Verification of Requirements for Safety-Critical Software

Paul B. Carpenter

Aonix

5040 Shoreham Place

San Diego, CA

619-457-2700

paulc@aonix.com

1. ABSTRACT

The purpose of this paper is to describe a methodology for the verification of safety-critical software. The methodology is implemented with use-case modeling notation of the Unified Modeling Language (UML). The methodology also contains techniques for creating requirements-based test cases from scenarios. The test cases are formatted for test scripts that exercised the software.

1.1 Keywords

Requirements Analysis, Requirements-Based Testing, Software Verification, UML, DO-178B, IEEE/EIA 12207

2. INTRODUCTION

Imagine walking on to an airplane for your next flight and you notice the following product warranty next to the entry:

Product Warranty

The software programs on this aircraft are sold "as is" without warranty of any kind, either expressed or implied. The entire risk as to the quality and performance of the software is with you.

The developer does not warrant that the functions contained in the program will meet your requirements or that the operation of the software will be uninterrupted or error free.

Would you get on the airplane? Probably not!

Fortunately, suppliers of safety-critical software, such as airplane control systems, are required to follow rigorous

software development standards that help ensure software quality.

This paper describes a process for developing verifiable software requirements using modeling. From the models, developers generate documents, review packets, traceability reports, requirements-based test cases, and design models. The test cases are used to create test scripts that exercise the implemented requirements.

3. SOFTWARE STANDARDS

Software development standards describe frameworks of life cycle processes using well-defined terminology. The standards provide guidelines for development and control of software and may be adapted and tailored to fit an organization's particular needs. Unless constrained by a contract each organization is allowed to establish its own procedures, methods, tools, and environments for the software development project.

Most process standards include the concept of software-requirements verification. The IEEE definition of verification includes:

- The process of determining whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase

- The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services, or documents conform to specified requirements[5].

Many industries have their own standards, such as RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification[8] for the avionics industry and IEEE/EIA 12207, Industry Implementation of International Standard ISO/IEC 12207[6] for the Department of Defense.

DO-178B: "… provides the aviation community with guidance for determining, in a consistent manner and with an acceptable level of confidence, the software aspects of airborne systems and equipment comply with airworthiness requirements." [9]

The software life cycle processes in DO-178B are organized into the following three groups: planning, development, and integral. The planning process defines

the plans that direct the activities of the development and integral processes, and describes how to coordinate those activities. The development process describes the activities that produce the software product and the integral process describes the activities that ensure correctness and control of the development activities. The integral process is performed concurrently with the development process.

4. SOFTWARE PLANNING PROCESS

The purpose of the software planning process is to define how the software product will be developed and certified. The planning process ensures the development of a product that satisfies the system requirements and one that conforms to the safety requirements.

The activities of the software planning process are described below.

4.1 Define Life Cycle Model

If not specified by the acquiring organization the planning organization chooses a life cycle model that is appropriate to the scope, magnitude, and complexity of the project. The activities of the development and integral processes are mapped into the development model. The detailed mapping includes the activities and tasks to be performed, when the activities will be performed, and the personnel responsible for performing the activities.

4.2 Define Software Life Cycle Environment

Standards require that planners define the methods that will be used by the developers.

The planners select a modeling language that will permit the capture and recording of all required and optional information. The example in this paper uses English text and the Unified Modeling Language (UML) to describe the requirements and design. "Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system's architecture. Using a standard modeling language such as UML, different members of the development team can unambiguously communicate the decisions to one another"[7].

The planners also select the tools that implement the plans. The example in this paper was developed with the Software through Pictures (StP) Lifecycle Desktop from Aonix with UML modeling, requirements-based test case generation, and Ada code generation capabilities. The StP Life Cycle Desktop has a standard DO-178B interface and is highly customizable. These tools give planners the capability to customize their selected life cycle model and all development methods, activities, documents, and tools for the project.

5. REQUIREMENTS PROCESS

The first and most important step in the development of quality software is the determination of the users' requirements. "It has been pointed out that up to 50 percent of the requirements for software development never get addressed in a proper manner in the industry. Specifically, in current practice, test requirements are missing from the requirements specification"[2].

The result of the requirements process is a requirements specification that communicates the requirements to the software designer, test designer, users, and all other interested parties. Information from the requirements specification is also used for project management and scheduling.

High-level software requirements are written as text statements and identified with unique key attributes. When a system requirement is allocated to software, the developer may reference the system requirements without re-writing it; thereby reducing work and the risk of introducing inconsistencies. But, new software requirements derived from the system architectural design must be written in detail.

