Formal Requirements of a Simple Turbofan Using the SpeAR Framework

ISABE-2015-20266

Aaron W. Fifarek
LinQuest Corporation
3601 Commons Blvd.
Beavercreek, OH 45431
aaron.fifarek@linquest.com

Lucas G. Wagner
Rockwell Collins
400 Collins Rd NE
Cedar Rapids, IA 52498
lgwagner@rockwellcollins.com

***Abstract*—Traditional system engineering methods focus on sequential activities of requirements specification, design, implementation, integration, verification, and validation. Often times this process proceeds with requirements that are ill formed, underspecified, inconsistent, or simply do not capture the designer's intent. The inherent flaw in this approach is that any specification or design errors found in the verification phase require rework through all steps in the process, often resulting in substantial cost and schedule overruns. In this paper we introduce a formalized requirements development framework named Specification and Analysis of Requirements (SpeAR) that is designed to aid users in developing more consistent, well-formed requirements that mathematically capture the designer's intent while adhering to a natural language look and feel. Additionally, these formally constructed requirements enable early analysis; reducing errors, reducing cost. SpeAR provides a set of formal patterns that map to English grammar commonly used in expressing system requirements. This paper will discuss the features of SpeAR and briefly discuss how to apply the framework to a simple turbofan which was used to specify and analyze requirements.**

***Keywords—* Requirements, Model Checking, K-Induction, KIND, Lustre**

I. Introduction

Software in safety-critical domains has grown exponentially in recent years and traditional verification methods have been unable to keep pace. [1] Specifically, requirements flaws discovered in traditional testing can cause both schedule delays and cost overruns.[2] Current literature estimates that in software development approximately 70% of system errors occur during the first stages of system design which includes requirement definition, architectural design, and early model development. Conversely, the percentage of overall errors discovered early in the design process is estimated to only encompass 3.5% of the errors found in the completion of the engineering process. [3] [4] With requirement generation an early step in system engineering models, enforcing a methodology to formalize and check requirements can help mitigate error propagation. [5] The use of formal methods in engineering design can aid the generation of requirements that can be traced to a mathematical basis. With such a foundation users gain the ability to analyze and validate those requirements against other defined aspects of systems throughout the development cycle. However, a limitation of widespread formal methods adoption has been the steep learning curve to teach Subject Matter Experts (SMEs) and line engineers to read and write requirements using various formal logics such as temporal and tree logics.[6] We propose a technique to enable natural language requirements to be translated in a manner that will enable access to these groups. The approach extends work where patterns identified in well-formed requirements provides the formalization basis to allow for analysis. SpeAR (Specification and Analysis of Requirements) is a framework that enables the user to create formal specifications by way of pattern identification. It also provides a set of automated analyses to reduce errors in the design and implementation phases of development. This work does not attempt to capture the entire scope of natural language but patterns specific to safety critical system requirements. In this paper, use of formal patterns will be demonstrated through a sampling of high-level turbofan engine requirements. The formalizations will be specified and the analysis techniques available will be discussed. The paper will conclude with a brief overview of the formal pattern techniques and discoveries.

II. Background

Development of today's complex systems require many different groups of subject matter experts (SMEs) to come together to develop complex systems. Therefore as the complexity of the system increases it becomes less likely that the entire system is fully understood by one individual or group. Therefore, the interactions of all components are less understood which can lead to incorrect or incomplete understanding of how the components interact. This inconsistent system view can allow emergent behavior to occur in systems. [7] Not only is system complexity a challenge but also the loss of knowledge transferred throughout the system development process. As SMEs develop requirements it is often the case that years of experience provide completion of the constraints of the system that is not captured in traditional requirement generation techniques. As others are not fully

DISTRIBUTION STATEMENT A: Approved for Public Release; Distribution Unlimited (Case Number: 88ABW-2015-2770)

aware of understood constraints, implementation often can no longer guarantee that the barrier will not be violated.

Throughout this paper, the authors will utilize a small selection of high-level requirements of a representative turbofan engine to show the formalization of the requirements to an applicable system. The overall process of developing requirements should be meant as an interactive process of discovery and description. One such technique to achieve this refinement is the use of the "Twin Peaks" paradigm that relates requirement definition to knowledge gained from other sources such as models and architectures. [8] This is also a foundational aspect of the spiral development cycle where multiple cycles encourage refinement of a design solution. This paper utilizes EngineSim (version 1.8a) on the NASA Glenn Research Center site to generate the representative model for analysis. [9] The primary need of the engine model is to determine the association between the safety constraints with those of the dynamics of the system.

