Helping System Engineers Bridge the Peaks

Neha Rungta[1], Oksana Tkachuk[1]
Suzette Person[2]
[1]NASA Ames Research Center, CA, USA
[2]NASA Langley Research Center, VA, USA

Jason Biatek, Michael W. Whalen
Dept. of Computer Sci.
University of Minnesota, USA

Joseph Castle, Karen Gundy-Burlet
NASA Ames Research Center, CA, USA

ABSTRACT

In our experience at NASA, system engineers generally follow the Twin Peaks approach when developing safety-critical systems. However, iterations between the peaks require considerable manual, and in some cases duplicate, effort. A significant part of the manual effort stems from the fact that requirements are written in English natural language rather than a formal notation. In this work, we propose an approach that enables system engineers to leverage formal requirements and automated test generation to streamline iterations, effectively "bridging the peaks". The key to the approach is a formal language notation that a) system engineers are comfortable with, b) is supported by a family of automated V&V tools, and c) is semantically rich enough to describe the requirements of interest. We believe the combination of formalizing requirements and providing tool support to automate the iterations will lead to a more efficient Twin Peaks implementation at NASA.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, tools*

General Terms

Algorithms, Languages, Verification

Keywords

Requirements, Model-Based Development, Formalization

1. INTRODUCTION

The essence of the Twin Peaks model is concurrent development of software requirements and architecture design in an iterative manner [13, 17], interleaving the processes to achieve architectural stability in the face of inevitable requirements changes. The original Twin Peaks model prescribes the "what" but provides little guidance on "how" to achieve this. Many solutions have been designed with software engineers in mind, rather than system engineers.

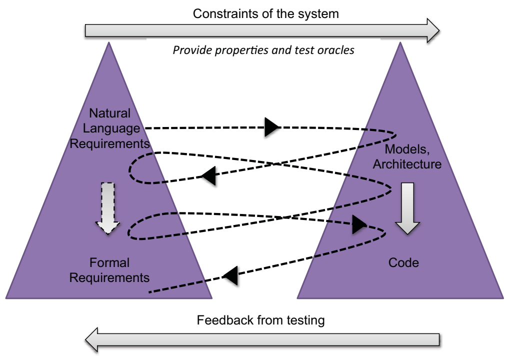
Constraints of the system
Provide properties and test oracles
Natural Language Requirements
Models, Architecture
Formal Requirements
Code
Feedback from testing

Figure 1: Iterative System Development

At NASA, the system development process for safety- and mission-critical systems often proceeds iteratively following the Twin Peaks model. System engineers start by specifying requirements in English natural language, and then use the requirements to derive the constraints on the system. They design the architecture in parallel by building the models (e.g., in Simulink), and auto-generating portions of the code. Each step of the process includes a variety of verification and validation (V&V) activities at different levels of abstraction. The bridges between the twin peaks are the constraints derived from the natural language requirements and the feedback from the V&V activities as shown in Fig. 1. Currently, each iteration involves a considerable amount of manual and duplicated effort in the testing and verification of the models and the code. For example, tests are written manually to test the model, and again to test the code auto-generated from the model with little re-use. The main reasons for the manual effort stems from the fact that (i) requirements are written in natural language, rather than in a formal notation, and (ii) lack of appropriate tool support for automated testing and verification.

Several papers have proposed formal requirements as a means to bridge the gap between the twin peaks [6, 12]. In the academic literature, several formal notations with varying degrees of expressiveness have been developed to facilitate the formalization of requirements, e.g., LTL and CTL [4], Z [16], and VDM [8]. They are often supported by techniques that can leverage formalized requirements as properties and test oracles, e.g., unit-level testing, symbolic execution [5, 10], and model checking [4]. These notations and tools to formalize requirements and the V&V techniques that use them were developed for formal methods experts and, to a lesser extent, software engineers. System engineers, however, face a steep learning curve with these formalisms and tools, and struggle to adopt them in practice.

In this paper, we propose a Twin Peaks model for developing safety and mission-critical systems that is based on formalized system requirements specified in Simulink—a notation that is semantically rich and currently used by NASA system engineers. Our approach proposes automated support for translating existing natural language requirements into Simulink models, effectively creating a bridge for system engineers from the requirements peak to the architecture peak through the properties and test oracles generated from the requirements as shown in Fig. 1 by the left-to-right arrows. The generated properties and oracles are then used as input to V&V tools such as Reactis[1] and KLEE [3] to automatically generate tests. The results from the tests, e.g., the set of failing test cases, provide a bridge from the architecture peak back to the requirement peak as shown by the right-to-left arrows in Fig. 1, and are used by the engineers to refine the requirements.