The high-level software requirements should be stated in "objective" terms and developers should be careful not to introduce software design constraints as requirements since they may inhibit the development of an optimal design.

Requirements are identified using techniques such as interviews, focus groups, problem reports from similar applications, etc. Information sources may be purchasing organizations, users, managers, and maintainers. These people may be referred to as "stakeholders", because they have a stake in the completeness and correctness of the requirements.

Typically, requirements are recorded as text using a sentence structure containing the word "shall", and each requirement is paired with a unique identifier so it can be tracked and managed. The requirements in the example were taken from a Microwave Landing System (MLS) case study.

Requirement	Definition
R-031; SetMode	During active operation the Mode Select Buttons shall determine how the MLS is operating. The ON light is lit for the selected mode. The Mode is set by the pilot (AUTO/MANUAL) or by the receiver (TEST).
R-122; ManualMode	During Manual Mode the pilot shall have the capability to change the Channel Number, Azimuth, Back Azimuth, and Elevation.

5.1 Traceability With System Requirements

Each system requirement allocated to software must map to one or more software requirement.

Source Rqt ID	Software Requirement ID
S-2133	R-031
D-1254	R-122

Table 2: Requirements Trace Table

Traceability analysis, which can be automated, determines the completeness of the mapping of system requirements to software. Integrated tools can also generate software to system requirements traceability tables showing which software requirements are allocated to system requirements. Derived software requirements will not trace to system requirements.

5.2 Modeling Requirements

A significant problem with recording requirements as text is the difficulty of analyzing them for completeness, consistency, correctness, and testability. This problem is reduced with a tools-based modeling approach. The modeling language should be easy to learn and it should capture all required information, including additional information that may be required by safety-critical applications.

A requirements modeling approach takes the text requirements and maps them to graphical representations. When implemented with tools, requirement models can be automatically evaluated for completeness, correctness, and consistency. In the example, the modeling language is the use-case model notation of UML. Use-case notations include use-case diagrams and sequence diagrams. The use-case model notation is ideal for requirements-based testing activities when notation is restricted to model high-level requirements. The models should exclude details such as sub-systems, modules, and internal objects from scenarios and the models should be extended to capture information necessary for requirements-based test case creation. "The result of the modeling activities is a document that represents a thorough understanding of the problem the proposed software is intended to solve"[1].

An advantage of using use-case modeling rather than the more familiar data flow diagram modeling is the use-case model's capability to enhance incremental development. Developer's can move a use-case forward into design as soon as the model has been reviewed, accepted, and placed in the baseline.

5.2.1 Use-case Diagram
Use-case diagrams show a component organization of use-cases. The scope of the components can be very general (ie. The entire system), or the components can be based on the system architecture, or they can be based on software packages.

Regardless of the organization, use-case diagrams define the boundaries of the components and the component's high-level features.
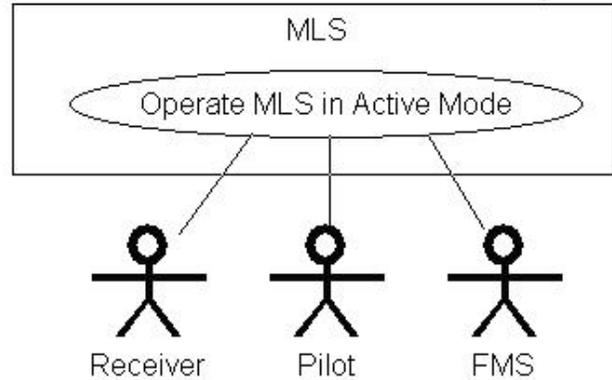

Figure 1 : Use-Case Diagram

A **system** (ex. MLS) represents the component that contains the features described by the use-cases.

A **use-case** (ex. Operate MLS in Active Mode) represents a high-level feature (function) and will map to one or more text requirements.

An **actor** (ex. Pilot) represents a user, software component, or external system that communicates with the use-case.

A **communication link** connects the actors to the use-cases. The communication link represents the existence of an interaction between the actor and the use-case. The link is non-directional and does not describe the direction or content of the interface. The details of communication links are modeled in sequence diagrams.

5.2.2 Sequence Diagram
Sequence diagrams model how the actors interact with the system to accomplish the required feature.