Using the simulated representation of a Pratt & Whitney F100 afterburning turbofan engine (common in the F-15 Eagle and F-16 Fighting Falcon), the basis of the requirement set was derived. [10] The challenge to capture these requirements within a formal definition has proved to be a barrier to advanced analysis techniques such as the implementation of the use of model checkers and theorem provers.[6] Advances in requirement definition techniques allowed for the development of two primary approaches to formalizing requirement sets: (1) parsing existing natural language requirements or (2) encouraging SMEs to utilize a constrained natural language paradigm when they write the requirements.

In order to parse the existing natural language requirements, toolsets, such as SRI's Automatic Requirements Specification Extraction from Natural Language (ARSENAL), were developed [11]. These tools require the ability to associate the natural language to a database of terminology that carries similar usage inside the text to decipher the linguistic meaning. Once the natural language is matched to the evolved terminology database, then a mathematical basis can be leveraged to the requirement set. Implementing this translation provides the necessary information to the system to generate provably correct analysis of the existing set.

The other methodology, although adding responsibility to the system engineers and SMEs, encourages use of a constrained natural language that will more closely translate into the mathematical foundations necessary for analysis. It is this methodology that SpeAR leverages in order to bridge the gap in the natural language to mathematical expression.

III. TurboFan Example

In order to best describe how SpeAR utilizes a constrained language to build requirements, the use of a turbofan example was chosen. Examination of this example provides a number of overall properties of the engine to be defined at the system level. These properties show overarching high-level requirements of the system that must be maintained. Often these represent the safety properties of a system at each component level. The component requirements at each level defines the behavior of the component. Therefore, each component must have agreement in its defined behavior and its properties. Looking specifically at the system properties that define a relationship between thrust and fuel flow rate, examples of the analysis capabilities can be demonstrated. Utilizing NASA EngineSim, some representative requirements are considered to demonstrate these cases and are simplified to ease understanding. These categories will be introduced without the constrained language that is used in SpeAR so to build the general idea of these requirements.

A. Direct Property Conflicts

Engine Property 1: *As the fuel flow increases, the thrust of the engine should also increase.*

Engine Property 2: *As the fuel flow decreases, the thrust of the engine will also decrease.*

It can be seen that these engine properties are simplified for clarity of the example. This is a violation that can be found where the overarching behaviors of a component or system cannot coexist. Looking back at the system level property that defines a relationship between the fuel rate and the engine thrust, the relationship defines in proportional increase/decrease in both as the expected reaction. If another property is defined that would contradict this relationship, the system will create a counterexample that shows one of the two properties cannot possibly exist based on the defined behavior. Although trivial in this example, as systems become more complex, this type of conflict commonly emerges.

B. Requirement Conflict / Dissatisfaction

Engine Property 3: *The engine must have a pump which will provide fuel to the combustion chamber.*

Pump Requirement 1: *While the engine is on, the fuel pump shall supply the engine with a fuel flow proportional to that of the desired throttle.*

Pump Requirement 2: *While the engine is on, the fuel pump shall supply the engine with a fuel flow independent of the throttle.*

The engine will contain a sub-component to manage the fuel flow into the turbofan combustion chamber. Assuming that this is directly managed by a fuel pump, we define the fuel pump component. This component utilizes throttle settings to manage the fuel rate. The second pump requirement violates that relationship by stressing that the fuel flow and throttle settings are independent forcing a direct conflict in behavior of the pump component in pump requirement 1 and 2.

This is one of the most common analysis results found utilizing this framework. Assume that the system level property defined an inverse relation of fuel flow to that of engine thrust. As the subcomponent fuel pump provides the system a proportional change (fuel use increase implies an increase in thrust), then a counterexample would be generated that shows that these cannot as prove independent responses in the engine behavior. This counterexample would specify that a fuel rate increase is not able to create a condition with less thrust according to the behavior defined in the fuel pump component.