2. BACKGROUND

Model-Based Development (MBD) refers to the use of domain-specific modeling notations that can be analyzed for desired behavior before a digital or control system is built. The use of such modeling languages allows a system engineer to create a model of the desired system early in the lifecycle that can be executed on the desktop, analyzed for desired behaviors, and then used to automatically generate code and test cases. Also known as correct-by-construction development, the emphasis in MBD is to focus the engineering effort on activities scheduled early in the development lifecycle such as modeling, simulation, and analysis. The goal is to automate the activities later in the development life-cycle such as coding and testing. This can reduce development costs by finding defects early in the lifecycle, avoid costly fixes when errors are discovered during integration testing, and automate coding and creation of test cases. In this way, model-based development significantly reduces costs while also improving quality. There are several commercial MBD tools, including SCADE [7], IBM Rhapsody [14] and iLogix StateMATE [15].

In this paper, we focus on Simulink [1] and Stateflow [2] from Mathworks, Inc., as these are the MBD tools most often used by NASA system engineers at our site. Simulink is a data flow graphical language as well as a tool for modeling and simulating dynamic systems (both the language and the tool are generally referred to as Simulink). Stateflow is a state-based notation similar to David Harel's Statecharts notation [9] (again, Stateflow also refers to the tool). Both Simulink and Stateflow are tightly integrated in the MATLAB environment and can refer to other languages available in the environment. In Section 5, we present a simple microwave example with Figures 3 and 4 making use of Stateflow and Simulink notations respectively.

[1]http://www.reactive-systems.com/

3. MOTIVATION

Successful missions such as LADEE, XSS-10, and XSS-11, among others, espouse the following principles: (i) design software in a manner that lends itself to be testable, (ii) use a model-based development environment that can facilitate rapid prototyping of algorithms based on the requirements, (iii) autocode the models to reduce transcription errors, and (iv) use extensive automation in the V&V processes[2]. We believe these principles are consistent with the philosophy behind the Twin Peaks model.

Several missions at NASA, including LADEE, follow a model-based development process where the flight software (FSW), the vehicle, and the environment are modeled in Simulink. Simulink provides an integrated development environment with various custom libraries and requisite support for modeling dynamic control systems. It allows developers to simulate the model in order to observe its behavior. The code (software) is auto-generated from the Simulink models and then integrated with legacy code (to facilitate reuse) and other hand-written code. There are three different types of tests used to validate the models and code: (i) workstation simulation, which consists of testing the Simulink models and occurs early in the development process to facilitate algorithm development and analysis of the requirements, (ii) processor-in-the-loop simulation, which consists of testing the auto-coded modules on processors that mimic flight-like processors, and (iii) hardware-in-the-Loop Simulation, which consists of testing the FSW on the actual flight avionics.

In the LADEE mission the design, requirements, functional tests, and Simulink models were prototyped in parallel in the workstation simulation environment. A unit test suite was manually generated to verify low-level requirements, and integration test suites were used to verify system/sub-system compliance with the associated requirements. The LADEE engineers manually specified the test oracles for the unit and system level tests by computing the expected outcomes for the specified inputs. The requirements for the guidance, navigation & control module in LADEE were formalized as assertions and Matlab expressions manually. While the LADEE engineers put in a considerable effort into automating the V&V processes in terms of executing tests and collecting data, they would like to further reduce the manual effort by automatically generating tests. Note that this is one instance of how V&V activities are performed missions at NASA; missions may often have varied V&V processes. This work is a step towards that goal. In this paper, we describe a process that embodies some of the practices used by the LADEE engineers in order to define a generic process, based on the Twin Peaks model, that can be applied across various NASA missions and other projects that develop safety-critical systems.

[2]Lunar Atmosphere and Dust Environment Explorer (LADEE) is a NASA mission orbiting the Moon. Its main objective is to characterize the atmosphere and lunar dust environment. Whereas, XSS-10 and XSS-11 are micro-spacecraft developed by the U.S. Air Force Research Laboratory.

