Treadstone: A Process for Improving Modeling Prowess Using Validation Rules

Mr. Michael J. Vinarcik P.E., University of Detroit Mercy

Michael J. Vinarcik is a Chief Systems Engineer at SAIC and an adjunct professor at the University of Detroit Mercy. He has thirty years of automotive and defense engineering experience. He received a BS (Metallurgical Engineering) from the Ohio State University, an MBA from the University of Michigan, and an MS (Product Development) from the University of Detroit Mercy. Michael has presented at National Defense Industrial Association, International Council on Systems Engineering, and American Society for Engineering Education regional and international conferences. He is a regular speaker at the No Magic World Symposium. Michael has contributed chapters to Industrial Applications of X-ray Diffraction, Taguchi's Quality Engineering Handbook, and Case Studies in System of Systems, Enterprise Systems, and Complex Systems Engineering; he also contributed a case study to the Systems Engineering Body of Knowledge (SEBoK). He is a licensed Professional Engineer (Michigan) and holds INCOSE ESEP-Acq, OCSMP: Model Builder – Advanced, Booz Allen Hamilton Systems Engineering Expert Belt, ASQ Certified Quality Engineer, and ASQ Certified Reliability Engineer certifications. He is a Fellow of the Engineering Society of Detroit, the President and Founder of Sigma Theta Mu, the systems honor society, and the current Treasurer of INCOSE.

Treadstone:
A Process for Improving Modeling Prowess
Using Validation Rules

Abstract:

The creation of descriptive models using SysML is a skill-focused discipline; the outcomes of a modeling effort depend upon the abilities of the modelers contributing to it. Ongoing shortages of skilled modelers are inhibiting the transition of systems engineering to a model-based discipline.

This paper illustrates the use of validation rules to support instruction (both stand-alone modeling exercises and a larger, collaborative modeling project). Validation rules have proven to be effective in reducing modeler errors when added incrementally in parallel with concepts introduced in class. The rules simplify grading (since the instructor can focus on value-added content instead of semantic correctness). In addition, the rules conform to the *Seven Keys to Effective Feedback* proposed by Grant Wiggins:

1. Goal-Referenced (Error reduction/style conformance)
2. Tangible and Transparent (Rules clearly explain what is wrong)
3. Actionable (Error messages direct the modeler how to fix the issue)
4. User-Friendly (Private feedback that marks elements with to simplify repair)
5. Timely (On demand and rapid feedback eliminates errors before they accumulate)
6. Ongoing (Available throughout the course of any modeling project)
7. Consistent (All students receive the same feedback) [1].

The rules were continuously updated throughout the term in which they were introduced; students corrected new errors and improved their model quality as they executed their term projects. Extracts from six team projects will be presented and contrasted with selected past projects (subjected to the same validation rules) to demonstrate the efficacy of the approach. Several models published by notable SysML modelers will also be analyzed.

Systems Engineering in 2020: A Discipline in Flux

At the time of this paper's publication, systems engineering is undergoing a transformation from document-intensive systems engineering (DISE) to model-based systems engineering (MBSE). This shift is driven by the increasing complexity of modern systems; traditional methods that rely on requirements statements, pictures-as-diagrams, and collections of disconnected artifacts and documents are not scalable. They are prone to drifting out of synchronization, resulting in errors and omissions in the communication of design intent that manifest themselves as errors in testing or failures in usage.

A well-crafted system model is inherently consistent and complete; relevant information can be readily extracted to serve individual stakeholder's needs. Unfortunately, many practitioners do not create well-crafted models. Many superficially replicate DISE artifacts by using modeling tools as drawing tools (focusing on visual representations instead of model consistency). Others fail to scrutinize their work to detect errors and omissions. Even skilled modelers disagree on modeling methodology; each has his own preferred modeling patterns and stylistic idiosyncrasies. Published textbooks and style guides tend to focus on superficialities (colors, capitalization of words, and number of elements on a diagram) instead of emphasizing the need to ensure that model elements, properties, and relationships are complete and consistent.

This paper will focus on how system modeling is a craft (with resulting pedagogical implications), the creation of descriptive and/or executable models in SysML (the system modeling language), and how the use of automated validation rules can help teach system modeling and improve model quality.

System Modeling as a Craft

Model-Based Systems Engineering (MBSE) is a skill-based discipline that shares many similarities with software development. Crafting descriptive system models in the System Modeling Language (SysML) is like writing code; models are an expression of intent. Models are also path dependent, in that the experience of the development team shapes the outcome. Although good modeling practices and styles can be recognized and applied, alternate expressions of the same solution (and the existence of multiple solutions in a design set) imply that, in practice, there is not one, perfect, absolutely correct and optimal representation of a system. Even if such a construct existed in theory (a matter left for others to debate), the effort to seek it and develop a model in conformance to it may exceed the benefits of a satisficing model available rapidly at a lower development cost.

Hillary Sillitto suggests that "Architecting a system is the activity of creating a system architecture with the aim that the system to be built will do the job it was meant to do – in other words "will be fit for purpose"…Architecting defines what to design, while design defines what to build." [2]

There is significant ongoing work in applying patterns and machine learning to system models (because the rigor of a purpose-built language enables visibility and analysis techniques impossible with natural language processing). There is promise in these efforts and they will likely enable measurable improvements in model quality. However, there is a need to execute effective systems architecture *now*, and that means that for the foreseeable future the capabilities of the humans-in-the-loop will dominate the outcomes of each modeling effort.

Merriam-Webster defines *craft* as "an occupation or trade requiring manual dexterity or artistic skill." [3] Ryan Noguchi of Aerospace Corporation states: "System modeling is not a conceptually simple or straightforward task. It is like computer programming in many respects, requires some similar skill sets, and involves similar design tradeoffs…The conceptual model

and the organization of models into modules can significantly impact their usability, consistency, and maintainability. It is important to have experienced architects familiar with both the problem space and the capabilities of the tools to lead that effort. Furthermore, building models using these tools is not always straightforward, and like software programming, is a skill that not everyone can learn equally well." [4] In essence, it is a craft.