DISTRIBUTION STATEMENT A: Approved for Public
Release; Distribution Unlimited (Case Number: 88ABW-2015-2770)

C. Under Specification

Engine Property 1: *The engine shall be provided fuel at a rate of no greater than 24,000 lb/hr. while at 0ft altitude.*
Engine Property 2: *When the engine afterburner is engaged the fuel flow rate shall be 67,000 lb/hr while at 0ft altitude.*

This is an example of an under specification conflict as a result of how the system properties were defined. In the first property, the lack of additional constraints of the requirement resulted in a design parameter of a max fuel flow that was insufficient should the afterburner be activated at the same conditions. Therefore, analysis of these engine properties would result in a failure if the resulting requirement set only utilized the lower flow rate.

This analysis is a common weakness in natural language requirements due to the ease for incomplete scope definition. For example, the system level property defines a safety check of the increased fuel flow rate that translates to some increase in thrust of the engine. In the physical world, this specification may be fully defined. However, in the cyber realm, this is an incomplete specification. There is no description of what is expected when the fuel flow rate is decreasing or is the same. Model checkers (as would software) exploit this since the behavior is not fully constrained. The behavior can be anything thereby opening the ability of a system to develop unconstrained or emergent behavior.

IV. SpeAR

The backbone of the SpeAR framework is a set of precise mathematical notations used to represent requirements, inspired by sets of formal property specification patterns developed at Kansas State University by Dwyer et al. [13]. These specification patterns were chosen because they captured a wide variety of temporal behaviors, were published and peer reviewed for accuracy, and are easily translated to the languages of various formal analysis tools such as the Kind and NuSMV model checkers. In addition, the patterns provide an abstraction; users are not required to understand the logical basis behind the formalisms allowing novices to use SpeAR with minimal training. This pattern set was further investigated and expanded upon by Castillos et al. in 2013 whose work sought to provide pattern definitions through the use of finite automata. [14]

Each requirement is implemented in SpeAR as a scope and predicate scope pairing. The scope defines the temporal relationship of when the defined variables should hold, while the predicate defines the expected state of the variables. In Figure 1, an example natural language requirement could specify an association between the engine state and the rotational shaft of the engine. The scope defines the understanding of an attribute of the engine that the system engineer sees while the engine is on. Since this requirement was formulated with an identified pattern, translation into the direct pattern and the mathematical representation of the statement can be achieved.

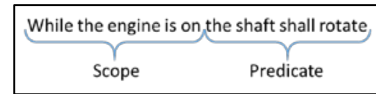

Figure 1: Scope and predicate example of a patterned requirement

```
while Engine_On :: always Shaft_Speed_RPM > 0.0;
```

Figure 2: SpeAR pattern formulation of the example in Figure 1

A. Scope and Predicate Relationship

In Dwyer's original work, the team identified five specific scopes that the majority of the specification patterns are based upon. These original scopes include Global, Before Q, After Q, After Q until R, and Between Q and R. In Figure 3, these scopes are defined graphically upon a timeline representing possible events. The scopes are driven by the identification of trigger events in temporal relationship with other events of interest

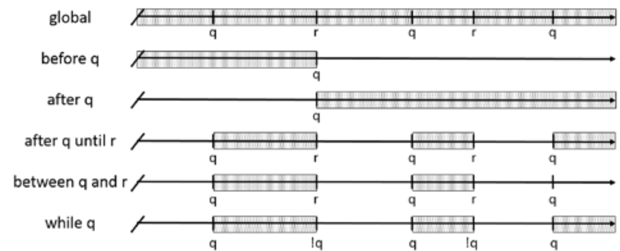

Figure 3: Patterns identified by Dwyer et al.

The Global scope is utilized in patterns where a predicate condition of a variable must be held valid for the entire lifespan of the system. The Before Q and After Q scopes are closely related in that they both utilize a relation to the trigger event to hold validity. The scopes After Q until R and Between Q and R are often confusing because in many situations they could appear to be equivalent. The difference is in the acceptance of the scope constraint range. With After Q until R, the scope is in the acceptance state after Q has occurred. Due to the unbounded natural of temporal logic, the scope guarantees that R will occur at some time in the future. Therefore, should the R event not occur until infinitely in the temporal future, then the scope is still valid. Between Q and R, this is not the case. For this scope to be valid, there must exist a Q that precedes the occurrence of an R event. Should the response of the R event not be defined prior to the termination of the analysis, then the scope was never determined to be valid and, therefore, its predicate specification would not be held.

