

Flight Critical Software and Systems Development Using ASSERT™

Kit Siu, Abha Moitra,
Michael Durling, Andy Crapo,
Meng Li, Han Yu,
Heber Herencia-Zapana,
Mauricio Castillo-Effen, Shiraj Sen
General Electric Global Research Center
Niskayuna, NY

Craig McMillan,
Daniel Russell, Sundeep Roy
General Electric Aviation Systems
Grand Rapids, MI

Panagiotis Manolios
Northeastern University
Boston, MA

Abstract—The size and complexity associated with software that monitors, controls, and protects flight critical products continues to grow. This is compounded by an increased use of autonomous systems which are just as complex, if not more so, since many operator responsibilities are supported and replaced by software in unmanned systems. Further, these systems are subject to cyber-enabled attacks, thereby necessitating another level of complex software to ensure security. General Electric has devoted a team to research and develop a new suite of tools to address the challenges with design, development, and verification of these software-intensive products. The goals are to develop technology, processes, and tools that result in more efficient software and system development as measured by cost and cycle time, and to enable new capabilities such as autonomy and the Industrial Internet. This paper will introduce the GE approach to formal requirements capture, requirements analysis, and auto test generation. We will introduce the ASSERT™ tool chain (Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests). We will demonstrate aspects of the tool on an autonomous aerial inspection system.

Keywords—UAS; Validation; Verification; Formal Requirements Capture; Semantic Model; Ontology; Requirements Analysis; Auto Test Generation.

I. INTRODUCTION

To address the cost and cycle time of developing flight critical software, we must first understand where faults are introduced in the software development process, when they are found, and the relative cost to repair them (Fig.1). As much as

35% of the faults are introduced in the requirements engineering phase, yet only 1% are found [1]. By the code development and unit test phase, collectively 90% of the errors will have been introduced, yet only 19.5% will have been found. Faults become costlier to fix later in the development process – 6.5x more to rework in the code development phase and as much as 40x to 110x more in system test and customer acceptance test [1]. There are numerous studies and publications that advocate for better requirements and better testing to minimize project rework cost and mitigate risks in later stages of the software development process [2, 3, 4]. Added to this is the special attention that flight critical software receives because its size and complexity continue to grow as we push more and more operator functionality to the software, and now even more so with autonomous systems where flight operation and safety must be handled without human intervention [5].

These concerns are not new to aviation and airborne systems. The need for quality software developed with managed cost has always been paramount to the reputation of the industry and the survival of its suppliers. There are published guidelines such as DO-178 by which certification authorities such as the FAA and EASA approve software based aviation systems. Motivated by the newest revisions to these guidelines [6] and the continued growth in size and complexity associated with flight critical software, GE has devoted a team of researchers to develop a new suite of tools that augment the software development process with technology that discovers and removes faults early (Fig. 2). In the Requirements Capture phase we introduce technology to write requirements in a way that is unambiguous, human

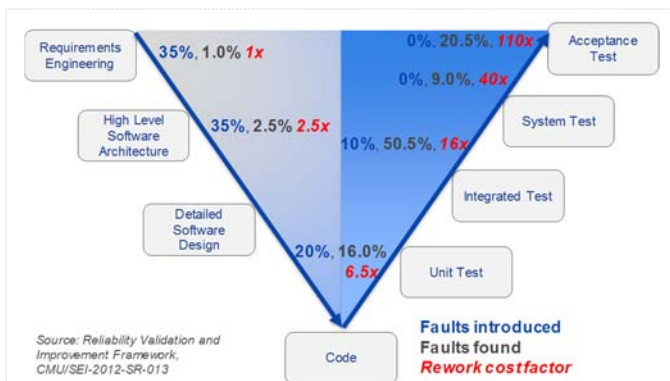


Figure 1. Software development process and where faults are introduced.

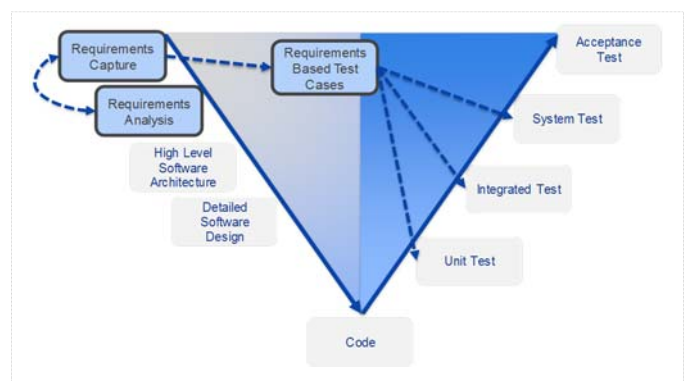


Figure 2. The software development process augmented with technology using ASSERT™ that discovers and removes faults early.

readable, and computer analyzable. Because the requirements are formalized, we analyze them in the Requirements Analysis phase to provably say, among other things, if two or more requirements conflict with each other or if a set of requirements are complete, etc. If requirements do not satisfy the analyses, the end user is then able to correct the requirements, guided by error localization and counterexamples, and apply Requirements Analysis again. After the requirements have passed analysis, test cases are automatically generated to produce a set of Requirements-Based Test Cases to be run against the artifacts produced during the Detailed Software Design and Code phases. These test cases are applied during various test phases. The technologies mentioned above are packaged within our tool called ASSERT™ - Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests.

There are other tools that capture formal requirements and analyze them [7, 8, 9, 10, 11]. One tool in particular is SpeAR – Specification and Analysis of Requirements [12, 13], which is an open source tool and language that allows for capturing and analyzing requirements. SpeAR uses semantics of the Lustre synchronous dataflow programming language and Past Linear Temporal Logic (Past LTL) to express temporal relationships. SpeAR supports requirements analysis, real-time type checking, dimensional analysis of unit computations, and other checks to provide feedback to users that assists them in the elaboration of well-formed requirements. Some features of SpeAR are comparable to our tool, though we believe that our tool differs in the following ways: 1) we rely on ontology-based technologies, which allow for automated reasoning about concepts and relationships in the domain relevant to the system and to the requirements that are being refined and analyzed, and 2) we generate tests based on requirements even before a single line of executable code is written. ASSERT™ packages all these capabilities in a tool suite.

In this paper we will work through one illustrative example using ASSERT™. The paper is organized as follows. Section II describes the autonomous aerial inspection system, a new GE product, to which we applied ASSERT™ and which we will use as the illustrative example throughout the paper. Section III describes the Requirements Capture technology. Section IV describes the Requirements Analysis technology. Section V describes the Requirements-Based Test Case generation technology. Section VI describes how code is validated against the auto generated requirements-based test cases in a relevant test environment. Conclusions are described in Section VII.