In *Apprenticeship Patterns*, Hoover and Oshineye describe the values of craftsmanship as including (emphasis added by the author):

- An attachment to… a "growth mindset." This entails a belief that you can be better and everything can be improved if you're prepared to work at it…
- ***A need to always be adapting and changing based on the feedback you get from the world around you***…
- ***A desire to be pragmatic rather than dogmatic. This involves a willingness to trade off theoretical purity or future perfection in favor of getting things done today.***
- A belief that it is better to share what we know than to create scarcity by hoarding it…
- A willingness to experiment and be proven wrong…
- A dedication to what psychologists call an internal locus of control. [https://en.wikipedia.org/wiki/Locus_of_control] This involves taking control of and responsibility for our destinies rather than just waiting for someone else to give us the answers.
- A focus on individuals rather than groups...
- A commitment to inclusiveness…
- We are skill-centric rather than process-centric. For us, it is more important to be highly skilled than to be using the "right" process…***This idea suggests that no process or tool is going to make everyone equally successful. Even though we can all improve, there will always be discrepancies in our skill levels.***
- A strong preference for what Etienne Wenger calls "situated learning." [http://wiki.c2.com/?LegitimatePeripheralParticipation***..Its essence is that the best way to learn is to be in the same room with people who are trying to achieve some goal using the skills you wish to learn.***" [5]

Teaching a Craft: Learning to Do by Doing

Figure 1: Lintel from A Closed School (Author's Photo)

The maxim in Figure 1 was cut into the lintel above the main entrance to a large, old, closed school. It is an educational idea that comes into and out of favor in academia but is critically important to passing on craft skills (consider the difficulties inherent in teaching blacksmithing solely by drawing diagrams and lecturing instead of giving students access to a forge, anvil, tools, and steel…and consider the implications if the instructor is not a skilled, practicing blacksmith).

Acknowledging modeling's nature as a craft has shaped the author's pedagogical approach to teaching system modeling using SysML and MagicDraw from Dassault Systèmes. The introductory system modeling class currently consists of two phases. In the first, students are required to construct small models and demonstrate their ability to create simple diagrams using MagicDraw. They also learn how to edit and submit their work using TeamWork Cloud (the Dassault Systèmes collaboration server for MagicDraw). This ensures they can perform the rudiments of system modeling and have overcome any technical challenges in connecting to the university's server. In the second phase, teams of students collaboratively model representative systems. This lets them experience the subtleties of working together on a large system model, understand and experience best practices for collaboration, and cultivate a visceral understanding of how modeling tools work and how to make a model serve a systems engineering effort. Systems-of-interest have included notional nuclear submarines, next generation Mars orbiters, and simulated Mars rovers.

During the second phase, normal classroom meetings are suspended, and the instructor conducts weekly meetings with each team to answer questions, provide recommendations, and generally shepherd each team towards maturing its model. In the past, a significant amount of the instructor's time was spent on resolving semantic and structural issues with each model in addition to suggesting methods and pointing out opportunities for improvement. Resolving these issues is critical to the creation of well-formed models that can support advanced querying and tactical analysis…which should be the desired outcome of a modeling effort. Crafting a beautiful, useless model (or set of diagrams) does a disservice to all stakeholders.

The Burden of Inspection-Based Feedback

As Noguchi states, "Many of the DoDAF architecture description models that have been built to date have been found to have many syntactical errors that would have been caught had they used the built-in model validation capabilities of their tools, but the problem would also have been apparent upon visual inspection by an experienced modeler… When a program cannot afford to populate their [sic] modeling team completely with experienced modelers, it is critical that model reviews be performed frequently by experienced modelers, particularly to check for semantic mistakes—those that won't be caught by the modeling tools' validation checks—by providing the review team with the actual electronic model files for them to review in detail using the modeling tool. Model reviews performed in a briefing format or through static captures of the model (typically via PDF files or HTML files) are much less effective at ferreting out errors." [4]

Unfortunately, this review process is not scalable. The author found that the student models tend to grow rapidly (10^5 elements are not uncommon) and it rapidly becomes impossible to conduct the equivalent of five or more detailed model reviews on a weekly basis. Although students were expected to use MagicDraw's built-in validation rules, these checks were not sufficient to enforce the author's modeling methodology and style. In addition, they do not completely test for common modeling patterns and practices.

Automated Reviews via Custom Validation Suites

In 2019, the author attended *Approaches to Marking and Validating Sensitive MBSE Models*, a presentation at the 2019 MBSE Cyber Experience Symposium [6]. This presentation, by Veejay Gorospe of the Johns Hopkins University's Applied Physics Laboratory, showcased the usage of customized validation rules to assess security classification issues. Gorospe created validation rules using the structured expression language built into MagicDraw.

The author is a proponent of using these structured expressions to create purpose-built tables, matrices, and dynamic legends to extract value from models. The language uses internal operations (such as *union, intersect, isEmpty,* etc.) and *metachain navigation* (MagicDraw's attribute, element, and relationship navigation syntax) to extract information from the model. The structured expression language layers a relatively simple user interface on patterns that MagicDraw converts into internally executed code.

To craft a validation rule, all that is necessary is to identify the type of element to which it is to be applied (for example, an *object flow*) and to develop a structured expression that returns a Boolean value. Elements for which the expression is *true* pass validation and those for which it is *false* fail validation. Severity level and documentation (which help the modeler correct the error) should also be created. MagicDraw's mechanism for executing rules is that they are *constraints* with the <<validationRule>> stereotype applied. They must also be resident in a package with the <<validationSuite>> stereotype applied. This exposes them to the validation engine.

