Bidirectional Text-to-Model Element Requirement Transformation

Marlin Ballard
Georgia Institute of Technology
270 Ferst Drive NW
Atlanta, GA 30332-0150
Marlin.B.Ballard@gatech.edu

Russell Peak
Georgia Institute of Technology
270 Ferst Drive NW
Atlanta, GA 30332-0150
Russell.Peak@gatech.edu

Selcuk Cimtalay
Georgia Institute of Technology
270 Ferst Drive NW
Atlanta, GA 30332-0150
cimtalay@gatech.edu

Dimitri Mavris
Georgia Institute of Technology
270 Ferst Drive NW
Atlanta, GA 30332-0150
Dimitri.Mavris@aerospace.gatech.edu

Abstract—**Elicitation, representation, and analysis of requirements are important tasks performed early in the systems engineering process. This remains true with the adoption of Model-Based Systems Engineering (MBSE) methodologies. Existing SysML-based methodologies often choose between (i) using external requirements documents and/or databases as the authoritative source for requirements truth versus (ii) generating requirements directly, as elements in the system model. In either case, there is often need for the systems engineer to manually develop a model-based requirements representation, as this faculty is not automatic in the commonly-used SysML feature set. Additionally, once the system model has been completed, systems engineers typically must prepare traditional "shall-statement" requirements for external review purposes, as not all stakeholders can be expected to be trained in system model interpretation.**

This paper details a novel effort to address both problems, by automatically transforming text-based requirements (TBR) into SysML model-based requirement (MBR) representations, and vice versa. The text-to-model based transformation direction uses requirement templates and natural language processing techniques, expanding on work from the field of requirements engineering. This paper also presents an aerospace-domain case study application of the developed tool. In the case study, a selected set of requirements were analyzed, and a system model was constructed. Then, the intermediate output system model was updated with additional elements, to represent the progression of the project's systems engineering process. The modified system model was then analyzed, constructing text-based requirements from the structure. The resulting text-based requirements were compared to the initial set of input requirements to assess consistency in both directions of analysis.

The methodology developed in this paper improves the systems engineering process by saving the systems engineer time constructing potentially repetitive model elements, and by enabling model-based requirement analyses to methodologies previously only capable of processing text-based requirements. Further, the methodology eases the responsibility of the systems engineer to maintain a copy of the model-based requirements in text-based format.

TABLE OF CONTENTS

1. INTRODUCTION

As systems become more complex, good practices for requirements engineering become paramount in reducing an engineering project's cost-due-to-error. With the increasing adoption of Model-Based Systems Engineering (MBSE) into systems engineering practice, care must be taken so that requirements documents and specifications do not become one of the examples of artifacts "thrown over the wall," i.e. disconnected from the efforts to create a lasting, unified representation of the system. With the International Council on Systems Engineering (INCOSE) Systems Engineering Vision 2025 setting goals for a complete acceptance of formal systems modeling as best practice[1], widespread adoption of MBSE is already occurring. Looking particularly at MBSE methodologies utilizing the OMG Systems Modeling Language (SysML), much effort has been placed into enabling the modeling and simulation components of design[2]. However, less effort has been introduced into understanding how requirements are treated differently in model-based practices in ways that leverage SysML.

978-1-7821-2734-7/20/$31.00 ©2020 IEEE

Paper Organization

The remainder of Section 1 discusses in more detail the motivation surrounding representing model-based requirements in SysML-based MBSE methodologies. It additionally discusses how a new feature of the SysML language enables the automated representation of these model-based requirements. Section 2 summarizes a literature review of the current state of practice in Requirements Engineering and Systems Engineering, and their treatment of model-based requirements. Section 3 of this paper details the methodology employed to transform text-based requirements into model-based requirements. Section 4 details the methods necessary for the reverse process: transformation from model-based requirements to text-based requirements. Section 3 and Section 4 both include applications of the methodology to an aerospace case study. Section 5 offers conclusions and directions for future research.

Motivation

Since the publication of STEP AP-233 (now ISO 10303-233:2012[3]), the concept of text-based requirements (TBR), property-based requirements (PBR), and model-based requirements (MBR) has been acknowledged and considered by the requirements engineering community[4]. Briefly, TBRs are the typically-understood "shall statements," written in natural language. PBRs add some formalism by including one or more properties to accompany the text. Finally, MBRs use system model constructs to represent the knowledge captured by the TBR requirement text. Papers such as [5] have been published to expand upon and attempt to formalize the topic. Methods for implementing PBRs have been included in the non-normative annexes of the SysML specification [4].

PBRs and MBRs have the potential to enable more formal knowledge capture, but come at an overhead of training and implementation. In modern practice, text-based requirements have remained the staple format of requirements. One reason for this is accessibility: it is natural for a stakeholder to express requirements in natural language during an elicitation process, without taking the effort to identify properties or model elements that represent them. Additionally, requirements appearing in contract awards have legal significance: a standard system of authoring PBRs and MBRs must be accepted to have legal meaning. Transition towards this has been underway, with NAVAIR investigating the use of system models as government-furnished information in a request for proposal (RFP)[6].

Even for systems engineers who have been trained in the application of PBRs and MBRs, the issue of communicating the meaning of the system model with untrained stakeholders remains. Just as it is unreasonable to expect a stakeholder to understand the source code of a software delivery, it is unreasonable to expect the stakeholder groups to learn a modeling language to interact with the engineering team. Even a stakeholder familiar with SysML would require an understanding of a particular engineering organization's modeling conventions. With this in mind, the stakeholder will probably always need some set of text-based requirements to consume, without requiring the systems engineering domain knowledge. Then, the systems engineer will need to translate between the TBR interactions with the stakeholders, and the MBR interactions with the system model for the engineering team. This translation, if performed manually, is a tedious process, especially if there are many repeated requirements for similar components used throughout the system. The vision of the method developed in this paper is to provide automation for this TBR-to-MBR transformation.

Background

As the SysML profile was developed specifically for use of systems engineers, one addition to the original specification was an element specifically for the tracking of text-based requirements: Requirement. In the original SysML specification[7], Requirement was a specialization of the UML Class, with additional properties and constraints. In particular, the base Requirement element was constrained to contain simply an "id" and "text" field, for representing a numeric identifier, and the requirement text, respectively. Additionally, particular dependency relationships were included to describe relations between the requirements and other system model elements. These include the Satisfy, Verify, and DeriveReqt dependencies. In particular, the Satisfy and Verify dependencies allow the modeler to allocate system design knowledge found in defined structural and behavioral elements to Requirements, as demonstrated in [8]. This enables traceability between the system design and requirements, as is needed for Systems Engineering practice.

Nevertheless, the base Requirement element in SysML 1.x suffers from innate deficiencies which prohibit it from being used to represent PBRs or MBRs in a direct and effective manner. Despite having possible dependencies to other parts of the system, there is no inherent model-based semantic meaning captured by a particular Requirement. The "text" field is a string, typically in natural language, so there can be no model-gnostic connection between the knowledge represented in field and the rest of the model. Thus, although the SysML 1.x Requirement element provides a way one can express a requirement in a model element, it is not a model-based requirement. A second deficiency of the Requirement element affects its usability with other model elements. Constraint 4 imposed on Requirement in SysML 1.0 states that Requirement may not participate in associations. This severely limits the structure permissible for the Requirement element. The ownedAttribute property of Class is how elements typically own Properties[9], such as ValueProperties of a Block (note: the capitalization of "Property" indicates the SysML model element, not to be confused with an element's "properties", fields that provide meta-information about the element). However, ownedAttribute is an Association between the Class and its Property, which means that SysML 1.x Requirements cannot have Properties in the usual way.