The operation of a use-case is described with one or more sequence diagrams. Each sequence diagram describes one scenario (operation) of the use-case. High-level sequence diagrams should be restricted to show only actors and the system component containing the use-case. Internal components should not be shown on high-level sequence diagrams.
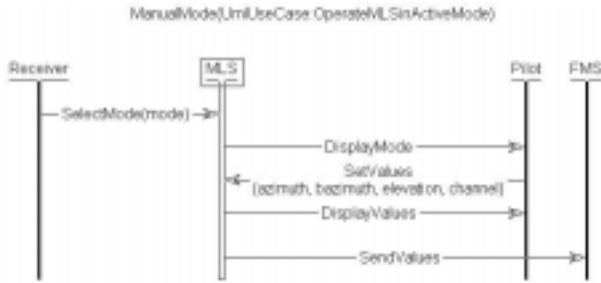
Figure 2: Sequence Diagram

A high-level sequence diagram contains a single **object** (ex. MLS) to represent the system. Internal details will be added during the Software Design Process.

The **actors** (ex. Receiver) on the sequence diagram must be the same as the communicating actors with the corresponding use-case. The "initiating" actor starts the scenario by sending the initial event to the system.

The **input events** (ex. SelectMode) represent external events that stimulates a response from the system. External data may be transferred to the system with input events.

The **output events** (ex. DisplayMode) represent the externally observable behavior of the system.

Even though UML allows for an unlimited number of sequence diagrams for each use-case, it may not be prudent to model all possible scenarios. Developers should start with a single "normal operation" sequence diagram and describe a scenario where all events occur at the appropriate time, all data is valid, and all logic is normal. Additional scenarios would be developed only when necessary to describe error correction requirements.

5.2.3 Requirements Model Traceability

Each software requirement maps to one or more use-cases. Requirements traceability tables document the links.

Use-case	Software Requirement ID
OperateMLSinActiveMode	R-031, R-122
OperateMLSinPassiveMode	R-032, R-234

Table 3 : Use-Case Trace Table

5.3 Modeling requirements for testability

The two objectives for testing safety-critical applications are the demonstration that the software satisfies its requirements and the demonstration that errors that could lead to unacceptable failures have been removed. Requirements-based testing has been found to be effective at revealing errors early in the testing process

Requirement models created with standard UML use-case notations are not robust enough to create requirements-based test cases, so the standard use-case model notation must be extended to allow developers to capture the additional information required to determine test cases.

The basic UML notation is extended to include testing attributes that are described in the following sections. Test cases can be automatically generated from "test-ready" use-case models.

5.3.1 Data Argument Definitions

The arguments associated with input events on sequence diagrams must be fully defined. Each argument is given a data type that contains user-defined or fixed values.

Argument	Data Type
mode	mode_type
azimuth	azimuth_type
bazimuth	azimuth_type
elevation	elevation_type
channel	channel_type

Table 4: Argument Definition

5.3.2 Data Type Definitions

Data type definitions must have one of the following type classes: string, integer, real, text, boolean, or unspecified. Each data type has valid and invalid subdomains. Valid subdomains are determined according to data type class. Invalid subdomains are "Out Of Type", "Below Bounds", "Above Bounds", "Out Of Bounds", and "Not In List".

5.3.2.1 Integer/Real Data Type Definitions

Automated tools generate sample values for integer and real data types from minimal definitions.

Type	Basic Attributes		Integer/Real Type Attributes		
Name	Class	Invalid Subdom.	Min Value	Max Value	Res.
azimuth_type	int	outofbounds	0	359	1
elevation_type	real	outofbounds	2.0	29.9	0.1
channel_type	int	outofbounds	500	699	1

Table 5: Numeric Data Types

When the minimum value, maximum value, and resolution value for a data type are defined five standard valid samples can be created. They are minimum, minimum + resolution, midpoint, maximum - resolution, and maximum. The "outofbounds" invalid subdomains in Table 5 causes the generation of two invalid samples: minimum - resolution and maximum + resolution.

5.3.2.2 String Data Type Definitions

Automated tools generate sample values for string data types from minimal definitions. Testers create a set of test values or they create a string definition using a regular expression.

Type	Basic Attributes		String Type Attributes		
Name	Class	Invalid Subdom.	Test Value	Test Value	Test Value
mode_type	string	none	auto	manual	test

Table 6: String Data Types

5.3.3 Logic Definitions

Logic definitions (constraints) correspond to externally observable conditions or business rules. Constraints are defined on the input events.