The author began constructing his own set of validation rules in the fall of 2019. These were intended to support the transition of the introductory SysML course to asynchronous online lectures supported with live, weekly lab sessions. These rules were designed to conduct stylistic, methodological, and semantic error-checking but were built from the perspective of an experienced modeler. As the author began to grade student models it rapidly became apparent that novice modelers could build compliant models that still did not conform to the desired style. These workarounds clearly identified gaps in the rule set and led to the development of more rules; the implicit and unspoken assumptions made from experience were instantiated in the expanded ruleset.

This meant that models that passed validation were graded in accordance with the rules as provided but that the next iteration of the model was expected to conform to the additional rules. Although the author tested each rule to eliminate false positives and false negatives,

idiosyncrasies in individual approaches highlighted additional adjustments needed to ensure that the rules functioned as intended.

The rule set began as a solely academic exercise at the University of Detroit Mercy; however, their utility led the author to integrate them into the models he created for his primary employer (SAIC). The rules were kept in two separate profiles (one at the university and one resident on the SAIC server). The author manually kept them in synch until his employer indicated a willingness to release them publicly as a service to the modeling community. The university profile was replaced with a pre-release version of the SAIC profile; this allowed more rapid development as additional errors and corner cases were revealed in both environments.

Testing the Rules: The Mars Society Rover Project

The subject of the Fall 2019 class modeling project was The Mars Society's 2020 University Rover Challenge [7]. This competition requires collegiate teams to build functioning rovers for competing at the Mars Desert Research Station in Utah. The rules are publicly published and executing a project based on them ensured that the student models do not contain proprietary information that would preclude their release.

The Challenge contained seventy-six requirements that were divided among the teams to copy/paste from the source PDF file into Microsoft Excel. They were then imported by the author to serve as a shared resource. Each team's model then used a library of common elements, the common requirements set, and the SAIC DE Profile.

Each team was instructed to singularize the requirements and establish <<satisfy>> and <<verify>> relationships between the requirements, architectural elements, and test cases. They refactored the requirements using the extended SysML requirement types (functional, performance, interface, design constraint, et al.) to leverage the built-in validation rules. They also established <<trace>> relationships between their requirements and the originals in the used project.

Student feedback was that this was a tedious process; that was the intent of this portion of the assignment. By translating the text-based requirements into model elements (and appropriate test cases), the students gained a visceral understanding of how poorly formed many textual requirements are. See Table 1 for a summary of the final requirements count for each team.

	Requirements Count		
Team	**Initial**	**Final**	**% Increase**
Curiosity	76	205	170%
JARS	76	216	184%
Strike Force Alpha	76	142	87%
Team Bolt	76	172	126%
Team Chimps	76	284	274%
Team Voodoo	76	110	45%

Table 1: Requirements Growth

Once the students had completed their analysis of the provided requirements, they were required to complete a logical architecture for their rover. These included activity diagrams, state machines, and internal block diagrams (IBDs). The IBDs illustrate the connections between structural elements of the model and what flows of energy, material, or information are transferred between each element. The modeling methodology taught in class emphasizes the harmonization of system behavior and structure; several rules check for consistency (for example, that an output from one system's behavior that is consumed as input of another element's function is mapped to a connection between them).

The models were graded four times during the term: Initial, Checkpoint, and Final submissions (as well as end-of-term). Project element counts and final error totals are displayed in Table 2. Note that three teams had zero errors and zero info (indications of immaturity, typically related to behavioral/structural disconnects). See Figure 2 and Figure 3 for plots of model elements vs. team and milestone.

	Model Elements				Validation		
Team	**Initial**	**Checkpoint**	**Final**	**End of Term**	**Errors**	**Info**	**Pages**
Curiosity	8587	22550	17588	21483	0	0	198
JARS	9548	16291	18357	21905	1	214	237
Strike Force Alpha	7070	8963	13351	16204	0	0	193
Team Bolt	8262	19440	15206	17773	0	39	199
Team Chimps	10015	22050	18732	21768	0	0	242
Team Voodoo	5664	8583	9846	11836	0	76	175

Table 2: Model Element Summary

The Pages column in Table 2 is the page count of a Requirements Report generated from MagicDraw. This standard report includes the requirements as well as diagrams and other model content. The author generated these to show the students how much material they had developed and integrated over the course of half of a semester. These reports also gave them an appreciation of the difficulties in generating artifacts of that size and complexity without using a model as an authoritative source of truth.