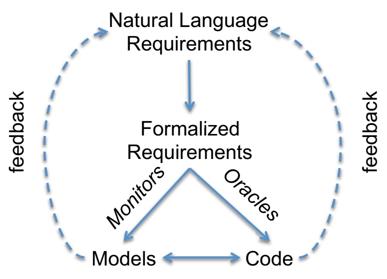

Figure 2: Overview of our approach

4. APPROACH

In Fig. 2 we present an overview of our Twin Peaks approach that leverages formalized requirements to facilitate development of safety-critical software. Fig. 2 describes the key artifacts and their relationships, and illustrates how formalized requirements can assist in bridging the gap between requirements and architecture. The three main artifacts in model-based development at NASA are: (a) natural language requirements, (b) models, and (c) code. At the core of our approach are the formalized requirements that are used to generate testing artifacts, e.g, monitors for testing models and oracles for testing code, as shown in Fig. 2. The results of testing the model and code provide feedback that is used by the system engineers to refine the requirements, and the process repeats. In the remainder of this section we describe the key components of our approach and the tools we propose to use to automate the various processes.

4.1 Formalizing Requirements

We propose that requirements for safety-critical systems be formalized using a modeling language such as Simulink. Since system engineers are often familiar with model-based development environments, using a language like Simulink to formalize requirements has the advantage of reducing, or even eliminating, the need to learn a special purpose property specification language, along with the requisite costs in time and training to become conversant in it.

To gauge the expressiveness of Simulink as a requirements formalization notation, we performed an exploratory study at NASA Ames, in which we analyzed requirements from the LADEE mission. The goal of our study was to estimate the percentage of the natural language requirements that could be formalized using Simulink. The results of our study indicate that approximately 80% of the LADEE requirements can be formalized using Simulink. The other 20% of the requirements were either (a) missing some domain information in the description of the requirement, or (b) were not formalizable. Requirements that were not formalizable were actually high-level goals written as requirements, e.g., the spacecraft should fly. From this informal study we concluded that Simulink is sufficiently expressive to encode the kinds of requirements used in the LADEE project.

Considerable effort is required to transform natural language requirements into a formal representation, even with a formalism that is intuitive for system engineers. The system engineers at NASA are often under pressure to meet deadlines that are driven by launch windows. Adding a new task to the development process, such as formalizing the requirements, creates an additional task for the already time-constrained system engineers. If, however, we can provide additional tool support that automates part of the tedious formalization process, we believe more system engineers would be willing to invest their time in developing formal requirements. Especially since the formalized requirements have the potential to conserve resources by reducing the manual effort necessary to perform the V&V processes.

Recent work in natural language processing of requirements (e.g., TRAM [11]) shows promise as a technique to support automated requirements formalization from natural language requirements. In our discussions with several engineers working in the domain of safety-critical systems, we were able to infer that a large number of natural language requirements for a particular domain or application follow a small number of patterns specific to the corresponding domain or application. We see this as an opportunity to be leveraged by designing templates that can be used by tools such as TRAM to assist in automatically formalizing requirements.

4.2 V&V with Formalized Requirements

Once requirements are formalized, they can be used to reduce the manual effort necessary in the current V&V processes. Currently, system engineers manually write each test as two different parts: (a) the inputs to the system, and (b) the expected outcome of executing the test on the specified inputs. The expected outcome of the test is what enables testers to determine whether the system has any errors or not. This is a manual and tedious process which takes a considerable amount of time and resources.

Automated test case generation techniques, e.g., random test generation or test case generation using symbolic execution, only provide the first part of the test cases: the inputs to the system. The user is generally required to reason about the expected outcome of the tests. This is an especially challenging task when the automated test case generation tool generates hundreds or thousands of test cases—which often occurs in practice. Formalized requirements can be used in conjunction with the automated test case generation techniques to overcome this challenge.

We have experimented with two different test case generation technologies combined with formalized requirements. The first one is Reactis, a tool for generating test cases on Simulink models using a directed random search. The formalized requirements are used as runtime monitors in Reactis, which attempts to find tests for the various "paths" through the requirements. The runtime monitors report a failure if an observed behavior violates the condition in the requirements. Another tool, KLEE [3], uses symbolic execution to generate test cases for C code. The formalized requirements are specified in Simulink, from which the test oracles are generated by auto-generating C code from the Simulink models.

Formalized requirements provide system engineers a straightforward way to measure coverage of requirements. Many safety-critical domains, such as aerospace applications, have certification demands, e.g., that each requirement is covered by at least one test. Automated test cases and formalized requirements can be used to determine which requirements are covered by the generated tests. The information about failing requirements can be used by developers to refine the model or the requirement.