The limitations of the base Requirement are acknowledged in even the first implementation of the SysML specification.

Annex C of SysML 1.0 offers several non-normative stereotypes to enhance the use of the base Requirement[7]. This includes extendedRequirement, which adds properties to capture risk information, verification methods and requirement source. There are also stereotypes such as functionalRequirement and performanceRequirement which impose constraints on the types of model elements which can be in a Satisfy dependency relationship with the Requirement. For example, only behavior elements may satisfy a functionalRequirement. External tools also make use of stereotypes to enhance the capabilities of Requirements. The MBSEPak software by Phoenix Integration[1] makes use of a stereotype called PropertyBasedRequirement (thus named, one implementation of PBR), which adds tagged values to apply upper and lower bounds, as well one as to track units. The software then uses those bounds to verify whether requirements have been satisfactorily met when performing external analysis. Such stereotypes can be used for modeling guidance, whether or not the integration with other external tools is used.

The above implementations (extendedRequirement, PropertyBasedRequirement) bypass Constraint 4's structural limitation by making the Properties belong to the stereotype instance instead of to the requirements themselves. In this case, they are referred to as "tagged values". This returns to the question above regarding where the knowledge is captured. Here, the tagged values are truly model elements, and can therefore be displayed in SysML diagrams and utilized in other automated/computer-sensible ways. While it is possible to mix tagged values and Properties in SysML diagrams, this is obfuscating the nature of the tagged value, implying the tagged value is actually a Property of the Requirement. In reality, the upperBound does not belong to the Requirement element, but to the PropertyBasedRequirement stereotype instance that connects to it. Trying to access upperBound in the modeling tool reveals this. This can be confusing to both model authors – who may not know where to look to access the properties they want – and to superusers – who may not know the proper mechanism by which to add needed fields for their projects' requirements. A better solution would allow the Requirement model element itself to possess the properties directly.

Recent Improvements

In commonly-used versions of the SysML language, translation between Requirement text field and system model elements can take the form of the Trace and Refine dependency relationships. Automated verification packages do exist which can examine the requirements text and compare numeric values found within to ValueProperties connected via a Verify relationship [10]. However, from the true SysML model view, the system model element is simply connected to the requirement text, without any semantic meaning contained by the connection. Adding a property with a PBR or a more complex model element with an MBR allows the system model element to have unambiguous traceability to requirement model elements.

Fortunately, Version 1.5 of the SysML specification offers a solution to direct PBR and MBR creation by changing the underlying metamodel of the Requirement element. To preserve backwards compatibility, the Requirement element remains a specialization of Class, and remains constrained to not participate in Associations. However, the text and id fields, as well as the participation on dependencies are made instead inherited properties of a stereotype AbstractRequirement, which the Requirement element uses. AbstractRequirement can be applied to any NamedElement, making it far more general than Requirement. Applied in this way, any NamedElement may represent a requirement, with the associated text and dependencies. For instance, the motor part property of a Vehicle block might represent the requirement "The vehicle shall have an electric motor." Value properties or constraint blocks can be naturally used to represent PBRs, such as "The vehicle mass shall be no more than 1,000 kg." Other structural features such as ports and flows could represent interface requirements, such as "The flight computer shall communicate with the instrument computer using MIL-STD-1553." The new AbstractRequirement stereotype in SysML 1.5 enables systems engineers to begin using MBRs.

Despite being an improvement to SysML-based MBSE methods, several issues still prevent the widespread implementation of AbstractRequirement-centered methodologies. As with practically any non-trivial language, training systems engineers how to utilize SysML properly has been a prevalent issue. Since SysML 1.0 was based in the software engineering thinking of UML, SysML has many formal constructs and patterns that are not typically taught in an engineer's general training. SysML diagrams are also very information dense – concisely displaying relationships which can seem obvious, but take many phrases to describe in text. The difficulty in training systems engineers is exacerbated in the case of AbstractRequirement, because it is a recent addition to the specification. The Object Management Group offers a certification exam series for SysML users, which is based on SysML 1.2[2]. This certification exam serves as a way to validate an engineer's SysML training, so some employers may suggest it as an end state for training. Indeed, some commercial trainings are specifically designed for the SysML 1.2 exam[3], so the thinking needed for AbstractRequirement is understandably beyond the scope of the training (note: this is not meant as a criticism of the OCSMP exam training material, since recent updates to the SysML specification may be misleading when considered under the lens of the SysML 1.2 specification). The methodology presented in this paper employs the AbstractRequirement stereotype to demonstrate its capability for capturing requirement knowledge in PBRs and MBRs.

[1] https://www.phoenix-int.com/product/mbse/

[2] https://www.omg.org/ocsmp/faqs.htm

[3] https://delligattiassociates.com/services/

2. Literature Review

Existing MBSE Methodology Treatment of Requirements

To understand how automatically-generated MBRs will assist MBSE methodologies, it is necessary to consider how such existing methodologies handle the treatment of requirements. The following is not an exhaustive list of MBSE methodologies, but a sampling to demonstrate different practices regarding requirements ([11] offers a more detailed survey of MBSE methodologies). IBM Rational Harmony is a methodology tied to the IBM Rational toolset (DOORS for requirements database, Rhapsody for SysML editor), but it can be applied by a project in a tool-agnostic approach[12]. In the Rational Harmony methodology, requirements are first elicited of stakeholders and stored in a (DOORS) database. The database is able to store property data as well as metadata such as linking requirements or providing verification methods. This database is treated as the authoritative source of stakeholder requirements, and an import capability is used to convert the requirement texts into Requirement elements in a SysML system model. Next, systems engineers perform a decomposition of system functionality, until they reach the level of the physical system under consideration. If it is discovered during this decomposition that requirements need to change, the requirements are propagated back to the database to maintain the database's requirement authority. This is a SE methodology which applies the MBSE components separately from its Requirements Engineering components.

By contrast, the Object-Oriented Systems Engineering Methodology (OOSEM) is more holistic, directly incorporating ideas of MBRs into the methodology[13]. OOSEM begins with the modeling of the environment in which the desired system will perform. This as-is model is shared with the stakeholder in order to elicit TBRs about the desired system. Then, the necessary system capabilities are decomposed into behaviors. These behaviors may be represented by the standard SysML behavior constructs. In OOSEM, these black-box functions are treated as first-class requirements in their own right. Thus, a valid system implementation would need to satisfy the behavior MBRs as well as the TBRs.

A third treatment of requirements in an MBSE methodology is a non-standard methodology developed in [14] specifically for aircraft design. In this paper, the methodology represents an example of fit-to-purpose methodologies developed by systems engineers to apply to particular domains. The methodology in [14] begins with stakeholder TBRs and follows a Requirements-Functional-Logical-Physical decomposition, where the lowest level of functions is used to derive additional system TBRs, which are treated as first-class requirements. This example uses TBRs as its requirement source at all levels, with the additional feature of performing MBSE methods to derive the lower-level TBRs. All three of these methodologies begin with the elicitation of TBRs from stakeholders, followed by a model-based decomposition. In each case, there is a step which could be assisted by automated TBR-to-MBR transformation.