The capability of these scopes to map directly to the temporal constraints provides the foundation of the mathematical rigor of this methodology. For example, the Global scope pattern maps directly to the global (□) temporal logic equivalency. Furthermore, we utilized automata in a similar fashion as Castillos et al. [14] to define the patterns utilizing these scopes. This will be discussed in greater detail in Section IV.B.

Upon examination of the turbofan, it can be seen that a number of the logical requirement sets are ideally suited for the pattern scope/predicate relationship. Requirements that meet

this capability are said to be "formalizable" meaning that the structure of the requirement itself defines the necessary information needed to develop a formal representation to analyze. Such a requirement is demonstrated with Figure 1. Since these requirements often well-define the functional behavior of systems, they are also known as functional requirements. In most all requirement sets, the existence of requirements that do not have a formal basis also exist in the set. Such requirements, known as "non-formalizable" requirements, are not readily translatable into a formal semantic. Often these requirements exist when system engineers add architectural information to the requirement set (e.g., "The system shall be quad-redundant"). Examples of formalized requirements with specific scope:

Table 1: Example formalized requirements pertaining to a turbofan engine

The rotation of the turbofan axle **will always be in the clockwise direction.**
The temperature of the combustion chamber will not exceed 1500 degrees Fahrenheit.
As the fuel flow increases, the thrust of the engine should also increase.
Prior to **the startup of the system, the engine shall be unpowered in the OFF state.**

B. Patterns

The patterned specifications (Table 1) demonstrates further examples on how natural language requirements with a formal basis can be classified into mathematical patterns. The SpeAR framework supports the patterns associated with absence/always, existence, precedence, and response from the original specification patterns (see Dwyer et al. [13]) and further supported by Castillos et al. [14] work. It also introduces new and derived patterns to capture requirements that were found to be useful including initial, delta, range, and invariant.

By utilizing patterns to generate a formal mathematical basis of the requirements, it is now possible to utilize model checking tools to examine if the specifications can properly exist in a defined system. Examining the requirements defined in Table 1, example SpeAR formulations can be completed as shown in Table 2. In order to properly define these requirements further definition of the distinctive varaibles of the pattern must also be defined. For example, in the first formulation which denotes the direction of the axel spin direction, the variable *axle_direction* would need to be defined prior its use here to represent an axle. The other requirements written here would need definition for associated variables that would allow inspection and evaluation. Furthermore, it is known for the third requirement that constant altitude, inlet velocity, and pressure to state a few additional constraints that would have to exist in the system for the fuel flow and engine thrust relationship to hold. This is simplified for the examples in this piece.

Table 2: SpeAR pattern equivalent requirements of those defined in Table 1

global::**always** axle_direction == Direction.Clockwise;
global::**always** combustion_chamber_temp <= 1500 F
while pre_flow_rate < flow_rate :: **always** pre_thrust_value < thrust_value
initial :: engine_state == OFF **and** power_level == 0;

V. Analysis

SpeAR provides two types of specification analyses: language level checking to ensure consistency and completeness of a specification and formal analysis for validation of requirements. As the user writes a specification, checks are performed to guide the user into making complete, well-formed specifications. Many of these checks, such as type-checking, are trivial. However, they provide quick and easy feedback to the user; reducing the errors that migrate to later stages of development. This analysis is conducted with the use of model checking tools and techniques that take advantage of computer science graph theory and the capability to quickly explore problem spaces.

A. Model Checking

In order to utilize the capabilities of model checkers, it is necessary to have a basic understanding of how this class of tools can efficiently explore problem state spaces. At its most fundamental understanding, these tools take advantage of the computer processor's ability to quickly calculate all possible paths from that of a given point or known state of the modeled system. For this given point there exists algorithms to test combinations of variables of the system to verify claimed requirement behavior. It is the initial known state that allows for the brute-force manner to propagate the state exploration forward [15]. The tool is able to propagate the decisions of the state forward using a defined model of the system in question. Illustrated in Figure 4, the system is compared to the requirements that should properly bound the model space. If the system and the requirements are proven to properly constrain the space, the tool returns a satisfied state; otherwise, a counter-example is generated that demonstrates how the inconsistency was achieved. It is critical to note that a counter-example only specifies that there is a disagreement between the definition of the formal specifications and the representative model of the system itself. It does not specifically demonstrate where the fault lies. What this tool does demonstrates is an inconsistency of the design of branches that feed the model-checking engine block. For example, the counter-example may demonstrate that a value could go negative. Was this behavior expected in the model? If not, then that shows that the model may be flawed; otherwise, should a negative value be valid? If so, then it points to a lack of a proper constraint provided by the requirement definition. Therefore, model checking tools provide the capability to point out inconsistencies for the user to examine and not an automated system that provide solutions without human reasoning.