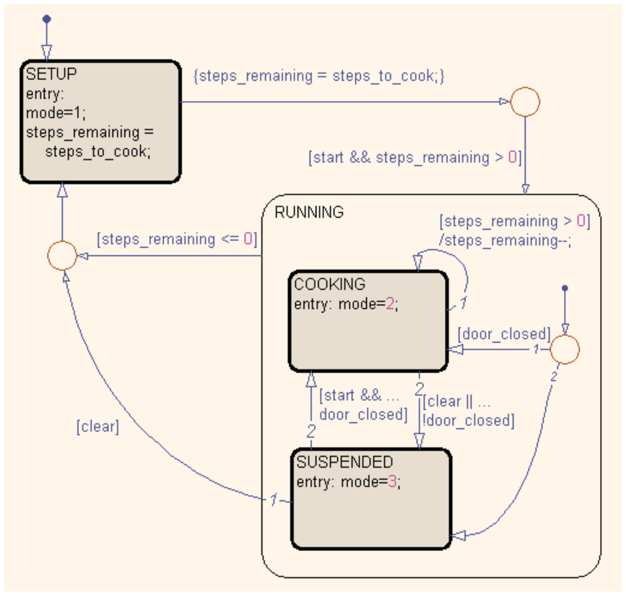

Figure 3: Simple microwave mode controller

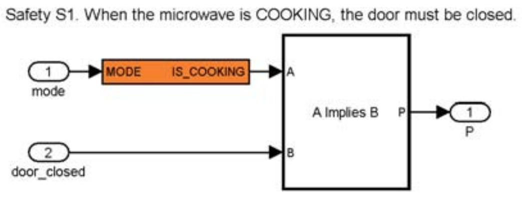
Safety S1. When the microwave is COOKING, the door must be closed.
1
mode
MODE IS_COOKING
A
A Implies B
P
1
P
2
door_closed
B

Figure 4: A safety requirement formalized in Simulink.

5. EXAMPLE APPLICATION

In this section we present a microwave oven controller as a proof-of-concept for our proposed approach. Note that in the example we manually create the formalized requirements; developing techniques to automate formalized requirements generation is ongoing work.

The controller consists of two parts: a mode controller, which controls the cooking element of the microwave, and a display panel controller, which manages the display. We describe our approach on the mode controller, shown in Fig. 3, using the Stateflow notation. Stateflow consists of state machines that can be organized hierachically and in parallel. The example in Fig. 3 demonstrates hierarchical composition: the *COOKING* and *SUSPENDED* states are contained within the *RUNNING* state. Transitions between states have Boolean *guards* that describe the conditions under which a transition can occur (indicated by expressions inside of braces: []) as well as *actions* that describe changes to variables that occur when the transition occurs. Transitions have *priorities*: if multiple transitions could cause a state transition from the currently occupied state, the enabled transition with the lowest priority number (e.g., the 1's and 2's on Fig. 3) is chosen.

The controller in Fig. 3 starts in the SETUP mode, and transitions to the RUNNING mode when the start button is pressed (if the seconds_to_cook parameter is > 0). In the RUNNING mode, the microwave can either be in COOKING or in SUSPENDED mode. If the microwave door is opened when in RUNNING mode, or the clear button is pressed, then the microwave mode switches to SUSPENDED, where the user can either press the start button to go back to COOKING, or press the clear button again to go back to SETUP.

The microwave has a number of safety and functional requirements expressed in English. One of the safety requirements is: *When the microwave is COOKING, the door must be closed.* This requirement can be formalized as a Simulink diagram, as shown Fig. 4. The output is a single Boolean value that indicates whether the requirement is satisfied. Another functional requirement is: *While in SUSPENDED mode, the microwave shall enter COOKING mode if the start key is pressed.*

The testing tool Reactis accepts Simulink diagrams as test oracles in addition to more traditional assertions written as logical expressions. The requirement in Fig. 4 can also be thought of as a runtime monitor, signaling a property violation if the requirement is not satisfied. The user selects the values to pass to the model in Fig. 4 as inputs during testing. During test case generation, if any output is 0 or false, the assertion is reported as *failed*.