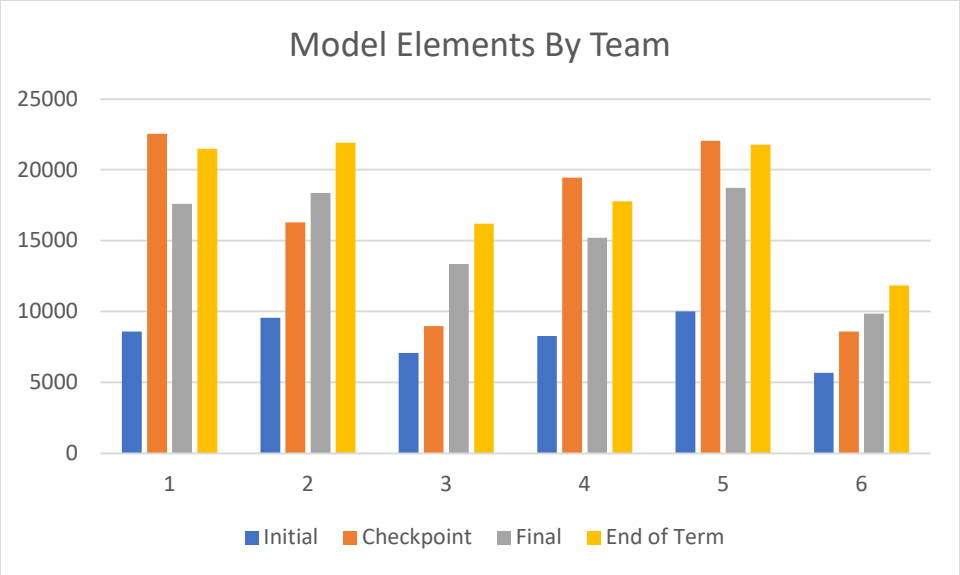

Figure 2: Model Elements by Team

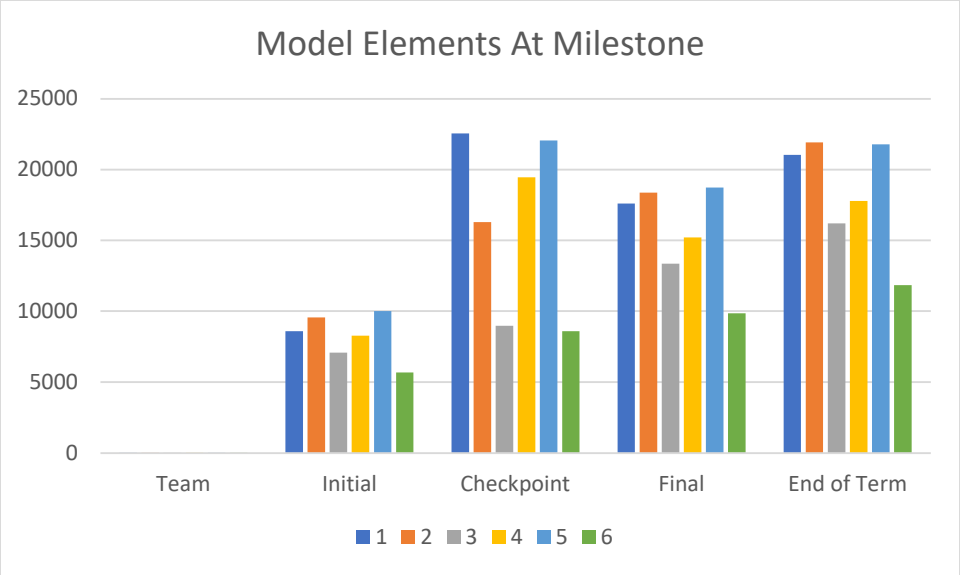

Figure 3: Model Elements by Milestone

Error Rates of Unassisted Modeling

The author applied the validation rules to models created by Lenny Delligatti [8], Sandy Friedenthal and Christopher Oster [9], and Robert Karban [10], and two other student models previously published (the NeMO [11] and PRZ-1 [12]). These models, all publicly available, represent a cross-section of modeling styles and approaches. The intent of this analysis was to confirm that the rules were broadly applicable and to measure the error rates of student and experienced modelers. Delligatti, Friedenthal, and Karban are well-regarded professionals; their

error rate is a good indication of how many errors skilled modelers introduce when modeling without automated validation (the average practitioner is probably much higher).

Row Labels	DellSat	Friedenthal	TMT	NeMO	PRZ-1	Grand Total
ACTIVITY	9	129	573	197	97	1005
ACTOR	6	6	5	19	24	60
CONBLOCK				12	4	16
OPERATION	45	18	390	326	119	898
SIGNAL	27	38	285	244	40	634
STATE	20	37	226	235	11	529
USE CASE	14	18	16	182	12	242
Grand Total	**121**	**246**	**1495**	**1215**	**307**	**3384**

Table 3: Undocumented Elements

Table 3: Undocumented Elements summarizes the documentation errors present in the models. This is a stylistic requirement (and these errors will not be included in the computation of error rate). However, they are shown here because in the author's experience properly documenting these element types prevents confusion downstream when a model has matured, and it is being actively used to make tactical decisions.

Row Labels	DellSat	Friedenthal	TMT	NeMO	PRZ-1	Grand Total
ACTIVITYNAME				2		2
ACTORNAME					3	3
BLOCKNAME			1			1
PACKAGENAME					1	1
REGIONNAME		4	46	10		60
STATENAME	1			22		23
Grand Total	**1**	**4**	**47**	**34**	**4**	**90**

Table 4: Unnamed Elements

Unnamed elements of the types listed in Table 4 tend to indicate modeling housekeeping errors; if an element is created and deleted from a diagram but not from the model it can be difficult to detect and remove. The rule to name region names in states is good modeling practice and is included in this set.

	DellSat	Friedenthal	TMT	NeMO	PRZ-1	Grand Total
ACCEPTEVENT TIMEEVENTTRIGGER			1			1
ACTIVITYEDGEGUARD		2	94	6	126	228
ACTIVITYPARAMETERFLOW	9	93	33	15	139	289
BUFFERFLOW			1		3	4
CALLOPERATIONOPERATION				4	2	6
CONSTRAINTPARAM		17				17
CONSTRAINTSPECIFICATION	2	40	338	28		408
CONTROLNODEINCOMING			4		2	6
CONTROLNODEOUTGOING		4	20	3	11	38
EXTENDEXTPOINT					1	1
EXTENSIONPOINTUSE					6	6
INPINSCONN		17	39	49	57	162
MESSAGESIGNATURE	12	10	1	9		32
PARATYPE			233	75	57	365
PARTTYPE			2		5	7
STATEREACHABILITY		2	1	32	2	37
TRANSITIONCHOICE		2				2
TRANSITIONTRIGGER	2	11	23	98	9	143
VALUETYPE		67	15	4	4	90
Grand Total	**25**	**265**	**805**	**323**	**424**	**1842**

Table 5: Omissions

The error totals in Table 5 are related to omissions in model elements; these represent unconnected elements, missing flows, undefined decision choices, and other content that should be included in a well-formed model. Some of these errors result from placing information in the wrong location in the model (for example, placing conditional information in the *Name* field instead of the *Guard* field); others may result from deleting some model elements improperly (a housekeeping issue).