II. AN AUTONOMOUS AIR INSPECTION SYSTEM

Early in 2017 GE Ventures spawned *Avitas Systems—a GE Venture* [14] which was the result of almost two years’ worth of effort between researchers and leaders from GE Global Research, GE Oil & Gas (O&G), and GE Ventures. This collaboration culminated in an offering to meet the needs of O&G customers – an autonomous flare-stack inspection product that reduces inspection time and cost and decreases risk to humans, while improving accuracy through automation. The inspection solution uses a small unmanned air system (sUAS) called Euclid. Euclid’s software components include the Robot Operating System middleware (ROS), mission planning, mission execution, flight management, user interfaces, and

vehicle health management. The health management includes Automated Contingency Management (ACM), which is modeled after prior work at NASA [15]. The goal of the ACM is to autonomously adapt to fault conditions while still achieving mission objectives. It brings the sUAS to a degraded nominal state if reconfiguration is feasible; otherwise it switches to a Fail-safe mode to guarantee safety (Fig. 3). The ACM will be the example in this paper to demonstrate how ASSERT™ was used to capture, analyze, and refine system requirements.

The ACM is dependent on the proper performance of other supporting components on the sUAS, such as the battery, sensors, and the radio link. Euclid’s software includes anomaly detection of these components. We will demonstrate how ASSERT™ was used to generate requirements-based test cases that test the software implementation to detect anomalies in the power supply (i.e., the battery).

III. REQUIREMENTS CAPTURE

We introduce a Requirements Capture language developed for use by a requirements engineer that is as close as possible to English, allowing her to write requirements using terms and concepts from her domain, yet is formal enough that requirements written in this language can be analyzed using formal methods. The underpinning concept is a model that captures domain concepts in an unambiguous way. By unambiguous we mean preciseness in concept definition—each concept has a single meaning that will be shared across all requirements and by users of the model. This forms a domain model, which is a formal representation of the concepts that are used in the domain and what those concepts mean. To build this domain model, we use an ontology language that is based on set theory and a decidable fragment of first order logic. It has been hypothesized that set theory is what humans apply when forming mental models [16, 17, 18] – an organization of objects based on a human’s perception of the world. Having a requirements language that formalizes objects in the same way that an engineer’s mind works makes the language feel simple and natural to use.

Set theory allows us to formalize collections of related objects using sets, also known as classes, and allows us to formalize relationships between classes, including class hierarchies. Classes and subclasses have members, or instances. Properties describe relationships between instances and can be refined with domain and range, the classes to which these instances belong. As was the case with classes, there can be hierarchies of properties. Set theory enables inheritance, which leads to parsimony in model development.

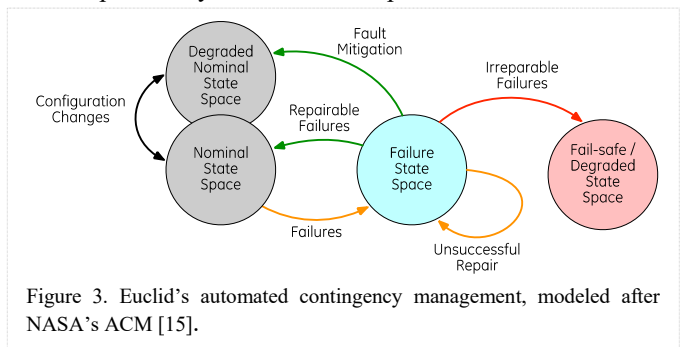


Figure 3. Euclid’s automated contingency management, modeled after NASA’s ACM [15].

Giving the requirements engineer a language that is well-founded allows her to come up with a model that captures the domain in a formal notation. Once these domain concepts are defined, they can then be used to express requirements. The requirements are understandable to practitioners in the domain because they are expressed in terms compatible with subject matter experts' mental models and compatible with the literature of the domain. The focus is then shifted away from the formalism itself and towards the domain.

A. From OWL to SACL to SRL

There are several choices when it comes to ontology languages. We selected the Web Ontology Language (OWL) [19], which combines set theory with a decidable fragment of first order logic, because it is a "standard" of the World Wide Web Consortium (W3C) [20] and it is widely used.

The standard serializations of OWL and its variants are not very easy or natural for human users to read and write. This was the motivation for developing a controlled-English language called the Semantic Application Design Language (SACL), which is Open Source under the Eclipse Public License [21, 22]. SACL can express all the constructs of OWL Version 1 and qualified cardinality from OWL Version 2.

SACL allows us to define domain models and also supports rules as a means of capturing additional domain knowledge. SACL also provides a number of constructs to help test, debug and maintain domain models. All valid SACL models are saved both in SACL textual syntax and in a user-specified standard serialization of OWL. If rules are present they are saved in a user-specified target rule language as well.

Besides being an ontology language, SACL is also an Xtext-based integrated development environment (IDE) [23]. This environment provides semantic coloring of different types of concepts in models, hyperlinking of concepts to their definitions and usage, integration with source code control systems via Eclipse plug-ins, graphical visualization of models, type checking, content assist, and other functionality useful for creating and maintaining models over their lifetime.

More recently, SACL has been extended to support requirements capture and analysis, which forms the requirements capture environment for ASSERT™. The SACL Requirements Language (SRL) [24, 25] is very similar to the SACL rule language but provides additional expressivity necessary for capturing requirements. It also provides specialized keywords and constructs to make it as similar to natural-language requirements as possible. Here's an example of a requirement written using SRL.

```
Requirement R1.3:  
SYSTEM shall set LNAV_Valid of Is_LNAV_Valid to true  
when Air_Ground of Aircraft is In_Air.
```

This requirement, with identifier R1.3, says that the *System* shall set *LNAV_Valid* to true when the property *Air_Ground* is *In_Air*. SRL differentiates between ontology properties whose value is being set in the "shall" part of the requirement and those providing the value to be used in the conditional "when" part of the requirement. The former we call the controlled variables, and the latter we call the monitored variables.

In addition to the basic type of requirement just illustrated, SRL supports various other kinds of requirements that requirements engineers have identified as important for requirements specification.

- Time-dependent conditions with or without a specified interval, using the keywords *previous*, *was*, *was ... in the past* <n> <time-unit>, *has been*, *has been ... for the past* <n> <time-unit>
- Events, where an event is something that happens at a point in time.
- Typed lists, where a list is a sequence of instances or values of a particular class or type. We allow users to define named and unnamed list classes and we support about a dozen list operations for replacement, insertion, retrieval, querying, appending, filtering, ordering, etc.
- Tables provide a compact representation for what would otherwise be a set of requirements. Each input column specifies a set of values of an expression over monitored variables; each output column specifies a corresponding set of values of a controlled variable. Each row corresponds to one requirement in the set of equivalent requirements.
- Decomposition provides abstraction and structuring mechanisms. Abstract decomposition is used to formalize requirements that contain unspecified behavior, which will be refined in lower-level requirements. Decompositions can be linked together, providing a structuring capability that allows one to define functionality that can be reused in multiple contexts.
- Equations are used to describe mathematical concepts and computations that can be used across different requirements and projects.
- Assumptions are used to place constraints on monitored variables. They are used to characterize the valid set of inputs for a component.
- Assertions are used to formalize conditions under which the value of the controlled variable is immaterial ("don't care" conditions).