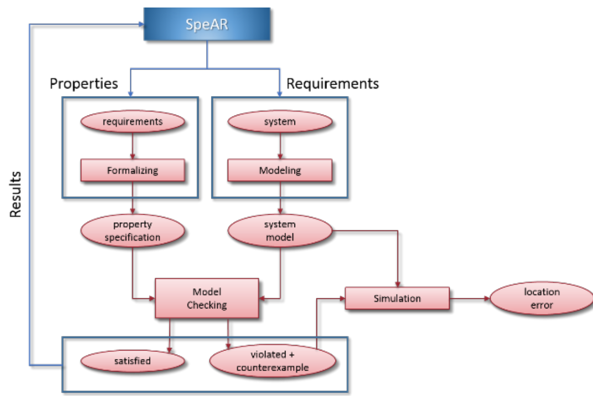

Figure 4: Schematic view of the model checking approach (red) [15] with SpeAR attachment into the process (blue)

1) Compositional Verification

Even though current model checkers have been shown to be powerful on certain classes of problems, they do come with limitations such as that of state-space explosion [16]. This challenge is common among complex systems due to the inability to feasibly traverse the possible bounded model space within a reasonable computation time. Systems that suffer from state-space explosion often cannot isolate the atomic nature of choices in parse trees and thus eliminates the ability to vary the system along an isolated degree of freedom to explore system setting combinations. Several techniques exist to mitigate state-space explosion in system design verification. One such method is the use of compositional verification which abstracts subcomponents as black boxes, disregarding the design details in favor of contracts that describe the relevant behaviors of the component [17]. This concept stresses that complex systems are made of more simplistic components that are easier to verify, thus reducing the monolithic expanse of variables in the systems. Then, once these subcomponents are verified, they can be treated as an atomic piece (the black box) higher in the system. Naturally, the challenge then becomes how to ensure that the relationship between the components and the higher level composite system is well understood and valid. Although also not a trivial problem, there has been a number of advances in architecture to leverage model checking to help solve these interaction problems [18].

In the turbofan example, compositional design and verification is illustrated with the relationship of the of an engine level output of overall thrust and fuel flow. In order for the fuel flow to be maintained in the system in a controlled manner a pump must supply and regulate the flow. Therefore, requirements on how responsive changes in the thrust are to the fuel flow in turn provide requirements that the pump must be able to achieve.

B. Formal Analysis

In our analysis, we define properties that represent the high-level specifications of the component. The requirements define the derived model that should satisfy the properties and disambiguates how the system will respond to a given input. This analysis is completed in SpeAR by translating the specification to a Lustre model. [19] In this model, all variables are represented as nondeterministic, varying inputs. Then, the requirements restrict the values that these variables are allowed to take on. This set of traces is then checked against each property. If a property does not accept a trace, one or more requirements allow an unintended behavior or the property itself is incorrect. This analysis is done by creating an obligation. In this obligation, the conjunction of all the requirements (R_j) and assumptions (A_i) must imply the properties (P_k) of interest. This is captured in Equation (1).

$$\bigwedge_{i=1}^{m} A_i \wedge \bigwedge_{j=1}^{n} R_j \Longrightarrow \bigwedge_{k=1}^{q} P_k \quad (1)$$

VI. Defining a System using the SpeAR Framework

The SpeAR Framework is developed using the Eclipse IDE with the XText markup extension [20]. This allows for the user to be able to define the system compositionally with the use of the Relation and Definition files.