	DellSat	Friedenthal	TMT	NeMO	PRZ-1	Grand Total
CALLBEHAVIORSELF			1			1
PARTLOOP			3		12	15
Grand Total			4		12	16

Table 6: Loops

The loops in Table 6 represent infinite loops of behavior or structure (behaviors that call themselves and part properties typed by elements that lead to recursive structures).

	DellSat	Friedenthal	TMT	NeMO	PRZ-1	Grand Total
Unnamed / Omissions / Loops	26	269	856	357	440	1948
Elements	3542	13187	275256	35293	44265	371543
Errors Per 1,000 Elements	7.3	20.4	3.1	10.1	9.9	5.9

Table 7: Error Summary

Table 7 lists the error totals for each model (not including documentation errors) and computes the number of errors per thousand model elements. The average error rate compares favorably with the industry average for software lines of code (15-50 errors per delivered lines of code) [13]. The models created by experienced modelers (DellSat/TMT) have a lower error rate than the student models. The Friedenthal/Oster model was built as a companion to a book and may not have been built with the same level of rigor as a model-for-use. Karban's model (TMT) has a larger fraction of parametric diagrams and elements (which have fewer associated rules); this may contribute to the proportionally lower number of defects. The two student models made without the assistance of validation rules have similar error rates.

Note that five of the six models constructed this term, supported by the validation rules, had zero errors (and the sixth had one error, a defect rate of 0.05 errors per thousand elements). This is a marked reduction in defects when compared with any of the other models, particularly when one considers these teams of five or six students learned SysML, MagicDraw, and the modeling methodology in the same term that they constructed the models. Table 8 illustrates the relative sizes of the models analyzed in this paper.

Team	Elements	Relative Size
DellSat	3542	1.3%
Friedenthal	13187	4.8%
TMT	275256	100.0%
NeMO	35293	12.8%
PRZ-1	44265	16.1%
Curiosity	21483	7.8%
JARS	21905	8.0%
Strike Force Alpha	16204	5.9%
Team Bolt	17773	6.5%
Team Chimps	21768	7.9%
Team Voodoo	11836	4.3%

Table 8: Relative Model Sizes

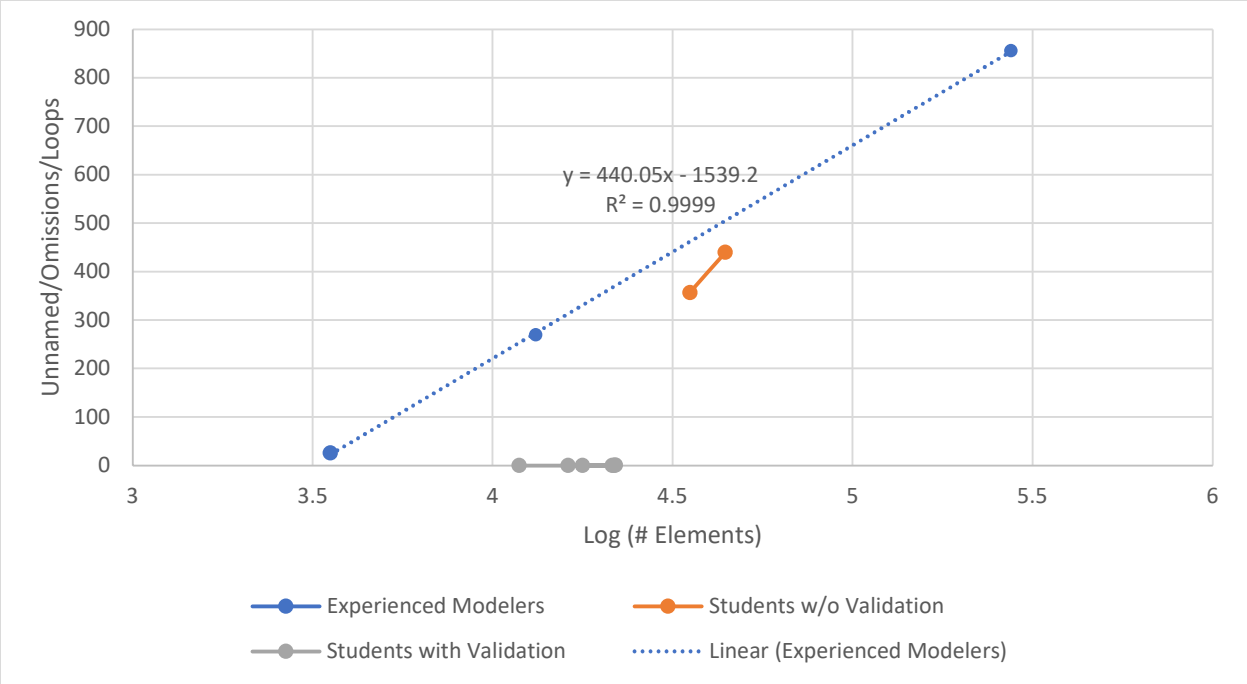

Figure 4: Error Rate

Figure 4 plots the number of errors in each model versus the $\log_{10}$ of the number of elements in each model. It suggests that the number of errors introduced without validation may be estimated as a function of the number of elements; it also suggests that the models created with validation are part of a different population (illustrating the efficacy of the rules in driving the desired outcomes).

Implementation in a Modeling Course

The use of the validation rules as a pedagogical aid demonstrated significant utility. The decrease in non-stylistic errors shows the value in detecting semantic and completeness issues; the lack of stylistic errors shows that the rules are effective in shaping modelers as they develop skill with the modeling tool, language, and methodology.

The validation rules are also aligned with Wiggin's *Rules for Effective Feedback*:

1. Goal-Referenced (Error reduction/style conformance)
2. Tangible and Transparent (Rules clearly explain what is wrong)
3. Actionable (Error messages direct the modeler how to fix the issue)
4. User-Friendly (Private feedback that marks elements with errors to simplify repair)
5. Timely (On demand and rapid feedback eliminates errors before they accumulate)
6. Ongoing (Available throughout the course of any modeling project)
7. Consistent (All students receive the same feedback) [1].