The ASSERT™ requirements capture environment is linked with IBM Rational DOORS®, which may be used for requirements management. Requirements are captured using Eclipse and managed using DOORS®. Movement of requirements between DOORS and ASSERT™ is made possible through DOORS eXtension Language (DXL) scripts.

B. Requirements Capture applied to Autonomous Air Inspection System

We developed a domain model for our Autonomous Air Inspection System. Fig. 4 shows a portion of the domain model that captures some of the classes and properties for the Autonomous Air Inspection System. The portion of the model shown in Fig. 4 focuses on aspects that relate to ACM and code validation that we will discuss in more detail later. Classes appear in bold blue, instances appear in light blue, and properties are in green.

```

// Portion of Model
Drone is a type of Equipment
described by mode with a single value of type DroneMode,
described by controlAction with values of type DroneAction.

component of Drone has exactly 1 value of type Battery.
component of Drone has exactly 1 value of type TempSensor.
component of Drone has exactly 1 value of type RC_Link.

Battery is a type of Equipment.
TempSensor is a type of Equipment.
RC_Link is a type of Equipment.

DroneMode is a class,
  must be one of {Drone-on-ground, Drone-in-air}.

DroneSystem is a type of System,
described by contingency_action with values of type boolean,
described by flight_type with values of type FlightType,
described by failure_type with values of type FailureType,
described by failure_state with values of type FailureState.

FailureType is a class,
  must be one of {Fault_Mitigation, Irreparable}.
FlightType is a class,
  must be one of {Pre_Flight, In_Flight}.
FailureState is a class,
  must be one of {Nominal,
                 Failure_State_Space,
                 Degraded_Nominal_State_Space,
                 Degraded_State_Space}.

battery_ is a class
described by warning_threshold with values of type decimal,
described by critical_threshold with values of type decimal.

```

Figure 4. Portion of the Autonomous Air Inspection System domain model.

A selection of the requirements for the ACM are shown in Fig. 5. This set considers *Failure_State_Space* and the conditions under which it stays in the same state or transitions to *Degraded_Nominal_State_Space* or *Degraded_State_Space* as illustrated in Fig. 3.

We want to highlight that both the model and the requirements are easily understood by a subject matter expert. Furthermore, capturing the model and the requirements is facilitated in the Eclipse IDE by type checking, content assist, etc.

IV. REQUIREMENTS ANALYSIS

The requirements are formally analyzed using the GE patented Requirements Analysis Engine (RAE) [26]. RAE is based on theorem proving technology and is the reasoning engine of the ASSERT™ tool. RAE is built on top of A Computational Logic for Applicative Common Lisp Sedan (ACL2s) [27]. ACL2s is an open source, Eclipse plug-in that provides an integrated development environment and extends the ACL2 theorem prover. The ACL2 theorem prover is an industrial-strength theorem prover that has been used to prove some of the most complex theorems ever proven about industrial systems [28]. ACL2s extends ACL2 with a powerful termination analysis capability [29, 30], an automated counter-

```

// Selection of ACM requirements: initial set
Requirement ContinueInFailureStateSpace:
DroneSystem shall set failure_state of DroneSystem to
Failure_State_Space
when previous failure_state of System is Failure_State_Space
and failure_type of DroneSystem is Irreparable
and battery_state of state_ is not battery-critical.

Requirement ToDegradedStateSpace:
DroneSystem shall set failure_state of DroneSystem
to Degraded_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and flight_type of DroneSystem is In_Flight
and failure_type of DroneSystem is Irreparable
and battery_state of state_ is battery-critical.

Requirement ToDegradedStateSpace2:
DroneSystem shall set failure_state of DroneSystem
to Degraded_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and failure_type of DroneSystem is Irreparable
and flight_type of DroneSystem is Pre_Flight.

Requirement ToDegradedNominalStateSpace:
DroneSystem shall set failure_state of DroneSystem
to Degraded_Nominal_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and failure_type of DroneSystem is Fault_Mitigation.

// Battery Level
Requirement loadBatteryLimits:
ContingencyParameters shall set setBatteryLimits of
contingency_parameters_ to true
only when (warning_threshold of battery_ >= 0.0 and
warning_threshold of battery_ <= 100.0)
and (critical_threshold of battery_ >= 0.0 and
critical_threshold of battery_ <= 100.0)
and warning_threshold of battery_
>= critical_threshold of battery_.

```

Figure 5. Selection of ACM requirements: initial set.

example generation engine [31, 32] and an expressive data definition framework [33].

The Requirements Analysis Engine (RAE) analyzes requirements captured in the ASSERT™ requirements capture environment. RAE can be used to analyze both complete and incomplete sets of requirements, i.e., RAE can provide meaningful results as soon as requirements are written, even if not all the requirements in a project are completely captured. Requirements are analyzed without requiring access to lower-level requirements, executable code or higher-level requirements [26].

RAE finds requirements errors early in the process. In fact, it finds the errors as soon as requirements are written. As we argued above, numerous studies have shown that one of the primary reasons that the development of safety-critical systems is so expensive is that errors tend to be introduced early in the design process, but are only caught late in the design process. The longer the gap between error introduction and error discovery, the larger the costs associated with the error. By

providing requirements engineers with immediate feedback on their requirements, RAE can prevent expensive rework that potentially involves not only correcting the requirements with the error, but also propagating those changes forward. This can affect everything that depends on the corrected requirements, including lower-level requirements, architecture decisions, derived requirements, code, etc. With RAE, the costs of fixing such errors can be significantly reduced: even orders of magnitude savings are possible.

If RAE reports an error, then it has a proof that the requirements are actually erroneous. This is important because false alarms tend to frustrate users of formal methods tools. If a certain percentage of errors identified by a tool turn out to not be actual errors, then the effort spent by requirements engineers in determining what is and what is not an actual error can partially offset the effort saved by finding errors early.