Requirement Templates/Boilerplates

Requirements Engineering has focused a significant amount of research into what makes a quality requirement. One hindrance to Hooks's Necessary, Verifiable, Attainable[15] characteristics or Zowghi and Gervasi's Consistency, Completeness and Correctness[16] is the ambiguity inherent in the natural language representation of requirements. A common practice to curb this ambiguity is to reduce the scope of all possible utterances of natural language. This is accomplished by imposing a particular grammar on requirements documents. To employ this technique, engineers form requirements in accordance with a number of combinable snippets of generalized text, known as requirement boilerplates, requirement patterns, or requirement templates. This was first popularized by [17]. One example, from [18], is "<Condition>, the <Element> shall have a <Property> of <Comparator> <Value> <Confidence> <Unit>." In an actual system requirement, the text within chevrons would be replaced with information about the system of interest, such as "When equipped with all the necessary instruments, the aircraft shall have a gross take-off weight of less than or equal to 200 kg" for the above template. [19-21] each provide additional examples of requirement templates. [22] offers a survey of how requirement patterns are being used by engineers in industry.

Natural Language Processing

Requirements Engineering has leveraged Natural Language Processing (NLP) capabilities to enable computer processing of requirements. Most modern NLP techniques make use of corpora of tagged sample text in the language of interest[23]. A typical Natural Language Processing pipeline consists of several important independent techniques, described briefly here. Detailed accounts of each of the phases listed may be found in [23]. The NLP process begins with a string of natural language text. This string is then tokenized into a vector of symbols. Tokenization is the process of identifying elements of text, such as words or punctuation within the string. Tokenization can be performed at various levels, identifying sentences before breaking those sentences into individual words and punctuation. Individual tokens are then classified using Part-of-Speech (PoS) tagging.

A PoS tagging classifier uses the corpora as training data to identify the parts of speech of the vector of tokens. The classifier makes use of standard PoS labeling conventions, such as the Treebank tagset[4]. Once tagging is complete, adjacent, similar parts of speech may be combined through a process called chunking. As an example, consider the text "systems engineer." Assume that the PoS tagger has identified "systems" as an *NNS* (plural noun) and "engineer"

[4] https://ling.upenn.edu

as *NN* (singular noun). It is desired, in this context, to consider the systems engineer as one entity, so chunking can be used to combine the adjacent *NNS* and *NN* into a single noun phrase (perhaps labeling the phrase with a common-but-nonstandard *NP*). Grammar rules to split portions of identified chunks can be helpful, and this process is referred to as "chinking".

Following chunking and chinking, an application may then process the chunk patterns and contents to extract knowledge from the previously natural-language string. One additional important concept in NLP techniques is term recognition. Term recognition identifies particular tokens as they appear through the overall text to produce a continuity of knowledge. For example, the noun phrase "aircraft system" could be referenced many times in a requirement document, with each instance providing more information about the aircraft. Term recognition can utilize an existing ontology or glossary defining the possible terms of interest [24], or can build from scratch a glossary of commonly-used terms found within the text of interest.

There have been several successful attempts to use NLP techniques in Requirements Engineering in conjunction requirement templates. [25] uses particular chunking patterns to identify which, out of several requirement templates, applies to a requirement as written. The DODT tool in [26] has similar functionality, combining the results with an inputted ontology to judge whether all features of an ontology have been covered. It also makes suggestions on how to rewrite requirements that do not fit any of the relevant templates. The Framework for Linking Ontology Objects and Textual Requirements (FLOOR) in [24] gives an alternative to DODT with a different scope of use. FLOOR performs the PoS tagging live as the requirements engineer is typing the requirement, suggesting continuations which will aid in following templates. Note that all of these examples identify the requirement template which is being followed.

MBSE research has also begun to explore use of NLP techniques. The Europa Mission Model-Based Analysis of Requirements (EMBARQ)[18] project made use of requirement templates and term recognition – separately – to compare an existing set of TBRs to an existing system model. The requirement templates allowed the EMBARQ team to create formal models of the constraints imposed by the requirements. Then, the formal model was compared to a formal implementation of the system to check for inconsistencies. Term recognition was used to verify whether all elements in the TBRs had associated model elements representing them, and conversely whether all system model major components were representing a TBR, and thus not superfluous. [27] presents a custom NLP classifier for use in parsing small UAS systems. Rather than the general corpora of English language text typically used for training NLP classifiers, this classifier was trained on a set of requirements used exclusively for UAS systems. This enabled the classifier to identify particular types of requirements, such as a duration requirement or distance requirement. [28] presents an application in knowledge extraction using a large ontology referred to as the Engineering Knowledge Base. The team utilizes a generalized Natural Language Understanding tool to extract information about the capability needs of the system and offers suggestions on company products which can satisfy the needs based on the Engineering Knowledge Base.

3. Text-to-Model–based Requirements Transformation

Towards the goal of automating the transformation between TBR and MBR, several components are required. For the scope of this paper, requirements are assumed well-formed to a set of pre-defined requirement templates (using an approach like [26] or similar). Given that context, the overall process flow for our method is shown in Figure 1. The semi-formal requirements are passed through the Requirement Parsing phase, which is an NLP pipeline – where PoS information is extracted. The structure of the tagged tokens is analyzed in order to identify the particular template a requirement applies to. Once all requirements have passed through the template identification step, like components in sentence structure are analyzed to identify keywords so duplicate model elements are not constructed. For instance, two noun phrases acting as subject of separate requirement texts are compared to determine if they refer to the same entity. During the Model Construction phase, modeling patterns are applied to the resulting relationship graph, and the system model is constructed.

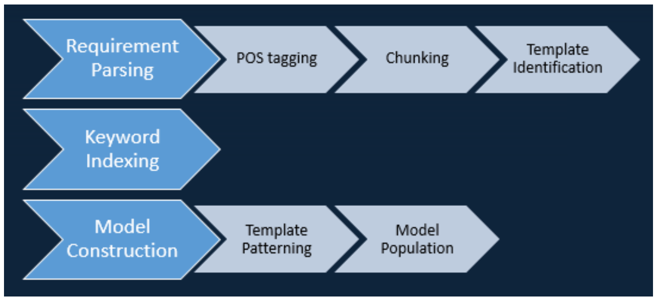


Figure 1. Text-to-Model–based Requirement Transformation Methodology

Requirement Templates

To streamline the case study for this paper, the structural requirement templates from the EMBARQ presentation [18] are used. Additional templates for state-based behavior are added from the work described in [29], originally derived from actual Request for Proposal (RFP) specifications. These templates are presented in Table 1. Other than the Composition and State Description templates, the requirement types are pairwise disjoint, simplifying the template identification process. The Composition–State Description similarity also acts as a unit test for the template identification grammar's ability to process syntactically similar templates.

Table 1: Requirement Templates

Requirement Type	Template	Source
Property	The <Element> shall have a <Property> of <Comparator> <Value> <Confidence> <Unit>.	EMBARQ[18]
Composition	The <Element> shall have a <Sub-Element>	EMBARQ[18]
State Description	The <System> shall have a <State> state.	NAVAIR SET[29]
State Transition	The <System> shall allow transition from the <source> state to <target> state.	NAVAIR SET[29]
Substate Decomposition	The <System> shall have a <Mode> mode in the <State> state.	NAVAIR SET[29]

Requirements Selection

Two sets of sample requirements were generated for this initial proof-of-concept case study. The first is an adaptation of a standard example from the SysML training material found in [30]. The example is of a generic vehicle that contains a fuel tank. The vehicle contains structural information, shown in Figure 2, as well as the state-based behavior information shown in Figure 3. These figures only represent a portion of the underlying system model, chosen to present an example of notable features used as AbstractRequirement. The Block Definition Diagram shown here also intentionally elides that there are additional requirements associated with the Fuel Tank's capacity and mass values. This is a demonstration of the idea of encapsulation of diagrams. Individual diagrams represent a particular view of the system model, rather than being the model themselves. In generating the AbstractRequirements, our case study places no emphasis on the presentation of the model elements in diagrams; how best to do that is left to the systems engineer as a method of displaying requirements to stakeholders or domain engineers according to their needs.