KLEE can also use the same formalized requirements. KLEE is used to generate symbolic inputs for the C code auto-generated from the Simulink models. Similarly we can use the code generator on the formalized requirements and synthesize assertions from the auto-generated code. We then instrument the code from the original model with the assertions. If a requirement is violated, KLEE reports an assertion violation and provides the failing test case.

Using these tools, one can quickly get feedback on problems in either the model or the requirements. Careful analysis of the mode controller in Fig. 3 reveals that the transition for decrementing the timer while COOKING has a higher priority than the one for moving to SUSPENDED, resulting in the safety property being violated when the door opens while cooking. Fixing the priority and re-running the analysis will show that the requirement is no longer violated. The functional requirement of *While in SUSPENDED mode, the microwave shall enter COOKING mode if the start key is pressed* is also violated. The failing test case characterizes the behavior of pressing both the start and clear keys simultaneously, and the clear key is taking precedence. This time, it is decided that the model is correct and the requirement is incomplete, and by adding a clause regarding the clear key to both the English requirement and the Simulink model the problem is fixed. This demonstrates an example where the feedback from the V&V processes can be used in refining the requirements.

6. DISCUSSION & CONCLUSION

In this paper we propose Simulink as a requirements formalization since several missions at NASA use the Simulink/-Matlab development environment. The same principles, however, can be applied to other modeling formalizations used by system engineers, e.g., LabVIEW or Scada. We believe it is best to represent requirements in a language that the engineers are already familiar with rather than impose a formal methods or software engineering notation on them.

When evaluating our approach using Reactis and KLEE on certain LADEE modules, we encountered several challenges. First, it is not easy for system engineers to install and run program analysis tools; this required someone with software engineering expertise. Secondly, it is not easy for system engineers to make sense of the output produced by these tools. For example, Reactis reports coverage results at the Simulink block level while KLEE reports coverage at the object-code level (LLVM bitcode). More specifically, it is not clear how to compare the coverage results computed on different structures. Finally, there are fundamental technical limitations to the tools as well. Reactis considers Simulink blocks that contain embedded MatLab as black boxes and does not attempt to cover the paths in those blocks. In KLEE, the underlying constraint solver cannot analyze code containing floating point operations and non-linear arithmetic. To solve the first two limitations, we believe it is important to build a common interface to a suite of tools so that the system engineers can leverage various tools without learning the nuances of each tool. We also need an additional meta-tool that can normalize the results from the various tools. To solve the technical limitations, we need to direct software engineering research to (a) develop better tools to target specific parts of the system based on their strengths, and (b) compose results from different tools.

Based on our experiences at NASA, there are many opportunities to transfer or adapt software engineering solutions for system engineers, however, there are several serious barriers to adoption. For mission-critical systems, the V&V processes are not as stringent as those for safety-critical systems. There are, however, tight bugdet restrictions and time schedules based on launch windows. If the deadlines are not met, these missions often run the risk of being cancelled. Anecdotal evidence suggests that the projects where the bridges between the Twin Peaks are shaky, often go over budget and end up getting cancelled. From talking to engineers who work on missions, we hypothesize that if there was a viable implementation of the approach proposed in this work, we could make inroads into helping system engineers take a more systematic approach based on software engineering principles.

7. ACKNOWLEDGEMENT

The authors would like to thank Steven Miller of Rockwell Collins Inc., for the Microwave controller Simulink model.

8. REFERENCES

[1] Simulink - simulation and model-based design. http://www.mathworks.com/products/simulink/.

[2] Stateflow - environment for modeling state machines. http://www.mathworks.com/products/stateflow/.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[6] J. Crow and B. Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Trans. Softw. Eng. Methodol.*, 7(3):296–332, 1998.

[7] Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/products/scade-suite/, 2014.

[8] J. S. Fitzgerald, P. G. Larsen, and S. Sahara. Vdmtools: advances in support for formal modeling in vdm. *SIGPLAN Notices*, 43(2):3–11, 2008.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] K. J. Letsholo, L. Zhao, and E.-V. Chioasca. Tram: A tool for transforming textual requirements into analysis models. pages 738–741, 2013.

[12] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *STTT*, 8(4):303–319, 2006.

[13] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.

[14] I. Rational. Rhapsody. http://www.ibm.com/developerworks/rational/products/rhapsody/, 2014.

[15] I. Rational. Statemate. http://www-03.ibm.com/software/products/en/ratistat, 2014.

[16] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[17] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam. Your "what" is my "how": Iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, 2013.