When RAE discovers an error, it localizes it and provides actionable feedback to the requirement engineer. Typically, this feedback is in the form of a counterexample [31, 32], a minimal set of requirements, the name of the analysis that failed, and a helpful explanation. The counterexample can be thought of as a test case that exhibits why the analysis in question fails for the set of requirements identified by RAE. Given that projects can easily contain thousands of requirements, localizing errors by identifying a small set of requirements that are needed to exhibit the error is of great practical importance. While most analysis failures generate counterexamples, some do not. For example, if requirement R is implied by a set of other requirements, then we have an independence error. Notice that no test case can exhibit an independence error. In fact, one would need to consider all possible test cases to see that requirement R is implied by a set of requirements. In such cases, RAE will identify a subset of the requirements that imply R and RAE will provide a helpful explanation. Given that RAE has a proof of independence, not just inconclusive evidence of independence, the requirements engineer can focus all her energy on determining what is wrong with the requirements.

The Requirements Analysis Engine performs a set of requirement analyses, as described in this section. These analyses are done in an order, prescribed in the following way. First, simpler analyses come before more complex analyses. Simpler analyses can be performed more quickly and they tend to find obvious, egregious errors that should be corrected right away. Second, if performing analysis A can help simplify analysis B, then analysis A should come before analysis B. Some of the analyses are simplified with the assumption that the preceding analyses have already been performed.

Analyses are turned into (ACL2s) theorem proving queries. In typical cases, such queries return “passed” or “failed.” However, in general, the result of a query can also be “undetermined.” This can happen if the query uses an undecidable fragment of logic. Another reason for undetermined results is that the underlying logic may be intractable (e.g., the computational complexity of the decision procedure for the logic may be non-polynomial) or the theorem prover may run out of available resources (e.g., a time-out may occur). “Undetermined” results are not reported as errors, but are reported as potential errors that requirements engineers are free

to ignore. Our experience has been that undetermined results are rare.

Currently, requirements analysis is performed only when initiated by the user. We are working on changes to the tool so that some of the analyses will be done immediately and feedback can be provided in real time. The goal is the integration of RAE and the Requirements Capture environment such that the selected analyses occur automatically as soon as the user updates the requirements, without the need for the user to explicitly initiate the analysis, and RAE errors are displayed as markers in the requirements editor.

The set of requirements analyses performed by RAE can be found in [26]. This includes type-safety, contingency, independence, conflict and completeness analyses. In the next subsection, we show how RAE is used to analyze our example system. The mathematical foundations of these analyses can also be found in [26].

A. Requirements analysis applied to Autonomous Air Inspection System

The model and requirements developed for the Autonomous Air Inspection System were analyzed using RAE and the results were presented in the RAE Viewer (see Fig. 6). The viewer is a compact, user-friendly way of communicating the analysis results to the end-user, with the results highlighted in color and

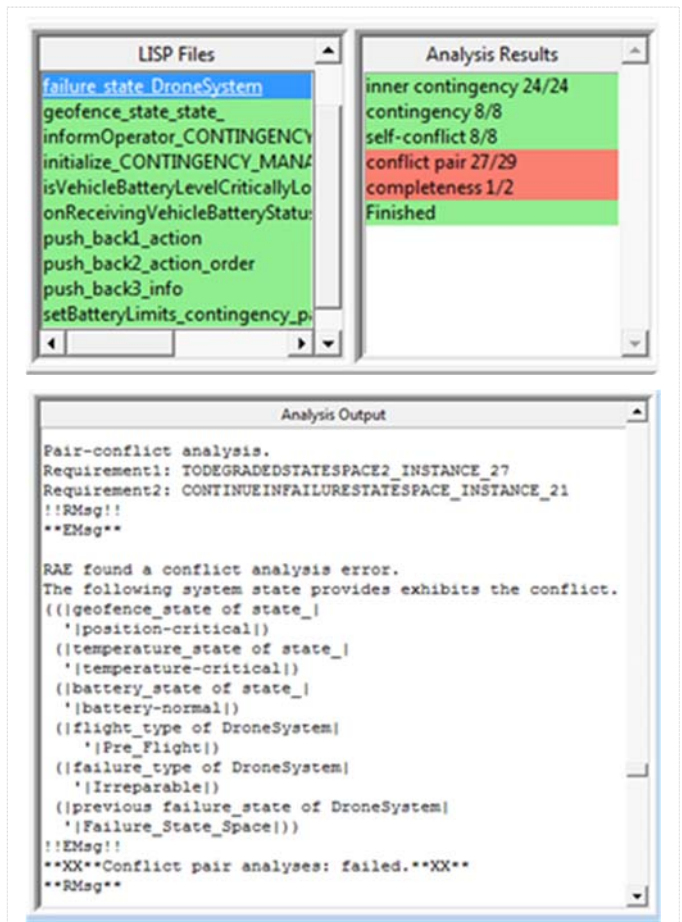


Figure 6. RAE analysis of the initial set of ACM requirements.

actionable feedback in the form of counterexamples displayed in the output window. Portions of the viewer with analysis results of the requirements from Fig. 5 are shown in Fig. 6. RAE groups outputs into ACL2s files. The top part of Fig. 6 shows all the ACL2s files that were generated and all but one of the files are shown with green background, meaning that all the requirement checks passed successfully. The remaining file (*failure_state_DroneSystem*) appears with red background indicating that some requirement checks have failed. Clicking on that file shows the details of requirement checks that were run for that file and whether the checks passed or failed. Fig. 6 shows that conflict and completeness checks failed. The RAE Viewer output window shows the detailed results of all the checks run and the reason for the conflict check failing. First of all, it identifies the two requirements that are conflicting (*ContinueInFailureStateSpace* and *ToDegradedStateSpace2*) and then it specifies a system state under which bodies of both of these requirements are satisfied and the controlled variable is being set to two different values. This illustrates how RAE provides localization (which two requirements have a conflict in this example) and provides a counterexample (the system state) which shows the conflict. The counterexample is a test case for which one requirement claims that *failure_state* of *DroneSystem* should be set to one value, but the other requirement states it should be set to a different value.

Based on the localization and counterexample, shown in the RAE Viewer, the requirements engineer will then be able to refine the requirements to correct the issue. In this case, an extra “when” condition was left off of one of the requirements and the corrected set of requirements are shown in the top of Fig. 7 (the condition that was added has the comment “// added” after it to make it easier to see the change). With this correction, the model and requirements were analyzed again using RAE and a portion of the RAE viewer is shown in the bottom of Fig. 7 indicating that all the requirements pass all the checks.

So, to recap, the requirements engineer would refine and revise the requirements over possibly multiple iterations until all the requirements pass all the checks.

V. REQUIREMENTS-BASED TEST CASES

Software architecture, detailed design, and code is developed after getting a set of good requirements. We demonstrated in the previous sections that the set of requirements coming from Requirements Capture and Requirements Analysis of ASSERT™ are unambiguous, conflict-free, and complete. After the software is designed it needs to be tested, to verify that it adequately satisfies the requirements. Verification of safety-critical flight systems can account for as much as a third of the overall cost of the system [34]. The timeliness of the verification process can also impact overall business costs.