A second set of requirements was adapted from the scrubbed RFP regarding a Vertical Takeoff Uninhabited Aerial Vehicle (VTUAV) system as presented in [29]. These were furnished for use in research as sample TBRs used in an actual RFP. The TBRs mapped ideally for use in state-based behavior, and the text needed minimal modification to conform to the state-based requirement templates listed above. A breakdown of the requirement counts by category is presented in Table 2.

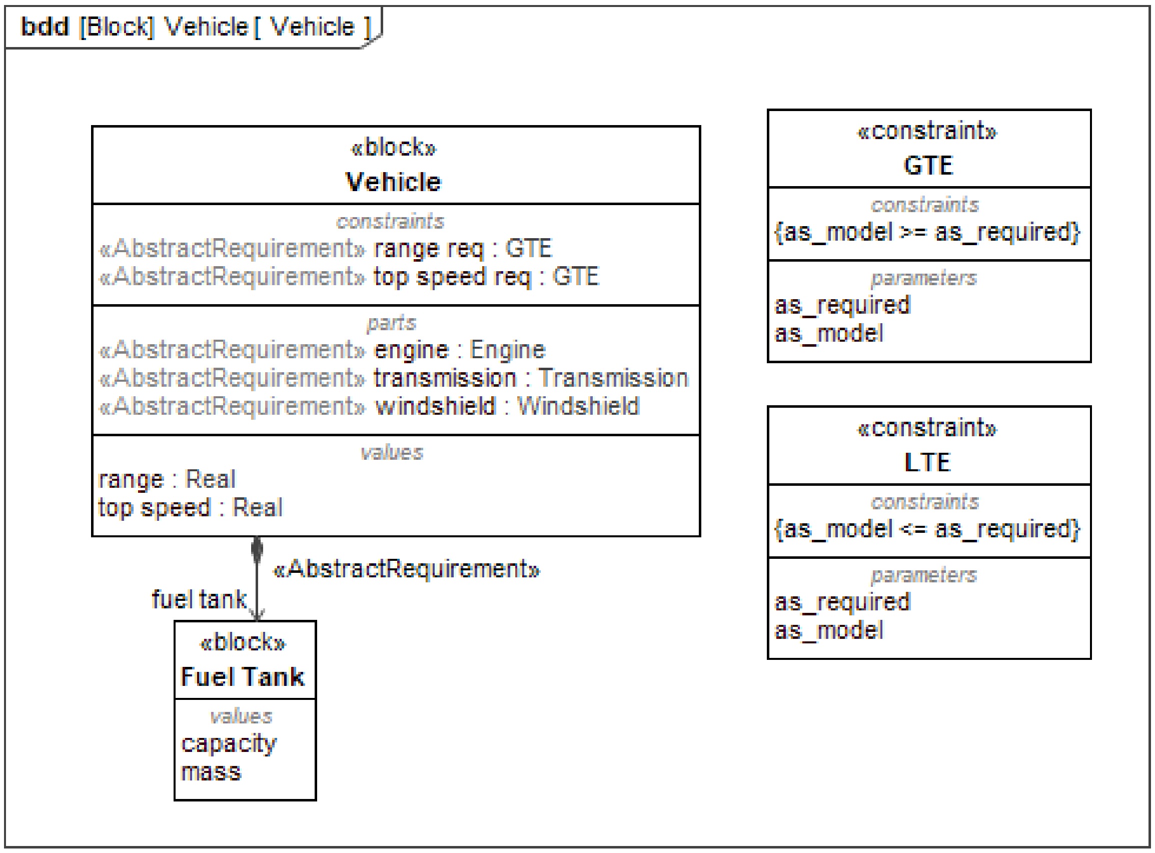

bdd [Block] Vehicle [Vehicle]
«block»
Vehicle
constraints
«AbstractRequirement» range req : GTE
«AbstractRequirement» top speed req : GTE
parts
«AbstractRequirement» engine : Engine
«AbstractRequirement» transmission : Transmission
«AbstractRequirement» windshield : Windshield
values
range : Real
top speed : Real
«AbstractRequirement»
fuel tank
«block»
Fuel Tank
values
capacity
mass
«constraint»
GTE
constraints
{as_model >= as_required}
parameters
as_required
as_model
«constraint»
LTE
constraints
{as_model <= as_required}
parameters
as_required
as_model

Figure 2: Vehicle Case Study Structure

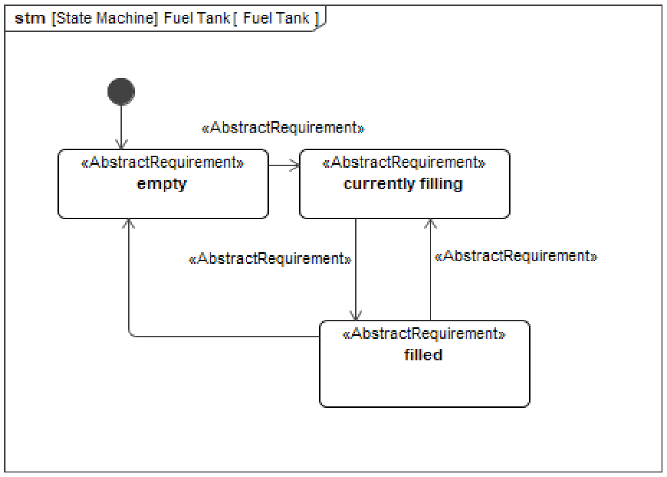

stm [State Machine] Fuel Tank [Fuel Tank]
«AbstractRequirement»
«AbstractRequirement» empty
«AbstractRequirement» currently filling
«AbstractRequirement»
«AbstractRequirement»
«AbstractRequirement» filled

Figure 3: Vehicle Case Study Behavior

Table 2: Requirement Template Use by Type in Our Case Studies

Case Study	Composition	Property	State Description	State Transition
Vehicle	4	4	5	3
VTUAV	0	4	4	6
Total	4	8	9	9

The approach taken used the two requirement sets for different purposes. Using machine learning parlance, the Vehicle requirement set was used for manual "training" of the NLP configuration, and the VTUAV requirements were used as verification.

Part of Speech Tagging

The NLP Pipeline consists of a tokenizer, a part-of-speech tagger, and a chunker. The current NLP Pipeline is implemented in the open source Python-based NLTK[5] package. The Text-Based Requirements chosen are each individual sentences, so the step and ambiguity of sentence identification can be omitted, simplifying the job of the tokenizer. Since this methodology is designed to be ontology-free, an off-the-shelf tagger (MaxEnt Treebank tagger) is used for general Part-of-Speech tagging. It is also possible to add a customization of particular tags to use with certain tokens. There are several reasons why this may be desired. The first reason for adding custom tagging models is the "state" example; it is known *a priori* that in these requirement templates, the word "state" is reserved for the template. Thus, we can enhance the tagging of "state" by ensuring that the tagger will identify it as a custom tag: *NNB*. The *NNB* tag is our custom extension of the Penn Treebank tagset, representing a "behavioral noun," a category also including the word "mode." The standard noun in the Penn tagset is *NN*. As discussed in the chunking section, the NLTK package uses a regular expression basis for its chunking rules. Then, the use of the custom *NNB* tag will allow "state" to be queried for explicitly by using the regular expression "*NNB*", while additionally allowing it to match queries searching for the more general "*NN.**". The latter is used when identifying noun phrases, for example, since the object of a transitive verb could be any choice of a singular noun (*NN*), plural noun (*NNS*), or proper noun (*NNP*), in addition to our behavioral noun.