In essence, the rules allow students to benefit from the expertise encoded into them by receiving on-demand, detailed modeling feedback. During the term, students responded well to the use of the rules; many of them were able to resolve their errors with minimal assistance and the focus of the lab sessions tended to be on more advanced topics. In addition, the time to grade and assess the teams' submissions was greatly reduced. Students learned that creating elements triggered a cascade of errors until they were documented, traced and/or related, and used properly. The presence of defects in the model highlighted "hot spots" that required attention and other sections of the models could receive more cursory inspections. Generic tables and dependency matrices were also used to conduct quality checks that supplemented detailed review of specific diagrams and model elements.

Educators wishing to implement this approach may download the rules from SAIC [14]; they have been made publicly available as a service to the modeling community. As of May 2020, the rules are also accompanied by an example model, videos explaining some of the customizations included, and a model-based style guide. This guide provides illustrative examples and rationales for each rule; the example model, based upon the *Ranger* lunar probes of the 1960s, is a complete representation of a system in SysML: requirements, logical and physical architectures, behavior diagrams, and parametrics. The rules are being ported to the IBM Rational Rhapsody tool; these will also be released to the modeling community.

Each modeling tool allows the user to tailor which rules are applied to a given model; educators are encouraged to select the rules from this set that are applicable and to create their own supplemental rules as needed.

Conclusions and Future Work

The construction and deployment of automated validation rules had a positive impact on the quality of student models. It also reduced the time needed to review and grade models and allowed more time to be spent on addressing more advanced modeling issues. Analysis of models created without automatic validation illustrated that even experienced modelers introduce errors into models at a rate comparable to authoring software.

It is the author's hope that widespread adoption of this ruleset will improve the quality of SysML models created by students and professional practitioners. The model-based style guide and *Ranger* model can serve as exemplars to support students as they undertake modeling projects. This will help accelerate the transformation to a model-based discipline by resolving the current talent shortage.

The validation ruleset will continue to be expanded and tested in both academic and professional contexts (as of the v1.5 May 2020 release there are 153 rules). The author intends to re-analyze these models with the expanded ruleset and explore how rules may be used to assist in collaboration between organizations (for example, making a set of rules contractually binding so that any model exchanged between organizations must pass an agreed-upon set of rules).

Acknowledgements

The author would like to thank the Fall 2019 students in MENG 5925 (Modeling of Complex Systems via SysML Programming) at the University of Detroit Mercy for their enthusiasm and perseverance during the development of the initial set of validation rules. He would also like to acknowledge Heidi Jugovic and Pedro Lepe for their contributions and reviews, Richard Parise and Minh Nguyen for their work on the *Ranger* model that identified many corner cases, Heidi Jugovic for her leadership in developing the model-based style guide, Brain Haan for suggesting the $\log_{10}$ defect plot and testing the rules' implications for executable models, and Douglas Orellana for his willingness to publicly release this work to the modeling community.

This ruleset (which includes a profile and additional useful customizations) is available from SAIC [14] and the student models from this term are published [15]. The author invites comments about the rules and other feedback from the system modeling community.

References

[1] G. Wiggins, "Seven Keys to Effective Feedback," *Educational Leadership*, pp. 10-16, 2012.

[2] H. Sillitto, *Architecting Systems: Concepts, Principles and Practice*, London: College Publications, 2014.

[3] Merriam-Webster, [Online]. Available: https://www.merriam-webster.com/dictionary/craft. [Accessed 2 February 2020].

[4] R. A. Noguchi, "Lessons Learned and Recommended Best Practices from Model-Based Systems Engineering (MBSE) Pilot Projects," Aerospace Corporation, 2016.

[5] D. H. Hoover and A. Oshineye, *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*, Boston: O'Reilly MEdia, 2009.

[6] V. Gorospe, "Approaches to Marking and Validating Sensitive MBSE Models," in *MBSE Cyber Experience Symposium*, Allen, Texas, 2019.

[7] "University Rover Challenge," The Mars Society, [Online]. Available: http://urc.marssociety.org.

[8] L. Delligatti, "DellSat-77 Model (from *SysML Distilled*)".

[9] S. Friedenthal and C. Oster, "Spacecraft Model (from *Architecting Spacecraft with SysML: A Model-Based Systems Engineering Approach*)".

[10] R. Karban, "Thirty Meter Telescope Model," Jet Propulsion Laboratory / Thirty Meter Telescope Corporation.

[11] M. J. Vinarcik, "The NeMO Orbiter: A Demonstration Hypermodel," in *Ground Vehicle Systems Engineering and Technology Symposium*, Novi, 2018.

[12] M. J. Vinarcik, "A Pragmatic Approach to Teaching Model Based Systems Engineering: The PRZ-1," in *ASEE Annual Conference & Exposition*, Columbus, 2017.

[13] S. McConnell, *Code Complete (Developer Best Practices)*, Redmond: Microsoft Press, 2009.

[14] SAIC, "Digital Engineering Validation Tool," [Online]. Available: https://www.saic.com/digital-engineering-validation-tool.

[15] Systems Architecture Guild, "Hypermodeling," [Online]. Available: http://hypermodeling.systems.