Scenario	Event	Constraint
AutoMode	SetMode	mode==auto
ManualMode	SetMode	mode==manual
TestMode	SetMode	mode==test

Table 7: Logic Definitions

5.4 Generating Requirements Test Cases

Test cases demonstrate the reliability of the individual features (use-cases) of the software. When executed, the test cases exercise the normal and abnormal operation of the feature. Test cases that evaluate the normal operation of a feature require that all input events occur in the appropriate sequence, all logic evaluates to true, and all data arguments have valid sample values. Test cases that evaluate the abnormal operation of a feature contain at least one of the following: an input event does not occur, some logic evaluates to false, or at least one data parameter has an invalid sample value.

5.4.1 Create Sample Values of the Inputs

Test case samples are created from five parts of the "test-ready" model. *Actions* correspond directly to the sequence diagrams, so a use-case will have the same number of test actions as sequence diagrams. Missing actions will be determined by reviewing the model and noting the scenarios that have not been modeled. *Information samples* are determined from the definitions of the arguments on input events. Samples are created for valid and invalid values of the information. *Logic samples* are generated from the condition definitions. Valid samples are created when the logic evaluates to "True" and invalid samples are created when the logic evaluates to False. *Event samples* are created from the input events. "Event occurs" is a valid sample, and "event does not occur" is an

invalid sample. *State samples* are used to "setup" and "cleanup" executable test scripts.

5.4.1.1 Input Events

Each input event on sequence diagrams has two sample values.

SetMode	SetValues
did_occur (valid)	did_occur (valid)
did_not_occur (invalid)	did_not_occur (invalid)

Table 8: Event Sample Values

5.4.1.2 Data Arguments

Each input data item has valid values based on their type definitions. For integer and real data types, five valid values are selected: reference, low_bound, high_bound, low_debug, and high_debug. Zero will be chosen as a sample if it lies within the boundaries. For strings and characters, samples are selected from a user-defined list, or generated from a regular expression definition.

mode	azimuth	bazimuth	elevation	channel
auto	0	0	2.0	500
manual	1	1	2.1	501
test	180	180	16.0	600
	358	358	29.8	698
	359	359	29.9	699

Table 9: Valid Sample Values

The invalid samples may be below-bounds, above-bounds, out-of-bounds, out-of-type, not-in-list, and abnormal. Invalid subdomains are optional. Table 10 shows the out-of-bounds values: below-bounds and above-bounds.

mode	azimuth	bazimuth	elevation	channel
none	-1	-1	1.9	499
	360	360	30.0	700

Table 10: Invalid Sample Values

5.4.2 Combine Samples Into Test Cases

The sample values are combined together to form set of test cases that provide adequate test coverage. Test cases 1-3 are screening test cases. The remaining test cases "probe" individual input dataitems by combining each of its samples with the reference samples of all the other input dataitems.

Test Cases			
	Sample Values	Probe	Test Type
1.	SetMode=did_occur mode=manual SetValues=did_occor	all at reference	Normal

	azimuth=180 bazimuth=180 elevation=16.0 channel=600		
2.	SetMode=did_occur **mode=auto** SetValues=did_occor azimuth=0 bazimuth=0 elevation=2.0 channel=500	all low bounds	Abnormal constraint evaluates to False
3.	SetMode=did_occur **mode=test** SetValues=did_occor azimuth=359 bazimuth=359 elevation=29.9 channel=699	all high bounds	Abnormal constraint evaluates to False
4.	…		

Table 11: Test Cases

6. TESTING PROCESS

DO-178B describes three levels of tests: low-level, software integration, and hardware/software integration[8]. Low-level tests test the low-level requirements; software integration tests test the interrelationships between requirements; and hardware/software integration tests test the high-level requirements.
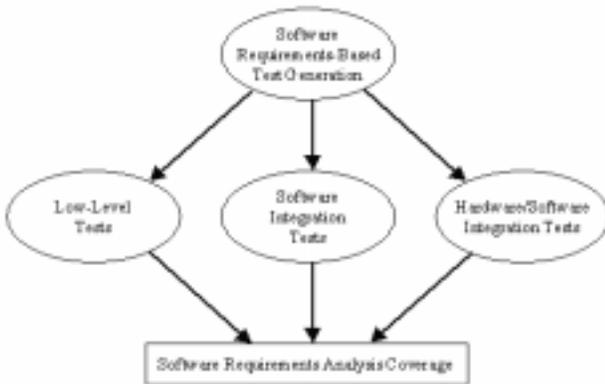

Figure 3: Requirements-Based Tests

The requirement-based test cases can be used for both low-level testing and integration-level testing, but there may be two test execution environments: one for low-level testing and the other for integration-level testing.

Low-level tests test individual features prior to component integration.