A second motivation requiring tagger modification – a lesson learned discovered during the testing process – is when the default tagger lacks context for the language used in the requirement templates. This situation came about when verifying the tags used in the State Transition template requirements. The PoS tagger was struggling with the phrase "The <system> shall allow transition from…". In each case, it incorrectly identified "allow" as a singular noun (*NN*). This led to the construction seen in the left of Figure 4, where "allow transition" was incorrectly chunked as a noun phrase,

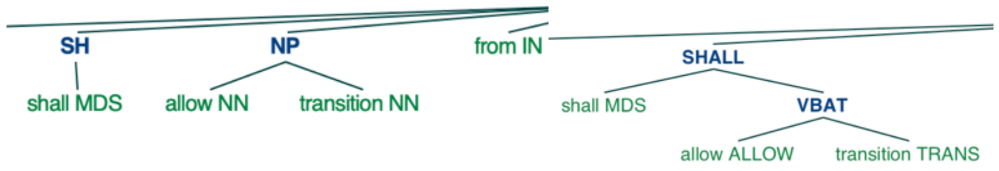


Figure 4: Improper Tagging of "Allow" (left) vs. Custom "ALLOW" Tag (right)

[5] https://www.nltk.org

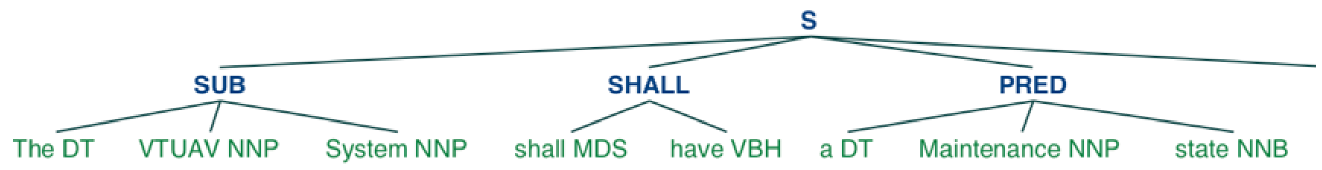


Figure 5: Intermediate Chunking Showing Triples

leading to template identification failure. Note that in each of these figures (4-8), green text indicates PoS tagging output, and blue text indicates named chunks. To overcome the misidentification, "*ALLOW*" was added as a special tag to the tagger, so that explicit chunking of the verb phrase "allow transition" could be completed.

Chunking and Chinking

A careful approach to chunking eases both the template identification and keyword indexing steps to follow. One pattern shared between each of the utilized requirement templates is "The <system> shall <verb phrase> <predicate>. Thus, a splitting of the sentence structure into such a triple (<SUB> <SHALL> <PRED>) means that template identification needs only consider the predicate chunk to properly identify the used template. An intermediate phase of chunking is shown in Figure 5. The only necessary continued processing of the <SUB> chunk is a simple determiner-chinking rule which removes the prepended "The". The processing of <PRED> chunk uses patterns suitable for the particular templates utilized. The phrase patterns remain general, where possible, to preserve the usefulness of the NLP technique.

An example of how chunking patterns are generalized for composition and reuse is the grammar rules for phrases describing states. First, the PoS tagger was modified to enable the *NNB* tag for "state" and "mode". Then, the phrase preceding the *NNB* is identified as the name of the particular state. The construction <DT> <NAME> <NNB> is stored as a "state phrase" <STATEP>. This is all that is needed for the State Description template, seen in Figure 6. The Substate Decomposition pattern (Figure 7) introduces a prepositional phrase "in the <state> state." To parse this phrase, the chunker first applies the <STATEP> rule, which matches both the "mode" and "state" phrase. The chunker then applies the "state prepositional phrase" <INPS> rule <PREPOSITION> <STATEP>, which only matches the state

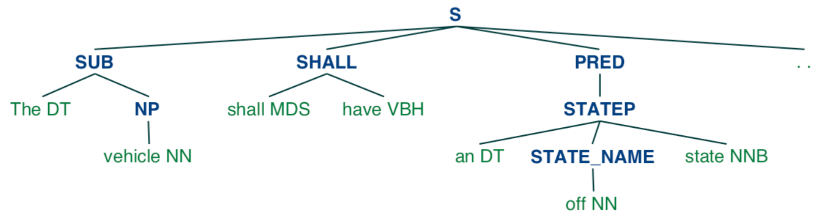


Figure 6: Complete Chunk Parsing for State Existence Template TBR

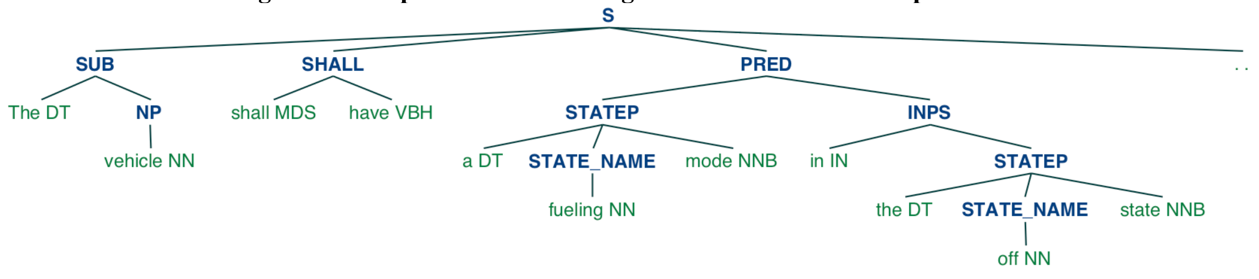


Figure 7: Complete Chunk Parsing for Substate Decomposition Template TBR

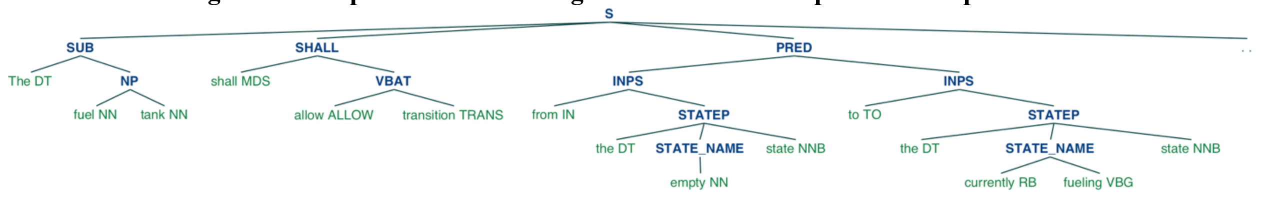


Figure 8: Complete Chunk Parsing for State Transition Template TBR

Table 3: Identification Patterns Per Requirement Template

Requirement Type	Predicate Pattern	Source
Property	*<DT> <NP>*	DT: Determiner NP: Noun Phrase
Composition	*<DT> <NP> <INC> <CDU>*	INC: Comparator Prepositional Phrase CDU: Cardinal Number (with optional Unit)
State Description	*<STATEP>*	STATEP: State Phrase
State Transition	*<INPS> <INPS>*	INPS: State Prepositional Phrase
Substate Decomposition	*<STATEP> <INPS>*	- (*uses rules defined above*)

phrase. The same state prepositional phrase rule applies to the State Transition pattern (Figure 8), working for both the source and destination.