There is a wealth of publications on automating aspects of the testing process to cut down the cost of verification. In fact, many software development environments provide the capability to automate test execution. Test generation, however, remains a largely manual process in the industry. There are several automation approaches in the literature, some suggesting the use of a specification model like UML [35, 36]. The idea is

```
// Selection of ACM requirements: corrected set
Requirement ContinueInFailureStateSpace:
DroneSystem shall set failure_state of DroneSystem to
Failure_State_Space
when previous failure_state of System is Failure_State_Space
and flight_type of DroneSystem is In_Flight // added
and failure_type of DroneSystem is Irreparable
and battery_state of state_is not battery-critical.

Requirement ToDegradedStateSpace:
DroneSystem shall set failure_state of DroneSystem
to Degraded_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and flight_type of DroneSystem is In_Flight
and failure_type of DroneSystem is Irreparable
and battery_state of state_is battery-critical.

Requirement ToDegradedStateSpace2:
DroneSystem shall set failure_state of DroneSystem
to Degraded_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and failure_type of DroneSystem is Irreparable
and flight_type of DroneSystem is Pre_Flight.

Requirement ToDegradedNominalStateSpace:
DroneSystem shall set failure_state of DroneSystem
to Degraded_Nominal_State_Space
when previous failure_state of DroneSystem
is Failure_State_Space
and failure_type of DroneSystem is Fault_Mitigation.
```

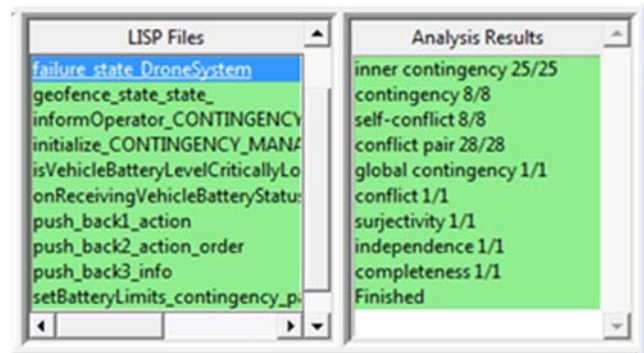


Figure 7. Selection of ACM requirements: corrected after reviewing RAE analysis.

to generate test cases from UML diagrams such as state-charts, use-case diagrams, and sequence diagrams, which describe the software independent of the way a given system is designed. This is a fine approach if UML is already part of the business software development process. Still researchers and practitioners have found that UML diagrams alone are not enough to generate testable test cases. In one case UML diagrams needed to be augmented with information from the design or from a data dictionary [37].

Then there are other approaches that automate test generation from a design model or a software program. This is an especially common approach for generating structural code coverage tests. There are several approaches that use formal methods. For example, one uses a model checker to generate

counterexamples for when an input executed on a program is “always not P,” where P is a formula that describes a desired structural coverage criteria [38]. This is also a fine approach, except that test cases generated from a design cannot be considered independent and a test generation method that depends on having a design does not help eliminate errors as early as possible in the software development process.

Our approach is to use what we have captured up to this point using ASSERT™ to generate a set of requirements-based test cases. In this way, we make further use of our well-developed, formalized set of requirements by developing technology to automatically generate test cases from requirements. These are tests cases that can be executed on the code during Unit Test, Integration Test, and System Test (Fig. 2). Requirements-based test cases specify sets of inputs, conditions, and expected outputs derived from requirements and used to verify compliance of the design with the requirements. Because the requirements are formalized, they are rich with information such as the inputs and outputs, their types, and their relationships. We built an Automated Test Generation (ATG) tool that automatically extracts this information from the requirements, translates the information into variable types such as boolean, real, or integer and constructs their relationships into expressions (Fig. 8).

ATG builds XML data models and refines them along the way. First, the ATG Translator takes as input the requirements from ASSERT™’s Requirements Capture and translates them into an Intermediate Model, which is an XML data model conforming to an ATG XML schema. This intermediate model contains all the expressions from the requirements, including any constraints expressed as assumptions in the requirements. The ATG Reasoner then analyzes the Intermediate Model and, based on the requirement expressions and data types, identifies the test strategies that need to be applied to generate the test cases. Test strategies can be thought of as ATG’s approach to

generating tests for Modified Condition/Decision Coverage (MC/DC), equivalence class, robustness, boundary value, etc. In addition to the Intermediate Model, the Reasoner can also incorporate manually generated test cases into its analysis. ATG supports the capability to add manual test cases because in industry there could be a set of legacy test cases for a product line, or there could be some special test points that an engineer would like to test. The Reasoner produces Internal Test Objective Models (ITOMs), which are data models of test cases in XML, produced to conform to an ATG ITOM schema. At this point the ITOMs are used to produce two things: human-reviewable test cases and machine-readable test procedures. The purpose of having human-reviewable test cases is so the test engineer can read from a text file a list of test cases and the test objectives they cover. This text file is produced by the ATG Interpreter. The machine-readable test procedures are in XML which can be further converted to a format executable in a test environment relevant to the design, making the test procedures generally applicable. The ATG Synthesis produces the test procedures by performing several things: 1) reachability analysis, 2) expression simplification, 3) test case optimization, 4) test data generation, and 5) test procedure generation.

A. Test Strategies

Test strategies can be thought of as ATG’s approach to generating tests that align with DO-178C test objectives, such as MC/DC, boundary value, equivalence class, and robustness test [6]. The test strategies are listed in Table 1 and are designed to map to the kinds of requirements found in industry, supported by ASSERT™’s Requirements Capture.