Appendix 1: Validation Rules Used (circa December 2019)

Name	Severity	Error Message
ACCEPTEVENTMATCH	error	The signal triggering an accept event action must match the signal typing its output pin.
ACCEPTEVENTOUTPUT	error	Accept Events must own an output pin. If you are modeling a signal that triggers a state transition, associate the object flow with an item flow and realize the transition.
ACCEPTEVENTPORTMATCH	error	The assigned and inferred ports (via item flow realization) must match.
ACCEPTEVENTTIMEEVENTTRIGGER	error	Accept events triggered by time events must have WHEN defined.
ACCEPTOUTGOING	error	If an Accept Event outgoing object flow is realized by an item flow or flow set, the signal that triggers the accept event must be conveyed by the item flows or flow set.
ACTIVITYACTIONSTM	error	All activities owned by state machines must have at least one action node.
ACTIVITYDOCUMENTATION	error	All activities must have documentation.
ACTIVITYEDGEGUARD	error	All control and object flows exiting a decision node must have guards defined.
ACTIVITYEDGEMISMATCH	error	Flows into and out of a control node (join, fork, merge, or decision) must be of the same type (object or control).
ACTIVITYFINAL	error	Activities that own diagrams must own one final node and it must have one incoming control flow.
ACTIVITYINITIAL	error	Activities that own diagrams must own one initial node and it must have one outgoing control flow.
ACTIVITYNAME	error	Activities must be named.
ACTIVITYOWNS	error	Activities must own at least one diagram or operation. If it will not be further decomposed, set its "Leaf" attribute to true.
ACTIVITYPARAMETERFLOW	error	All activity parameter nodes must have incoming or outgoing object flows.
ACTIVITYPARAMETERSTM	error	State Machine Entry, Do, Exit, and Transition activities may not have parameters.

Name	Severity	Error Message
ACTORASSOCIATION	error	Actors may not be associated with other actors.
ACTORASSOCIATIONS	error	All Actors and other use case elements must be associated with at least one use case.
ACTORDOCUMENTATION	error	All Actors must have documentation.
ACTORNAME	error	All Actors must have names.
ACTORUSECASE	error	All use case elements must be associated with at least one use case or be specialized by other actors.
ACTPARTYPE	error	All activity parameter nodes must be typed by signals.
ACTREALIZATION	error	All Actors and other use case elements must be realized by at least one part property in the structure tree of a system context block. Environmental effects may be realized by value properties in the structure tree of a system context block.
ALLOCATIONPROHIBIT	error	Allocations are prohibited; use realization (between levels of abstraction) or satisfy (between requirements and other model elements).
BLOCKNAME	error	Blocks must be named.
BUFFERFLOW	error	Buffers and data stores must have at least one object flow (incoming or outgoing).
CALLBEHAVIORBEHAVIOR	error	Call behavior actions must have the called behavior specified.
CALLBEHAVIORSELF	error	Call behavior actions may not call the activity that owns them.
CALLOPERATIONOPERATION	error	Call operation actions must have the called operation specified.
CHANGEEVENTEXPRESSION	error	All transitions triggered by change events must have CHANGE EXPRESSION defined.
CONBLOCKDOCUMENTATION	error	All blocks that type part properties of the system context must have documentation.
CONNECTIONPOINTCONNECTED	error	Connection points must have one transition (outgoing or incoming).
CONNECTOREND	error	Connector ends must be proxy ports.

Name	Severity	Error Message
CONSTRAINTPARAM	error	Constraint blocks must own one or more constraint parameters.
CONSTRAINTSPECIFICATION	error	Constraint specifications may not be empty.
CONTEXTPARTS	error	System context blocks must own at least one part property.
CONTEXTREALIZATION	error	All part properties owned by system context blocks must realize one or more use case elements.
CONTEXTTYPE	error	Part properties may not be typed by system context blocks; they should typically be the top-level block that owns the system context IBD.
CONTROLNODEINCOMING	error	Joins and merges must have at least two incoming flows.
CONTROLNODEOUTGOING	error	Forks and decisions must have at least two outgoing flows.
CONVEYTYPE	error	Item flows may only convey signals.
DATASTORETYPE	error	Data stores must be typed by signals.
DECISIONNODENAME	error	Decision nodes must have a name (this is used to specify the decision).
EXTENDEXTPOINT	error	Extend relationships must be assigned to at least one extension point.
EXTENSIONPOINTUSE	error	Extension points must be associated with at least one Extend relationship.
EXTERNALPARTTYPE	error	Part properties typed by external blocks must be owned by system context or external blocks.
FLOWCONNECTOR	error	This flow is not realized by any connectors.
FLOWDIRECTION	error	All flow properties must be out or inout; this ensures consistent conjugation (all 1-way in flows are conjugated).
FLOWFINALINCOMING	error	All flow final nodes must have one incoming flow.
FLOWSETENDS	error	If a flow set has individual flows assigned, the individual flows must connect the source and target of the flow set.
FLOWSETSOURCE	error	Ports that are the source of a flow set must have flow properties compatible with the conveyed signals of the flow set.

Name	Severity	Error Message
FLOWSETTARGET	error	Ports that are the target of a flow set must have flow properties compatible with the conveyed signals of the flow set.
FLOWTYPE	error	All flow properties must be typed by signals.
IBDOWNER	error	IBDs must be owned by a block.
IBNOTSPECBLOCK	error	Interface blocks may not specialize non-interface blocks.
IMPITEMFLOWCOMPAT	error	Flow properties of proxy ports connected by flow sets must be compatible.
INPINCONN	error	Input pins must have an incoming object flow (target pins are exempt from this rule).
INTBLOCKFLOW	error	Interface blocks must own at least one flow property or port.
INTERFACENEEDED	info	The owners of the ends of this object flow are different and it is not realized by an item flow.
ITEMFLOWCONVEYED	error	All item flows must convey one or more signals or be part of a flow set.
LIFELINETYPE	error	All lifelines must be typed by blocks.
LOGICALARCH	error	Part properties that are owned by a block with a logical stereotype must be typed by a block with a logical stereotype.
LOGICALCONNFLOWS	info	All connectors that connect ports in the logical architecture must have at least one flow.
LOGICALPHYSICAL	error	Blocks cannot have both logical and physical stereotypes applied.
LOGICALPORT	error	All proxy ports owned by blocks with the <<logical>> stereotype applied must be typed by interface blocks with the <<logical>> stereotype applied.
LOGTERMPARTS	error	Logical blocks with ATOMIC = TRUE may not own part properties.
MESSAGEFLOWNEEDED	info	This message signature is a signal and is not realized by any item flows or flow sets.
MESSAGEFLOWS	error	If a message is associated with item flows or flow sets, they must convey its signature signal.