Similar procedures were executed to chunk TBRs that match the Composition and Property templates, repeating the pattern for prepositional phrases and name identification. The chunking library uses an identical regular expression matching technique for chunk names as PoS tags, meaning a query for <IN.+> will yield both standard prepositional phrases <INP> and the state-based prepositional phrases <INPS>.

Template Identification

In the Template Identification step (Figure 1), the tree structure of the tagged predicate chunks is compared against the following grammar rules. The detailed chunking parsing means that a simple, non-recursive grammar suffices. For test of template identification, requirement text was classified according to template, and checked against the predicted template identification. The template identification section had 100% success identifying both training and verification sets; however, some of the internal chunking grammar needed modification to have appropriate splitting between chunks when moving to the combined dataset.

Model Construction

Given the appropriate pattern, the Model Construction step (Figure 1) develops the structure for the resulting SysML model. The Requirement Templates shown in Table 1 were selected to provide unambiguous meaning as to the implementation of the AbstractRequirement. For example, the Composition template will create a block named <Element> with a part property typed by a block named <Sub-Element>. Using the keywords obtained in the previous step, overlapping patterns were reduced, resulting in one model element with branching patterns, e.g. two part properties belonging to the same block, as seen in Figure 2.

This methodology is intended to be, for the most part, tool-agnostic. To this end, the Model Construction step develops a set of instructions which describe the resulting SysML model without being constructed of calls to the MagicDraw tool's OpenAPI. In addition to producing a representation of the model in a computer-usable data structure, our TBR-to-MBR prototype tool uses string templating to generate a set of plain-text instructions. A systems engineer can read these plain-text instructions and implement the model manually if desired (which serves as a kind of round-trip validation test). The results of this can be found in Appendix A. The individual actions taken represent generic SysML modeling instructions to be followed, so one can imagine a commercial application capable of making calls to the MagicDraw API, or that of other similar SysML-authoring tools, for each leaf-level instruction.

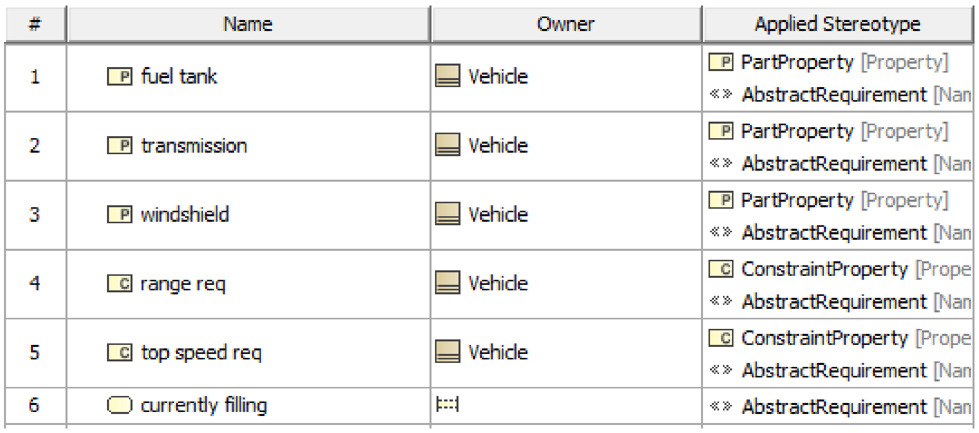
4. MODEL-TO-TEXT–BASED REQUIREMENTS TRANSFORMATION

The method for the inverse-direction MBR-to-TBR requirement transformation is simpler in structure than the TBR-to-MBR transformation. First, our method identifies MBRs in the model by querying for NamedElements stereotyped by AbstractRequirement. Then, the method selects the appropriate text template to populate. The tool fills the fields of the text template by querying the appropriate data from the model API. As implemented in Python, the tool uses the built-in string.Template utility. More difficult templates may need a more robust template engine.

This work expands upon an existing MBR-to-TBR method developed as part of [29]. The initial work utilized the OpenMBEE framework[6]. In particular, Viewpoint Methods were created to invoke external UserScripts to extract information from particular model elements of interest. In [29], AbstractRequirements were not considered as the tag to identify which elements would be used to generate TBRs. Instead, the model elements of interest were connected manually to the View on which they would be displayed using an Expose relationship. The MDK package of OpenMBEE would then package the exposed elements and invoke the UserScript, which would interrogate the model elements using the MagicDraw OpenAPI and extract the information needed to fill the templates. This was demonstrated for the state-based behavior templates described above, and for a different implementation of the Property template. This implementation also manually informed the execution which template style to fill.

The methodology this work proposes does not require the OpenMBEE software. Using AbstractRequirement means that a query can be made to the model which returns all elements having that stereotype. An example of such a query

[6] https://www.openmbee.org



#	Name	Owner	Applied Stereotype
1	fuel tank	Vehicle	PartProperty [Property] AbstractRequirement [Nan
2	transmission	Vehicle	PartProperty [Property] AbstractRequirement [Nan
3	windshield	Vehicle	PartProperty [Property] AbstractRequirement [Nan
4	range req	Vehicle	ConstraintProperty [Prope AbstractRequirement [Nan
5	top speed req	Vehicle	ConstraintProperty [Prope AbstractRequirement [Nan
6	currently filling		AbstractRequirement [Nan

Figure 9: GenericTable Querying Model for AbstractRequirements

is shown as a MagicDraw GenericTable in Figure 9. The third column demonstrates that multiple types of model elements are returned from the single query. The software is then able to make the appropriate model interrogation using the results of the query, proceeding using the same external scripts as the OpenMBEE implementation. In this case, the software can identify what is the base type of the AbstractRequirement model element, and thus invoke the appropriate translation script. As the translation UserScripts are called externally by the OpenMBEE MDK plugin, they can be repurposed to a general plugin which does not require the OpenMBEE software environment to execute.

5. Conclusions and Future Work

A successful proof-of-concept demonstration of the NLP-based text-to-model–based requirements transformation has been performed. From a reduced-order set of requirements templates, we were able to extract knowledge about the system-of-interest–as-required. From this knowledge extraction, a data structure was constructed and exported to be implemented as SysML model elements in MagicDraw as a representative SysML tool.

There are several additional improvements to the methodology that are planned for incorporation in future work. The requirement templates were selected for ease of implementation into a system model structure. The templates found in [17] and [21] do not have as natural model-based counterparts. A more detailed set of patterns will be developed to accommodate these requirement templates, and the patterns necessary to represent their MBRs in the SysML system model. Additionally, a larger body of requirements will be gathered for model validation, ideally including requirements from two or more current medium/large aerospace projects. Meanwhile, portions of this research are being applied in [31].

Looking toward the future of this research topic, several avenues of research remain open. The grammar for template identification is currently non-recursive. However, one strength of requirement boilerplates is that formal requirement utterances can combine boilerplates to form more complex requirements texts. Investigation into a recursive chunking grammar for template identification would further aid the systems engineering requirement elicitation task by allowing these more complex TBRs to still form automated transition to MBRs. The output model of such a transformation could be inspected with model complexity analyses to identify critical components of the inferred ontology. A second future research area is the use of metrics such as those presented in [18] for comparison between a body of TBRs and a system model. [18] used this metric to assess how well an existing set of requirements are covered by an existing system model. A modified interpretation of this could be used to compare between a set of TBRs and MBRs, offering an automated verification of the transformation procedure.