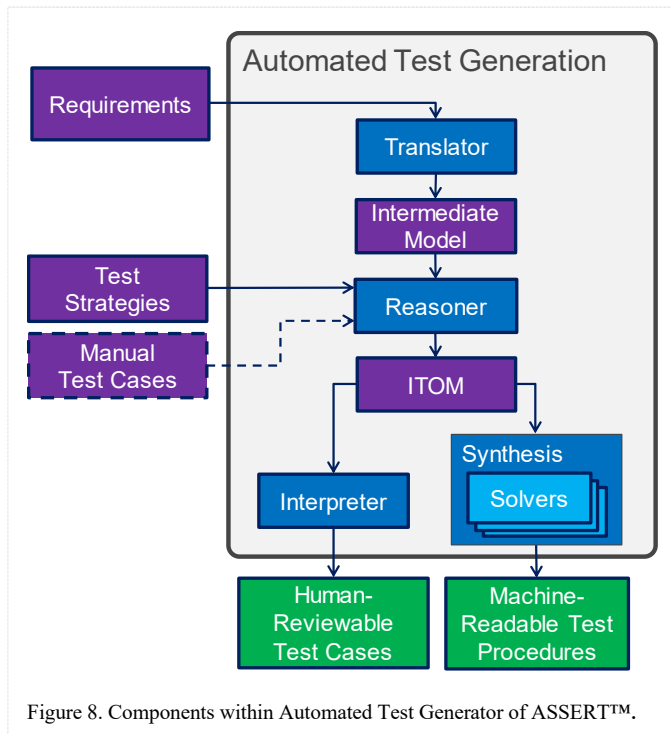


TABLE I. REQUIREMENTS-BASED TEST STRATEGIES

Test Strategies	Applied Requirement Examples	Test Objectives
Logic Coverage	SYSTEM shall set mode of CMA to Normal when level > min and level <= max.	MC/DC, Decision Coverage, Condition Coverage
Event	SYSTEM shall set x to y when event1 is received for SYSTEM	Event can only be the triggers
Equivalence Class	SYSTEM shall set x to y when y > 50 and z < 100	Boundary value, Equivalence class, and Robustness
List	SYSTEM shall replace x in list1 with z when list1 contains x	Min and max list size. First, middle and last elements.
Timing	SYSTEM shall set x to y when x has been y for the past 5 seconds	Leading trigger and Lagging trigger
Assumption	SYSTEM assumes that x is y	Assumption s invalidate test cases
Equation	SYSTEM shall set x to Formula(x, y). Equation Formula (real i2, real i8) returns real (sin(i2)^2 + cos(i8)^2)/2	Sufficiently test parameter range

Figure 8. Components within Automated Test Generator of ASSERT™.

We will now provide a more in-depth discussion of one of these test strategies – logic coverage as it relates to MC/DC test objective [39]. MC/DC is a method of ensuring adequate testing for safety-critical software. It refers to having test cases that cover all the following: 1) each entry and exit point in a software statement is invoked, 2) each decision in the software takes on all possible outcome, 3) each condition in a decision in the software takes on all possible outcome, 4) each condition in a decision in the software independently affects the outcome of the decision. MC/DC is software centric – it is thought of in the context of code coverage, thereby it invokes ideas on how to exercise the code and leads to automation solutions that rely on using the code to generate tests [40, 41, 42]. A point of departure with our tool is that we insist on testing 100% of the requirements, automatically generating test cases *from the requirements*. Our conjecture is that if the code is a truthful representation of the high-level requirements, then MC/DC test cases generated from requirements are applicable to the design and are expected to provide a high degree of coverage. After applying the requirements-based test cases, if the design and structural coverage analysis finds gaps in code coverage, then we review the code for things like undesired behavior, dead code, and derived requirements. The requirements are also further refined and implementation decisions are made, including decisions on data structures and algorithms.

Here is an illustrative example of requirements-based test cases automatically generated following MC/DC test objectives. Suppose ATG receives a requirement that says:

```
SYSTEM shall set LNAV_Valid of Is_LNAV_Valid to true
when
( Air_Ground of Aircraft is In_Air and
( Speed of Aircraft > 100) or
(Active_Plan_Exists of FlightPlanManager is true) or
(LNAV_TO_Capable of Is_LNAV_Valid is true) ) ).
```

The ATG Translator produces an intermediate model containing a mathematical expression equivalent to the logic diagram in the top of Fig. 9. This requirement has 4 conditions

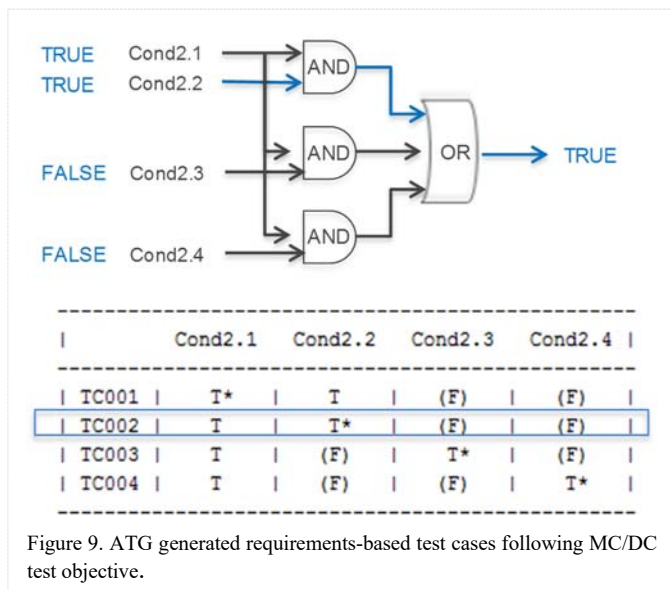


Figure 9. ATG generated requirements-based test cases following MC/DC test objective.

and 4 decisions. The ATG reasoner applies the logic coverage test strategy and produces the set of test cases in a human-reviewable format as shown in the bottom of Fig. 9. In this example, four test cases are produced following MC/DC criterion. The highlighted test case TC002 demonstrates that condition 2.2, marked with an asterisk T*, independently affects the decision in this requirement. (Note: the parentheses around F have no meaning; they are only there for readability, to more easily distinguish the letter T from F.) This set of test cases alone would not satisfy MC/DC because it does not contain a test for when all the conditions are set to F. ATG only generates test cases that satisfy each requirement. A test case where all the conditions are F would have to come from another requirement where *LNAV_Valid* is set to F. If this requirement was missing from the set, the user would have been notified by the RAE of a requirements completeness error.

B. ATG Synthesis

The ATG Synthesis takes as input the ITOMs and performs several functions: 1) reachability analysis to filter out the test cases that are invalid, 2) expression simplification to reduce redundant expressions in the ITOM, 3) test case optimization to combine test cases to get a reduced set, 4) test data generation to generate test data that satisfies the constraints indicated by the test cases, and 5) test procedure generation to sequence the test cases with the test data into test procedures.

The ATG Synthesis analyzes the ITOMs and synthesizes their equivalent satisfiability modulo theories (SMT) formula. Different SMT constraint solvers are invoked depending on the input data types of the SMT formulas. For example, one SMT solver may be invoked to solve an SMT formula that contains nonlinear arithmetic constraints, while another SMT solver may be invoked to solve the simple first-order relational or logical constraints. The solver utilizes applicable theories (e.g., theory of arrays, theory of datatypes, theory of linear and nonlinear arithmetic) to check the satisfiability of the SMT formulas. If the SMT formulas are satisfiable, a test vector will be generated with concrete values assigned to the variables in the ITOM, making up the test data for the test procedures. In case of unsatisfiability, the ITOM will be reported as unreachable with the corresponding rationale provided from querying the solver.

