Department of Defense. 2018. *Digital Engineering Strategy*. Washington, D.C.: Office of the Deputy Assistant Secretary of Defense for Systems Engineering. <https://man.fas.org/eprint/digeng-2018.pdf>.
- U.S. House of Representatives. 2016. *National Defense Authorization Act for Fiscal Year 2017*. Report 114–840. <https://www.congress.gov/114/crpt/hrpt840/CRPT-114hrpt840.pdf>.
- UML Diagrams. n.d. "Sequence Diagram Overview." Accessed December 21, 2021.
<https://www.uml-diagrams.org/sequence-diagrams/sequence-diagram-overview.png>.
- Van der Aalst, Wil. 1994. "Putting High-Level Petri Nets to Work in Industry." *Computers in Industry* 25 (January) 45-54. [https://doi.org/10.1016/0166-3615\(94\)90031-0](https://doi.org/10.1016/0166-3615(94)90031-0).

Vitech Corporation. n.d. “Function Flow Block Diagram (FFBD).” Accessed December 21, 2021. [https://www.vitechcorp.com/resources/core/onlinehelp/desktop/Views/Function_Flow_Block_Diagram_\(FFBD\).htm](https://www.vitechcorp.com/resources/core/onlinehelp/desktop/Views/Function_Flow_Block_Diagram_(FFBD).htm).

Vitech Corporation. 2015. *COREsim User Guide*. Blacksburg: Vitech Corporation.

West, Timothy D. and Mark Blackburn. 2017. “Is Digital Thread/Digital Twin Affordable? A Systemic Assessment of the Cost of DOD’s Latest Manhattan Project.” *Procedia Computer Science* 114 (October): 47–56. <https://doi.org/10.1016/j.procs.2017.09.003>.

Wikipedia. 2021. S.v. “List of BPMN 2.0 Engines.” Last modified November 19, 2021. https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines.

Wikipedia. 2023. S.v. “Functional flow block diagram.” Last modified June 1, 2023. https://en.wikipedia.org/wiki/Functional_flow_block_diagram.

INITIAL DISTRIBUTION LIST

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



DUDLEY KNOX LIBRARY

NAVAL POSTGRADUATE SCHOOL

WWW.NPS.EDU

WHERE SCIENCE MEETS THE ART OF WARFARE