The prospects of this work improving the ability of systems engineers to perform MBR-enabled tasks are clear. Looking at the steps generated in Appendix A, a total of 67 steps were identified to implement the SysML-based MBRs based on the 18 text-based requirements. These steps are each SysML (tool-agnostic) instructions, and they individually typically require several atomic computer interactions. Performing these tasks manually does not add additional knowledge to the initial requirements analysis, so alleviating the systems engineer of this task frees them to spend more time performing actual design rather than performing more mouse-clicks.

References

[1] "A World In Motion: Systems Engineering Vision 2025." International Council on Systems Engineering, 2014.

[2] Reilley, K. "Modeling and Simulation for Model-Based Systems Engineering," AE8900 Report, Aerospace Systems Design Laboratory, Georgia Institute of Technology, Atlanta, GA, 2016.

[3] International Standards Organization. ISO 10303-233:2012, "Industrial automation systems and integration — Product data representation and exchange — Part 233: Application protocol: Systems engineering." Nov., 2012.

[4] "OMG Systems Modeling Language," Tech. Rep. 1.5, Object Modeling Group, May 2017.

[5] Micouin, P. "Toward a property based requirements theory: System requirements structured as a semilattice." Systems Engineering, Vol. 11, 2008, 235-245. doi:10.1002/sys.20097

[6] Blackburn, M., R., Bone, M. A., Dzielski, J., Grogan, P., Giffin, R., Hathaway, R., Henry, D., Hoffenson, S., Kruse, B., Peak, R., Edwards, S., Baker, A., Ballard, M., Austin, M., Coelho, M., and Petnga, L., "Transforming Systems Engineering through Model-Centric Engineering, Final Technical Report SERC-2018-TR-103", RT-170 (NAVAIR), February 28, 2018.

[7] "OMG Systems Modeling Language," Tech. Rep. 1.0, Object Modeling Group, Sept. 2007.

[8] Sindiy, O., Mozafari, T., Budney, C., "Application of Model-Based Systems Engineering for the Development of the Asteroid Redirect Robotic Mission." AIAA Space 2016, AIAA Space Forum, (AIAA 2016-5312). Sep. 2016.

[9] "OMG Unified Modeling Language," Tech. Rep. 2.5, Object Modeling Group, March 2015.

[10] "Cameo Simulation Toolkit 19.0 SP2 User Guide," No Magic, Inc., 2019.

[11] Estefan, J. 2008. "Survey of Candidate Model-Based Systems Engineering (MBSE) Methodologies", rev. B. Seattle, WA, USA: International Council on Systems Engineering (INCOSE). INCOSE-TD-2007-003-02

[12] Hoffmanm, H.-P.. IBM Rational Harmony Deskbook. Release 4.1. IBM Corporation, 2011.

[13] Friedenthal, S., Moore, A., and Steiner, R., A Practical Guide to SysML. Third Edition. Elsevier Inc, 2015.

[14] Karagoz, E., Reilley, K.A., and Mavris, D.M.. "Model-Based Approach to the Requirements Analysis for a Conceptual Aircraft Sizing and Synthesis Problem", AIAA Scitech 2019 Forum, AIAA SciTech Forum, (AIAA 2019-0498).

[15] Hooks, I.,"Writing Good Requirements", Proceedings of the Third International Symposium of the NCOSE, Vol. 2, 1993.

[16] Zowghi, D. and Gervasi, V., "The Three Cs of Requirements: Consistency, Completeness, and Correctness." Article, 2002. oai:CiteSeerX.psu:10.1.1.13.7861

[17] Dick, J., Hull, E., and Jackson, K. Requirements Engineering, Fourth Edition. Springer International Publication, Switzerland, 2017.

[18] Ernst, Z., and Wilson, C. "Europa Mission Model-Based Analysis of Requirements." ASDL External Advisory Board Presentation. Atlanta, May 2016.

[19] Rajan, A. and Wahl, T., (ed.), CESAR – Cost-efficient Methods and Processses for Safety-relevant Embedded Systems. Springer-Verlag Wein, 2013.

[20] Tommila, T. and Pakonen, A, Controlled natural language requirements in the design and analysis of safety critical I&C systems, VTT, 2013. (VTT-R-0167-17)

[21] Pohl, K. and Rupp, C. Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam – Foundation Level – IREB compliant. Rocky Nook, 2011.

[22] Palomares, C., Quer, C., and Franch, X., "Requirements reuse and requirement patterns: a state of the practice survey," Empire Software Eng. Springer Science+Business Media, New York, 2017. doi: 10.1007/s10664-016-9485-x

[23] Bird, S., Klein, E., and Loper, E. Natural Language Processing with Python. O'Reilley Media, 2009.

[24] Zontek-Carney, E. "FLOOR (Framework for Linking Ontology Objects and Textual Requirements): A New Requirements Engineering Tool that Provides Real-time Feedback," Master's Thesis, University of Maryland, College Park, MD, 2017.

[25] Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F. and Gnaga, R., "Automatic Checking of Conformance to Requirement Boilerplates via Text Chunking: An Industrial Case Study," 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, 2013, pp. 35-44. doi: 10.1109/ESEM.2013.13

[26] Farfeleder, S., et al., "DODT: Increasing Requirements Formalism using Domain Ontologies for Improved Embedded Systems Development." IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, 2011, p. 212–226.

[27] Justin, C.Y., Ramamurthy, A., Beals, N., Spero, E., and Mavris, D.N., "On-Demand Small UAS Architecture Selection and Rapid Manufacturing using a Model-Based Systems Engineering Approach", 31st Congress of the International Council of the Aeronautical Sciences, 2018. (ICAS 2018_0851). doi: 10.2514/6.2019-0498

[28] Feldman, Y.A. and Broodney, H., "A Cognitive Journey for Requirements Engineering," INCOSE International Workshop, 2016. IBM Corporation, 2016.

[29] Ballard, M. and Baker, A. "Facilitating the Transition to Model-Based Acquisition." ASDL 26th Annual External

Advisory Board Presentation, Georgia Institute of Technology. Atlanta, GA, 2018.

[30] Peak, R., De Spain, M, Steiner, R., et al., "SysML 101/201 BLS - Model-Based Engineering Fundamentals: Understanding and Creating SysML Models", Georgia Institute of Technology. Atlanta, GA, 2019.

[31] Ballard, M., Baker, A., Peak, R., Cimtalay, S., Blackburn, M., and Mavris, D., "Facilitating the Transition to Model-Based Acquisition," IEEE 41st Aerospace Conference 2020, IEEE Aerospace, Mar. 2020.