The ATG Synthesis reduces the number of test cases using an SMT solver. For example, heuristics-based optimization can guide the SMT solver to generate a single test case which satisfies the constraints in more than one ITOM. This implies that several test cases can be combined and tested together. Fig. 10 shows a simple example of how test cases can be combined. Reducing the number of test cases is always desirable since it has a direct impact on reducing the cost of testing, an important factor in industry.

C. Requirements-Based Test Cases applied to Autonomous Air Inspection System

ATG generated 105 test cases to cover all the test strategies for the ACM requirements in the Autonomous Air Inspection System. Fig. 11 shows one of these for testing the requirement

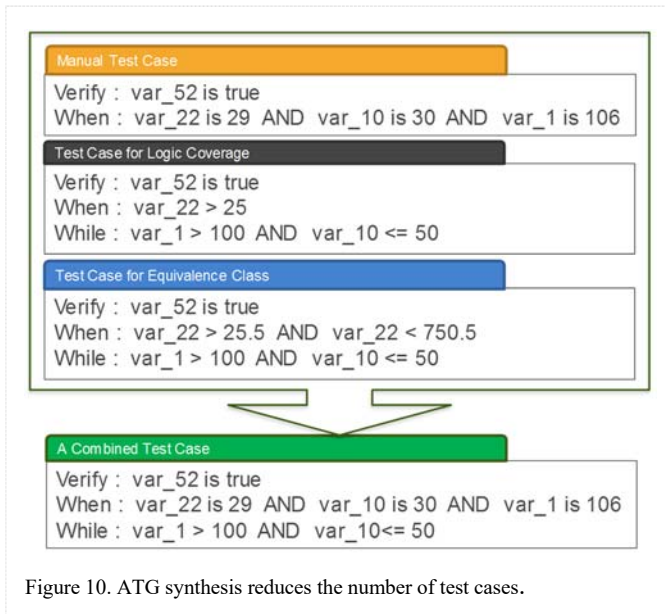


Figure 10. ATG synthesis reduces the number of test cases.

named *loadBatteryLimits* from Fig 5. This requirement specifies when the property *setBatteryLimits* is to be set to true; a portion of the human-reviewable test case file is shown at the top of Fig. 11 for logic coverage for that requirement. Notice that the human-reviewable test case file shows which condition is being verified in the test case (under the “when“ part) and which conditions are being held (under the “while” part). The human-reviewable test cases do not contain the concrete test values. These values come from the auto-generated test procedures, which will be used in code validation.

VI. CODE VALIDATION

The auto-generated test procedures containing concrete test values are written in XML and converted to an executable format for testing in an environment relevant to the design. Some commonly used test environments in the aviation industry are Esterel SCADE LifeCycle Qualified Testing Environment, Rational Rhapsody Test Conductor, and VectorCast. To test a design implemented using SCADE, for example, we wrote a translator to convert the test procedure from XML to .sss scripts. For our Autonomous Air Inspection System, we demonstrate the use of Google Test to test the ACM code which is written in C++.

Google Test is a testing framework for C++ based on the xUnit architecture [43]. Behavior of the system is expressed in Google Test using assertions, which are macros that resemble function calls. The results of the assertions can be success, nonfatal failure, or fatal failure. Fatal failures cause Google Test to abort the program execution, otherwise it continues with the run. In either case, when an assertion fails, it provides the test engineer with information about the file, line number, and a customizable message about the failure. Fig. 11 shows an example where we translate our test case into nonfatal assertions expressed using the functions `EXPECT_FALSE(condition)` and `EXPECT_TRUE(condition)`. The test case from the top of Fig. 11 is automatically translated into a two-step test procedure, shown in the middle of Fig. 11. The first step of the test procedure is a negation of the verify part, negation of the when

```

365 Test Case setBatteryLimitsTrue-TC002_00A:
366 verify
367 setBatteryLimits of contingency_parameters_ is true
368 when
369 warning_threshold of battery_ <= 100.0
370 while
371 (warning_threshold of battery_ >= 0.0) AND
372 (critical_threshold of battery_ >= 0.0) AND
373 (critical_threshold of battery_ <= 100.0) AND
374 (warning_threshold of battery_ >= critical_threshold of battery_)
...

////////////////////////////////////
//Checking SetBatteryLimits()
////////////////////////////////////
TEST(SetBatteryLimits, setBatteryLimitsFalse_TC002_00A)
{
  ContingencyParameters test;
  int critical_threshold = 0.0;
  int warning_threshold = 101.0;

  EXPECT_FALSE(test.setBatteryLimits(critical_threshold, warning_threshold));
}

TEST(SetBatteryLimits, setBatteryLimitsTrue_TC002_00A)
{
  ContingencyParameters test;
  int critical_threshold = 0.0;
  int warning_threshold = 100.0;

  EXPECT_TRUE(test.setBatteryLimits(critical_threshold, warning_threshold));
}

[ RUN      ] SetBatteryLimits.setBatteryLimitsFalse_TC002_00A
Contingency Parameters: Battery Warning Threshold of 101 is not valid
[ OK      ] SetBatteryLimits.setBatteryLimitsFalse_TC002_00A (0 ms)
[ RUN      ] SetBatteryLimits.setBatteryLimitsTrue_TC002_00A
[ OK      ] SetBatteryLimits.setBatteryLimitsTrue_TC002_00A (0 ms)

```

Figure 11. ATG test cases, Google Test script and results.

part, and satisfaction of the while part. The test procedure automatically generated concrete values for *critical_threshold* = 0.0 and *warning_threshold* = 101.0 to satisfy the first step. The second step is a set of concrete values to satisfy the verify, when, and while parts of the test case, notably *critical_threshold* = 0.0 and *warning_threshold* = 100.0. Notice also that the first step of the test procedure is actually the test case from the inverse of the requirement for when the property *setBatteryLimits* is to be set false. Because ATG automatically sequences test cases, what we are seeing here is two test cases sequenced and tested together. The result from Google Test are shown at the bottom of Fig. 11, demonstrating that the code adequately satisfies the requirement.

VII. CONCLUSION

The ASSERT™ tool chain is currently being used on several projects within GE where we have seen reduced cost and improved cycle time. The tool chain is generally applicable and is most relevant to projects for industries that must demonstrate adherence to requirements, either as a contractual obligation, insurance liability, or as part of customer acceptance test. We have used ASSERT™ on a flight management application, an avionics network switch program, an aircraft engine health monitor program, and a safety controller for a power system. All these programs experienced the following common benefits. Capturing requirements using ASSERT™ allowed users to have the same mental model, which was necessary on large projects with many developers spanning multiple functions and organizations. Analyzing requirements with ASSERT™ allowed us to catch and correct requirement errors early in the development process. Auto-generated test cases and test procedures using ASSERT™ allowed for test cases to be generated before writing a single line of code. We hope that with more demonstrated successes, ASSERT™ can become a companywide, or even an industry, standard.