Testers begin testing individual components of the software as soon as the components are available from development. Low-level testing is not slowed down by delays in the development of external components, and the components may be thoroughly tested in a simulation environment.

The integration testers test the integration of the software components with full knowledge that individual components have been thoroughly tested.

6.1 Converting Test Cases To Test Scripts

A problem that occurs when implementing requirements-based testing is the necessity to translate language-independent test cases into language-dependent test scripts. Events in test cases must be converted into actions that cause the system to recognize the occurrence of the event and arguments must be into implementation-oriented names.

Tools can automatically generate executable scripts from the sequence diagrams, with the exception of the detailed actions associated with events. So, in most cases the testers need to manually create test scripts that correspond to the events.

6.2 Executing Low-level Tests

One approach for low-level testing is to use a test-harness-generation tool. Test-harness-generators create test scripts from the software under test. The test scripts read data files containing the input data and the expected output data. The data files can be generated from the test cases if the argument names are translated into source code names. This is accomplished with tables that map requirement arguments to source code names. When the data file is generated, the tool reads the correlation table and substitutes the argument with the variable. Table 12 contains a fragment of an Ada correlation table.

azimuth	1	position.azimuth
bazimuth	1	position.bazimuth
elevation	1	position.elevation
channel	1	channel_number

Table 12: Ada Correlation Table

Figure 4 contains a fragment of a low-level data file.

```
-- Begin TESTID:1

position.azimuth := 180;
position.bazimuth := 180;
position.elevation := 16.0;
channel_number := 600;

...
```

Figure 4: Low-Level Test Script

6.3 Executing Integration-Level Tests

One possible approach for integration-level testing is to create an integration test facility to simulate the safety critical system. The integration-level test scripts are executed in that environment. The input data values would be put on the data bus and the expected outputs would be read off the bus.

The integration data names correspond to data bus names. For safety-critical software, the input processing routines may be required to perform source selection algorithms, so a single requirement data name may have multiple bus data names.

azimuth	2	FC_L.wFED_0.azimuth FC_R.wFED_0.azimuth
bazimuth	2	FC_L.wFED_0.bazimuth FC_R.wFED_0.bazimuth

Table 13: Subsystem Correlation Table

Shown below is a fragment of an integration-level data file.

```
-- Begin TESTID:1

FC_L.wFED_0.azimuth:= 180;
FC_R.wFED_0.azimuth := 180;

FC_L.wFED_0.bazimuth:= 180;
FC_R.wFED_0.bazimuth:= 180;

...
```

Figure 5: Integration-Level Test Script

7. SUMMARY

The work of verifying requirements for safety-critical software is leveraged through the use of tools, which capture and analyze the requirements. The tools are able to automatically check the syntax, semantics, and testability of the requirements. Tools also generate documentation, joint review documents, operational profiles, and test cases. Tools also generate a manageable number of test scripts based on the intended use of the software.

Automation reduces work, and the level of automation described in this paper is made possible only when developers start by creating verifiable requirements models.

8. ABOUT THE AUTHOR

Paul B. Carpenter is the Director or Lifecycle Technology at Aonix. He is responsible coordinating the development and marketing of the Software through Pictures (StP) Life Cycle Desktop, which supports IEEE/EIA-12207, DO-178B, and other process life cycles.

Prior to his current position, he has held a variety of technical management, consulting and training positions with Aonix.

Previously, he was with Honeywell where he created an automated requirements-based testing environment that automatically generated test cases and test scripts for the Engine Indicators and Crew Alerting System (EICAS) software component for the Boeing 777 jetliner.

9. REFERENCES

[1] Cho, Chin-Kuei, Quality Programming, John Wiley & Sons, Inc, 1987, p 21.

[2] Cho, p 154.

[3] Douglass, Bruce, Real-Time UML, Addison-Wesley, 1998.

[4] Fowler, Martin, UML Distilled, Addison-Wesley, 1997.

[5] IEEE 729, IEEE Standard Glossary of Software Engineering Terminology, 1983, p 37.

[6] IEEE/EIA 12207, Industry Implementation of International Standard ISO/IEC 12207, 1998.

[7] Kruchten, Philippe, The Rational Unified Process, Addison-Wesley, 1999, pp 11-12.

[8] RTCA/D0-178B, Software Considerations In Airborne Systems And Equipment Certification, RTCA, Inc, 1992.

[9] RTCA/D0-178B, p 1.

[10] Texel, Putnam and Williams, Charles, Use Cases Combined with Booch OMT UML: process and products, Prentice Hall PTR, 1997, p 22.