A. SpeAR Relation File

The Relation file (Figure 6) is the location where all the relation information of the system or component is captured. The file is constructed with three parts; (1) the heading block, (2) the input/output (I/O) and local variable block, and (3) the component Definitions/requirements block. The heading block (area 1 in Figure 6) has a Relation label (only a single entry) that defines how the file will be referenced in the overall project and a Uses section that lists all the external files that have information needed in this file. The Uses label can be utilized as many times as necessary in order fully allow access to other external files. This is commonly used to incorporate the contents of Definition files into the Relation file. It is also used to ensure that the different data types that are specified in other files may also be used here. The next block in the Relation file is the I/O and local variable block (area 2 in Figure 6). One of the constructs necessary in the SpeAR file format is the premise that components have known inputs and outputs for the system. This represents the first two parts of this block; Inputs define the information that is input into the component, Outputs define the information that the component will send out. Ultimately, these labels define the input and output of a “clear box” representation of system. The State section provides an area for the system engineer to define variables that will be local to the component (i.e., not provided by inputs). The Macros section provides the ability to create shorthand calls to complex conditions and/or calculations (not pictured). The last block in a Relation file is the component definition/requirements section (area 3 in Figure 6). The first section in this block is the Requirements which defines the component level requirements (also known as the derived requirements). This section defines the behavior of the component that will be tested by the model checker. The last section in this block is the Properties which define the conditions that must be held for the overarching component. Also known as high-level requirements, this section typically describes safety and/or security constraints that the component must ensure are maintained.

DISTRIBUTION STATEMENT A: Approved for Public Release; Distribution Unlimited (Case Number: 88ABW-2015-2770)

```
Relation Ctrl_Tanks.HoldingTankCtrl;

Uses Ctrl_Tanks.definitions.*;
Uses Ctrl_Tanks.definitions.holding_tank_ctrl_output_type.*;

Inputs:
    sensor_high : bool;
    sensor_low : bool;

Outputs:
    output : holding_tank_ctrl_output_type;

State:
    pump_state : bool;
    valve_state : bool;

Requirements:
    a01 = global :: always sensor_high implies sensor_low;
    a02 = global :: always not sensor_low implies not sensor

    init = initial :: pump_state and not valve_state;
    r00 = while sensor_high :: always not pump_state and val
    r01 = while not sensor_low :: always pump_state and not
    r02 = while not sensor_high and sensor_low :: always (pu

    r_output = global :: always output == new holding_tank_c

Properties:
    p01 = global :: always not (pump_state and valve_state);
```

Figure 5: Representation of a SpeAR Relation File

B. *SpeAR Definition File*

It is important to note that parts of the Definition file can also be included in the Relation file if desired. Often it is better to have a specific Definition file especially in compositional designs of a system. In order for many specifications to be properly verified, there is the need to have a unified understanding of the units, constants, datatypes, etc. that are used across multiple files. In Figure 7, there are three blocks of the representative SpeAR Definition file; (1) the file unique name, (2) the agreed upon units of measure for the system, and (3) the constants and datatype definitions. The first section (area 1 of Figure 7) gives the unique name to the Definition label. It is good practice to use the example naming convention as seen in the Figure 7 where the project name and further description of the Definition file is listed. The second section (area 2 of Figure 7) explicitly defines how Units will be related in this system. This is an important aspect of many requirement sets due to the changing between understood units have caused a number of accidents including the loss of the NASA Mars Climate Orbiter in 1999 [21]. The last block (area 3 in Figure 21) defines the Constants and Types sections of the file. The Constants allow the user to define the understood constants of the system in a single location that can be brought into Relation files. The Types section defines datatypes that the user can define for the system and incorporate into Relation files as well.

```
Definition Ctrl_Tanks.definitions;

Units:
    m;                  // meter
    s;                  // second
    m3 : m * m * m;     // meters cubed
    m3_p_s : m3 / s;    // cubic meters per second

Constants:
    HOLDING_TANK_SENSOR_HIGH : real = 5.0;
    HOLDING_TANK_SENSOR_LOW : real  = 3.0 ;

    COOLING_TANK_SENSOR_HIGH : real  = 3.0 ;
    COOLING_TANK_SENSOR_MID : real  = 2.0 ;
    COOLING_TANK_SENSOR_LOW : real  = 1.0 ;

Types:
    system_output_type : {production_flow : real , emergency_flow : real};

    cooling_tank_output_type : {production_flow : real , emergency_flow :
    cooling_tank_ctrl_output_type : {valve_p_state_cmd : bool, valve_e_sta

    operation_button_type : [STANDBY, OPERATE, MAINTENANCE];
    operation_modes_type : [om_OFF, om_STANDBY, om_STARTUP, om_OPERATE, om
```

Figure 6: Representation of a SpeAR Definition File

VII. Conclusion

In this paper, we introduced the SpeAR framework for developing a formalized set of requirements. It provides a set of formal semantics for expressing desired system behaviors, a domain specific language to ensure consistency and coherency, and formal analysis capabilities to help the user verify and validate the specified behaviors are correct. The authors plan to continue this work by enhancing the SpeAR framework to allow for increased expressibility of requirements through patterns that users can develop. Furthermore, the authors plan to extend the capability here into Assume/Guarantee systems which look to leverage high-level and derived requirements to define compositional interactions which will lead to "system of systems" concepts to be used in modern system development. In addition, there are plans to investigate techniques to generate testing artifacts from formalized requirement specifications that would aid in the future certification process of systems.