Please contact Michael Durling (durling@ge.com) for more information about using ASSERT™.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge Christin Rauche, Scott Stacey, Gary Quackenbush, and Michael Idelchik for their unwavering support throughout the course of this research.

REFERENCES

- [1] Software Engineering Institute, "Reliability validation and improvement framework," Special Report, CMU/SEI-2012-SR-013, November 2012.
- [2] R. Charette, "Why software fails," IEEE Spectrum, pp. 42-49, September 2005.
- [3] Research Triangle Institute, "The economic impacts of inadequate infrastructure for software testing," NIST Planning Report 02-3, May 2002.
- [4] D. Galin, "Software quality assurance: from theory to implementation," Harlow, England: Pearson Education Limited, 2008.
- [5] D. Dvorak, "NASA study on flight software complexity," AIAA Infotech Aerospace Conference and AIAA Unmanned, Unlimited Conference, 2009.
- [6] "DO-178C Software Considerations in Airborne Systems and Equipment Certification," RTCA, 12/13/2011.
- [7] E. Borger, E. Riccobene, J. Schmid, "Capturing requirements by abstract state machines: the light control case study," Journal of Universal Computer Science, pp. 597-620, 2000.
- [8] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, "Requirements specification for process-control systems," IEEE Transaction on Software Engineering, September 1994.
- [9] S. Rayadurgam, A. Joshi, M. Heimdahl, "Using PVS to prove properties of systems modelled in a synchronous dataflow language," International Conference on Formal Engineering Methods, November 2003.
- [10] J. Badger, D. Throop, C. Claunch, "VARED: Verification and analysis of requirements and early designs," IEEE Requirements Engineering Conference (RE14), Karlskrona, Sweden, 2014.
- [11] Y. Zhao, "K. Rozier, "Formal specification and verification of a coordination protocol for an automated air traffic control system," Science of Computer Programming Journal, volume 96, number 3, pp. 337-353, December 2014.
- [12] GitHub SpeAR [Online] <https://github.com/lgwagner/SpeAR>.
- [13] K. Gross, A. Fifarek, J. Hoffman, "Incremental formal methods based design approach demonstrated on a coupled tanks control system," IEEE 17th International Symposium on High Assurance Systems Engineering, 2016.
- [14] Avitas Systems—A GE Venture [Online] <http://www.avitas-systems.com/>
- [15] NASA Automated Contingency Management. [Online] <https://ti.arc.nasa.gov/tech/dash/pcoe/automated-contingency-management/>
- [16] P. Johnson-Laird, "Mental models: towards a cognitive science of language, inference, and consciousness," Harvard University Press, Cambridge, MA, 1983.
- [17] P. Johnson-Laird, "Mental models: the computer and the mind," Harvard University Press, Cambridge, MA, 1988.
- [18] P. Johnson-Laird, "Human and machine thinking," Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.
- [19] OWL Web Ontology Language Reference: W3C Recommendation 10 February 2004, available on-line at <http://www.w3.org/TR/owl-ref>.
- [20] W3C [Online] <https://www.w3.org>.
- [21] Semantic Application Design Language (SADL). [Online] <http://sdl.sourceforge.net/index.html>.
- [22] A. Crapo, A. Moitra, "Toward a unified English-like representation of semantic models, data, and graph patterns for subject matter experts," International Journal of Semantic Computing, Vol. 7, No. 3, 2013, pp. 215-236.
- [23] Xtext: Language Engineering for Everyone. [Online] <http://www.eclipse.org/Xtext/>.
- [24] "Safety critical software and systems research @ General Electric," Safe and Secure Systems and Software Symposium (S5), July 12-14, 2016. [Online] http://www.mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0845_Durling.pdf.
- [25] A. Crapo, A. Moitra, C. McMillan, D. Russell. "Requirements capture and analysis in ASSERT™," to appear in IEEE Requirements Engineering Conference (RE17).
- [26] P. Manolios, "Scalable methods for analyzing formalized requirements and localizing errors," United States Patent No. 9,639,450, May 2017.
- [27] H. Chamathi, P. Dillinger, P. Manolios, D. Vroon, "The ACL2 sedan theorem proving system.," TACAS, 2011, Springer.
- [28] M. Kaufmann, P. Manolios, J. Strother Moore, "Computer-aided reasoning: an approach," Kluwer Academic Publishers, 2000.
- [29] P. Manolios, D. Vroon, "Termination analysis with calling context graphs," Computer Aided Verification (CAV), Lecture Notes in Computer Science 4144, Springer, pp. 401-414, 2006.
- [30] P. Manolios, D. Vroon, "Interactive termination proofs using termination cores," *Interactive Theorem Proving*, July 2010, Springer LNCS 6172.
- [31] H. Chamathi, P. C. Dillinger, M. Kaufmann, P. Manolios, "Integrating testing and interactive theorem proving," ACL2 2011, EPTCS 70, pp. 4-19.
- [32] H. Chamathi, P. Manolios, "Automated specification analysis using an interactive theorem prover," FMCAD 2011, pp. 46-53.
- [33] H. Chamathi, P. C. Dillinger, P. Manolios, "Data definitions in the ACL2 sedan," ACL2, 2014.
- [34] D. Boren, "Management of test complexity for emerging safety critical control systems program," Air Force Office of Scientific Research Final Report, May 2006.
- [35] J. Offutt, A. Abdurazik, "Generating tests from UML specifications," 2nd International Conference on Unified Modeling Language, Fort Collins, CO, 1999.
- [36] L. Briand, Y. Labiche, "A UML-based approach to system testing," Software and Systems Modeling, pp. 10-42, 2002.
- [37] M. Sarma, D. Kundu, R. Mall, "Automatic test case generation from UML sequence diagrams," International Conference on Advanced Computing and Communications, 2007.
- [38] J. Rushby, "Automated test generation and verified software," Working Conference on Verified Software: Theories, Tools, and Experiments, Springer, Berlin Heidelberg, 2005.
- [39] Federal Aviation Administration, "Rationale for accepting masking MC/DC in certification projects," Certification Authorities Software Team, Position Paper, CAST-6, 2001.
- [40] Z. Awedikian, "Automatic generation of test input data for MC/DC test coverage," Soccer Lab, Ecole Polytechnique de Montreal.
- [41] "Improving the quality of complex control logic design using model verification", [Online] <http://www.matlabexpo.com/in/2013/proceedings/improving-the-quality-of-complex-control-logic-design-using-model-verification-and-validation-techniques.pdf>.
- [42] B. Prasad, D. Priyanka, V. Ramanathan, S. Ulka, "System and method for automatic generation of test data to satisfy modified condition decision coverage," United States Patent No. 8,612,938, December 2013.
- [43] GitHub Google Test. [Online] <https://github.com/google/googletest>.