BIOGRAPHY

Marlin Ballard *is a PhD candidate at the Georgia Institute of Technology's Aerospace Systems Design Laboratory. He received his M.S. in Aerospace Engineering from Georgia Tech in 2019. He received a B.S. of Aerospace Engineering and a B.S. in Computer Science at the University of Maryland in 2015. At Georgia Tech, Marlin has three years of research experience in SysML-related MBSE. He has additionally held three Systems Engineering internships at the NASA Jet Propulsion Laboratory for the development of a collaborative MBSE framework. Marlin is an OMG Certified Systems Modeling Professional Model Builder - Intermediate.*

Russell Peak, PhD *is a Senior Researcher at Georgia Tech in the Aerospace Systems Design Lab where he is MBSE Branch Chief. Dr. Peak originated the multi-representation architecture (MRA)—a collection of patterns for CAD-CAE interoperability—and composable objects (COBs)—a non-causal object-oriented knowledge representation. This work provided a conceptual foundation for executable SysML parametrics. After six years in industry at Bell Labs and Hitachi, he joined the research faculty at Georgia Tech. He is an active INCOSE member where he co-leads the MBSE Challenge Team for MBX Ecosystems. He represents Georgia Tech on the OMG SysML task force, and he is a Content Developer for the OMG Certified Systems Modeling Professional (OCSMP) program. Since August 2008 he has led a SysML/MBSE training program that has conducted numerous short courses for working professionals.*

Selcuk Cimtalay, PhD *is a Senior Research Engineer at the Aerospace Systems Engineering Laboratory at the Georgia Institute of Technology. He formerly received his PhD. in Mechanical Engineering from GT. Model-Based Systems Engineering, He has several years of teaching, research, and industry experience. Dr. Cimtalay has conducted research on various projects on model-based systems engineering (MBSE).*

Dimitri Mavris *earned his B.S. (1984), M.S. (1985), and Ph.D. (1988) in Aerospace Engineering from Georgia Tech. He is the Boeing Chaired Professor of Advanced Aerospace Systems Analysis in Georgia Tech's School of Aerospace Engineering, Regents Professor, and Director of its Aerospace Systems Design Laboratory (ASDL). He is an S.P. Langley NIA Distinguished Professor, AIAA Fellow, Fellow of the Royal Aeronautical Society, and a member of the ICAS Executive Committee, the AIAA Institute Development Committee, and the US Air Force Scientific Advisory Board. He is also the Director of the AIAA Technical, Aircraft and Atmospheric Systems Group, and co-chair of the Committee on Aviation Environmental Protection's review board of independent experts.*

For the past 25 years, Prof. Mavris and ASDL have specialized in the integration of multi-disciplinary physics-based modeling and simulation tools. ASDL's signature methods streamline the process of integrating parametric simulation toolsets and enable huge runtime improvements that facilitate large scale design space exploration and optimization under uncertainty. Recent research focuses on combining these methods with advances in computing to enable large-scale virtual experimentation for complex systems design.

APPENDICES

APPENDIX A: GENERATED PLAIN-TEXT INSTRUCTIONS FOR MODEL-BASED REQUIREMENT IMPLEMENTATION

The following is the output of the plain-text instruction generation for the Vehicle case study (partially shown in Figure 2). The prototype tool generated these 67 steps automatically. The ordering of instructions below is changed slightly from the direct output to collect similar requirement templates, whereas the direct output is ordered automatically to collect all the actions of a particular system/subsystem.

Detailed steps to build MBR as a SysML model **(as auto-generated from the TBR-to-MBR transformation technique)**	**Informal summary using basic SE layperson's terminology**
1. Add a Block named "vehicle". 2. Add a Block named "fuel tank". 3. Add a Block named "engine". 4. Add a Block named "windshield". 5. Add a Block named "transmission". 6a. Add a part property named "fuel tank" to the "vehicle" Block. 6b. Apply the "fuel tank" classifier to the part property. 6c. Apply the AbstractRequirement stereotype to the part property. 7a. Add a part property named "engine" to the "vehicle" Block. 7b. Apply the "engine" classifier to the part property. 7c. Apply the AbstractRequirement stereotype to the part property. 8a. Add a part property named "windshield" to the "vehicle" Block. 8b. Apply the "windshield" classifier to the part property. 8c. Apply the AbstractRequirement stereotype to the part property. 9a. Add a part property named "transmission" to the "vehicle" Block. 9b. Apply the "transmission" classifier to the part property. 9c. Apply the AbstractRequirement stereotype to the part property.	Specify the main system (vehicle) and its subsystems.
10a. Add a value property named "top speed" to the "vehicle" Block. 10b. Apply the "kilometers per" unit to the value property. 10c. Add a constraint property of the appropriate type to match the "of at least" conditional. 10d. Set the value in the constraint property's constraint to "135". 10e. Add a binding connector with ends on the value property and ConstraintParameter of the constraint property. 10f. Apply the AbstractRequirement stereotype to the constraint property. 11a. Add a value property named "range" to the "vehicle" Block. 11b. Apply the "kilometers" unit to the value property. 11c. Add a constraint property of the appropriate type to match the "of at least" conditional. 11d. Set the value in the constraint property's constraint to "500". 11e. Add a binding connector with ends on the value property and ConstraintParameter of the constraint property. 11f. Apply the AbstractRequirement stereotype to the constraint property.	Specify the measures of performance (MOPs) of the vehicle, and their desired value ranges.

15a. Add a value property named "capacity" to the "fuel tank" Block. 15b. Apply the "gallons" unit to the value property. 15c. Add a constraint property of the appropriate type to match the "of at least" conditional. 15d. Set the value in the constraint property's constraint to "12". 15e. Add a binding connector with ends on the value property and ConstraintParameter of the constraint property. 15f. Apply the AbstractRequirement stereotype to the constraint property. 16a. Add a value property named "mass" to the "fuel tank" Block. 16b. Apply the "kg" unit to the value property. 16c. Add a constraint property of the appropriate type to match the "of no more than" conditional. 16d. Set the value in the constraint property's constraint to "25". 16e. Add a binding connector with ends on the value property and ConstraintParameter of the constraint property. 16f. Apply the AbstractRequirement stereotype to the constraint property.	Specify the measures of performance (MOPs) of the vehicle's subsystems, and their desired value ranges.
12a. Add a state machine to the "vehicle" Block. 12b. Make this StateMachine the ClassifierBehavior of the "vehicle" Block. 13a. Add a State named "operational" to the ClassifierBehavior StateMachine. 13b. Apply the AbstractRequirement stereotype to the "operational" State. 14a. Add a State named "off" to the ClassifierBehavior StateMachine. 14a-1. Change the State to a SubmachineState. 14a-2. Add a StateMachine somewhere appropriate to classify it. 14a-3. Add State(s) named ['off', 'fueling'] to the new StateMachine. 14a-4. Apply the AbstractRequirement stereotype to the new State(s). 14b. Apply the AbstractRequirement stereotype to the "off" State.	Specify state-based behavior which must be offered by the vehicle.
17a. Add a state machine to the "fuel tank" Block. 17b. Make this StateMachine the ClassifierBehavior of the "fuel tank" Block. 18a. Add a State named "empty" to the ClassifierBehavior StateMachine. 18b. Apply the AbstractRequirement stereotype to the "empty" State. 19a. Add a State named "filled" to the ClassifierBehavior StateMachine. 19b. Apply the AbstractRequirement stereotype to the "filled" State. 20a. Add a State named "currently fueling" to the ClassifierBehavior StateMachine. 20b. Apply the AbstractRequirement stereotype to the "currently fueling" State. 21a. Add a State named "currently filling" to the ClassifierBehavior StateMachine. 21b. Apply the AbstractRequirement stereotype to the "currently filling" State. 22a. Add a transition from State "empty" to State "currently fueling" in the ClassifierBehavior StateMachine. 22b. Apply the AbstractRequirement stereotype to the transition. 23a. Add a transition from State "currently fueling" to State "filled" in the ClassifierBehavior StateMachine. 23b. Apply the AbstractRequirement stereotype to the transition. 24a. Add a transition from State "filled" to State "currently filling" in the ClassifierBehavior StateMachine. 24b. Apply the AbstractRequirement stereotype to the transition.	Specify state-based behavior which must be offered by the vehicle's subsystems.