VIII. References

[1] G. Van Oss, *Avionics Acquisition, Production, and Sustainment: Lessons Learned -- The Hard Way,* DTIC, 2002.

[2] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer,* pp. 135-137, January 2001.

[3] D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson/Addison-Wesley, 2004.

[4] NIST, "The Economic Impacts of Inadequate Infrastructure for Software Testing," 2002.

[5] M. Dorfman, "System and Software Requirements Engineering," in *IEEE Computer Society Press Tutorial*, 1990.

[6] J. A. Davis, M. Clark, D. Cofer, A. Fifarek, J. Hinchman, J. Hoffman, B. Hulbert, S. P. Miller and L. Wagner, "Study on the Barriers to the Industrial Adoption of Formal Methods," in *18th International Workshop on Formal Methods for Industrial Critical Systems*, Madrid, Spain, 2013.

[7] H. V. D. Parunak and R. S. VanderBok, "Parunak, H. Van Dyke, and Raymond S. VanderBok. "Managing emergent behavior in distributed control systems.," in *ISA-Tech*, Ann Arbor, 1997.

[8] J. Cleland-Huang, R. S. Hanmer, S. Supakkul and M. Mirakhorli, "The twin peaks of requirements and architecture," Software, IEEE 30.2, 2013.

[9] NASA Glenn Research Center, "EngineSim Version 1.8a," NASA, 2013. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/ngnsim.html. [Accessed 13 5 2015].

[10] United Technologies Corporation – Pratt & Whitney Division, "F100 Engine," 2015. [Online]. Available: http://www.pw.utc.com/F100_Engine. [Accessed 14 5 2015].

[11] S. Ghosh, N. Shankar, P. Lincoln, D. Elenius, W. Li and W. Steiener, "Automatic Requirements Specification

Extraction from Natural Language (ARSENAL)," DTIC, 2014.

[12] M. Heimdahl and N. Leveson, "Completeness and consistency in hierarchical state-based requirements," *IEEE Transactions on Software Engineering,* vol. 22, no. 6, pp. 363-377, 1996.

[13] M. B. Dwyer, G. S. Avrunin and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference*, 1999.

[14] K. C. Castillos, F. Dadeau, J. Julliand, B. Kanso and S. Taha, "A Compositional Automata-based Semantics for Property Patterns," in *In Integrated Formal Methods*, 2013.

[15] C. Baier and J.-P. Katoen, Principles of model checking, Cambridge, MA: MIT Press, 2008.

[16] E. M. Clarke, W. Klieber, M. Novacek and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification*, Springer Berlin Heidelberg, 2010, pp. 1-30.

[17] H. Garavel and F. Lang, "SVL: a scripting language for compositional verification," in *Formal Techniques for Networked and Distributed Systems*, 2002.

[18] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan and M. P. Heimdahl, ""Your "What" is my "How": Iteration and hierarchy in system design," *IEEE Software,* vol. 30, no. 2, pp. 54-60, 2013.

[19] N. Halbwachs, The synchronous data-flow language Lustre.

[20] The Eclipse Foundation, "Eclipse," The Eclipse Foundation, 2015. [Online]. Available: http://www.eclipse.org/. [Accessed 17 March 2015].

[21] R. Lloyd, "Metric mishap caused loss of NASA orbiter," CNN.com, 30 September 1999. [Online]. Available: http://www.cnn.com/TECH/space/9909/30/mars.metric.02/. [Accessed 17 March 2015].

[22] E. Vassev and M. Hinchey, "Autonomy Requirements Engineering," *Computer,* vol. 46, no. 8, August 2013.

DISTRIBUTION STATEMENT A: Approved for Public Release; Distribution Unlimited (Case Number: 88ABW-2015-2770)