Name	Severity	Error Message
MESSAGESIGNATURE	error	All messages on sequence diagrams must have signatures assigned (signal or operation).
NOATTACHMENT	error	Embedding files in the model is not allowed. Use a hyperlink to an authoritative source instead (use an artifact if necessary, to represent the embedded file).
OBJECTFLOWCOMPAT	error	If an object flow is realized by an item flow or flow set, those flows must convey the signal typing its source.
OBJECTFLOWENDS	error	Object flows must have input/output pins as their source/target (no direct connection with send or accept events).
OBJFLOWSOURCE	error	Object flows must have pins as their source (not call operations or call behaviors).
OPAQUEACTIONBODY	error	Opaque actions must have a BODY specified.
OPDOCUMENTATION	error	All operations must have documentation.
OPERATIONNAME	error	Operations must be named.
OPOWNER	error	Operations must be owned by activities or blocks with context, logical, or physical stereotypes applied.
OPUSAGE	info	This operation is not used (called on an Activity or Sequence) in the model
OUTPINCONN	error	Output pins must have an outgoing object flow
PACKAGENAME	error	Packages must be named.
PARATYPE	error	All parameters owned by operations must be typed.
PARTIB	error	Part properties may not be typed by Interface blocks.
PARTLOOP	error	There is a part property loop associated with this block (a block in the structure owns a part property typed by a block "upstream," leading to recursion in the structure tree.
PARTTYPE	error	All part properties must be typed.
PERFORMANCEFUNCTIONREFINE	error	Performance requirements must refine one or more functional requirements.
PHYSICALARCH	error	Part properties that are owned by a block with a physical stereotype must

Name	Severity	Error Message
		be typed by a block with a physical stereotype.
PHYSICALPORT	error	All proxy ports owned by blocks with the <<physical>> stereotype applied must be typed by interface blocks that have the <<physical>> stereotype applied.
PHYSTERMPARTS	error	Physical blocks with ATOMIC = TRUE may not own part properties.
PROXYPORT	error	All ports must be proxy ports.
PROXYPORTTYPE	error	Proxy ports must be typed by interface blocks.
REALIZEDIRECTION	error	Realization relationships between logical and physical elements must have the physical element as the source and the logical element as the target.
RECEPTIONPROHIBIT	error	Receptions are prohibited; use operations instead.
REFPROPPROHIBIT	error	Reference properties are prohibited. These may be represented as part properties at a higher level in the system model structure.
REGIONNAME	error	Regions of orthogonal states must be named.
REQEXTEND	error	Non-extended requirements are forbidden.
REQTRACE	error	Requirements must have at least one outgoing trace (to artifact) or refine relationship.
REQUIREMENTSATISFY	info	This requirement does not have any satisfy relationships. (Requirements that have blank text are exempted from this rule).
REQUIREMENTVERIFY	info	This requirement does not have at least one verify relationship. (Requirements that have blank text are exempted from this rule).
SENDINCOMING	error	If incoming object flows to a Send Signal event are realized by an item flow or flow set, the signal of the event must be conveyed by the item flows or flow set.

Name	Severity	Error Message
SENDSIGNALMATCH	error	The signal sent by a send signal action must match the signal typing its input pin.
SENDSIGNALPIN	error	Send signal actions must have at least one input pin.
SENDSIGNALPORTMATCH	error	The assigned and inferred ports (via item flow realization) must match.
SEQUENCELEVEL	error	Sequence diagrams may not mix physical and logical lifeline types.
SIGNALDOCUMENTATION	error	All signals must have documentation.
SIGNALEVENTSIGNAL	error	Signal Events must have a signal defined.
SIGNALNAME	error	All signals must be named.
SOFTWAREFUNCTION	info	This software element does not own any operations.
SRCCNT	error	All source content elements must have either a file name or hyperlink.
STATEDOCUMENTATION	error	All states must have documentation.
STATEMACHINEOPERATIONS	error	State machines may not own operations in their structure (move operation to a block or activity).
STATENAME	error	States must be named.
STATEOWNER	error	State machines must be owned by blocks.
STATEREACHABILITY	error	All states must have at least one incoming transition.
STMINTEGRITY	error	State machines may only call operations owned within their owning block's structural decomposition (owned by blocks typing its parts).
SUBMACHINECONNECTIONS	error	States that are submachines must have all entry and exit points associated with connection points.
SWIMLANEPROHIBIT	error	Swimlanes are prohibited; see customizations for operations and flows that can display part-level ownership if operations are owned by blocks. Dynamic legends may also provide similar functionality to swimlanes in a more compact representation.
TIMEEVENTWHEN	error	All transitions triggered by time events must have WHEN defined.
TRANSITIONCHOICE	error	All transitions exiting a choice must have guards defined.

Name	Severity	Error Message
TRANSITIONSOURCE	error	No operation owns an output parameter typed by the signal (or its general classifier) that triggers this transition.
TRANSITIONTRIGGER	error	All transitions (except those exiting connection points or pseudostates) must have triggers.
TRANSITIONTRIGGERFLOW	info	This transition is triggered by a signal but is not associated with any item flows or flow sets.
TRIGGERFLOWMISMATCH	error	The signal triggering this transition is not conveyed on any related item flows or flow sets.
UCASSOCIATION	error	Use cases may not be associated with other use cases.
UCDOCUMENTATION	error	All use cases must have documentation.
UCTRACE	error	All use cases must have an outgoing trace, extend, or refine relationship or an incoming include relationship
USECASENAME	error	Use Cases must be named.
VALUENAME	error	Value properties must be named.
VALUETYPE	error	Value properties must be typed by value types.