Verification of Requirements For Safety-Critical Software

Paul B. Carpenter
Director, Life Cycle Technology

Aonix, 5040 Shoreham Place, San Diego, CA USA, 92122
Email: paulc@aonix.com; Tel: 602-816-0245; Fax: 602-816-0216

Abstract – *DO-178B, "Software Considerations in Airborne Systems and Equipment Certification" provides guidelines for the development and verification of software for airborne systems and equipment. These guidelines describe (1) objectives for life cycle processes; (2) descriptions of activities for achieving those objectives; and (3) descriptions of the evidence that indicate that the objectives have been satisfied.*

This paper describes a well-defined process for collecting and modeling requirements; generating requirements-based test cases; and executing requirements-based tests.

1 Purpose

The purpose of this paper is to describe a well-defined methodology for the development of safety-critical software using industry standards. The pertinent industry standards are for life cycle processes, development methods, modeling languages, and deliverable documentation.

The case study described in this paper was developed with "best-of-class" commercial tools. The tools have been integrated to support the development of safety-critical applications using the standards, processes, methods, and modeling languages described below.

2 Life Cycle Process Standards

Process 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.

Two common industry process standards are RTCA/DO-178B (*Software Considerations in Airborne Systems and Equipment Certification*) and IEEE/EIA 12207 (*Industry Implementation of International Standard ISO/IEC 12207*). IEEE/EIA 12207 contains clarifications, additions, and changes to the international standard for software life cycle processes, ISO/EIA 12207 (*Standard for Information Technology – Software life cycle processes*).

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."

IEEE/EIA 12207: "… establishes a common framework for software life cycle processes, with well-defined terminology, that can be referenced by the software industry." IEEE/EIA 12207 is a more generalized standard for commercial and Defense applications. In May of 1998, The United Department of Defense adopted IEEE/EIA 12207 as its life cycle process standard, replacing Mil-Std-498.

This paper will use the terminology of DO-178B and focus on the definition, analysis, and verification of requirements of safety-critical software.

2.1 DO-178B

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 processes that produce the software product. The software development processes are software requirements, software design, coding, and integration. All activities of the development organization are contained within these four processes. The integral process describes the activities that ensure correctness and control of the development processes and they are performed concurrently with the development processes.

A primary goal of DO-178B is to prevent errors in the software.

3 Software Planning Process

The purpose of the software planning process is to define how the software product will be developed. The planning process must ensure the development of a product that will satisfy the system requirements and provide a product that is consistent with airworthiness requirements. The activities of the software planning process are described below.

3.1 Define Life Cycle Model

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

The case study uses a life cycle model based on DO-178B. The focus of the case study is the software requirements process.

3.2 Define Software Life Cycle Environment

DO-178B requires that planners define the methods that will be used by the developers. The case study describes an incremental/iterative development method based on requirements modeling using use case and object-oriented modeling techniques.

Once the development method has been selected, the planners must then select a modeling language that will permit the capture and recording of all required and optional information. The case study uses English text and the Unified Modeling Language (UML) to describe the requirements.

Research has shown that UML is not sufficient to model requirements for safety critical applications, so the notations have been supplemented with text as necessary.

The planners must also select a programming language. In this case study, Ada was selected as the programming language.

After the life cycle model, development methods, modeling language, and programming language have been selected the planners must then select the tools.

The case study was developed with the Software through Pictures (StP) Life Cycle Desktop from Aonix. The StP Life Cycle Desktop has a standard DO-178B interface and it is highly customizable to provide planners the capability to implement the selected life cycle model, and all development methods, activities, documents, and tools specified during the planning process.

The following picture shows the top level of the StP Life Cycle Desktop.

The following tools were used to develop the case study. Company names are enclosed in "()".

(1) Software through Pictures (StP) (Aonix)
(2) DOORS (QSS)
(3) DocExpress (ATA)
(4) StP/UML (Aonix)
(5) Validator/Req (Aonix)
(6) ObjectAda (Aonix)
(7) TestDirector (Mercury Interactive)

3.3 Produce Software Plans

The information collected during the software planning process must be documented in a Software Development Plan. Plans generated during the planning process are:

- The Plan for Software Aspects of Certification
- Software Development Plan
- Software Verification Plan
- Software Configuration Management Plan
- Software Qualify Assurance Plan

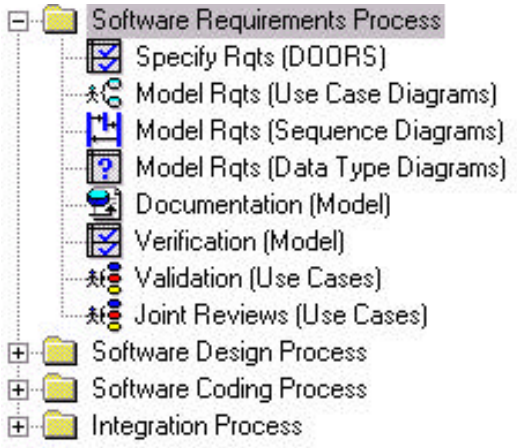
The developers must use the methods, tools, and programming languages that documented in the plans.

4 Software Requirements Process

DO-178B: "The software requirements process uses the outputs of the system life cycle process to develop the software high-level requirements. These high-level requirements include functional, performance, interface and safety-related requirements."

The StP Life Cycle Desktop organizes the development and integral activities for the software requirement process as specified in the Software Development Plan.

The development and integral activities are organized in a logical format to provide the developers with guidance as they perform their individual tasks.



4.1 Specify Requirements

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

The high-level software requirements should be stated in "objective" terms, and they should not contain software design details, unless they are pre-approved design constraints.

Constraints are restrictions placed on the software and a project decision is required to determine how the constraints will be implements. Developers should be careful not to introduce software design constraints as software 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 must contain a unique identifier so it can be tracked and managed.

The StP Life Cycle Desktop has a seamless integration with DOORS and the StP Requirements Table Editor. The table below shows one requirement captured with the StP Requirements Table Editor.

Requirement	Definition
E-01; DisplayTAT	The EICAS shall have the capability to receive as input Total Air Temperature (TAT) and generate as output the graphical display if the status of Total Air Temperature Status (TAT_status) is valid.

4.2 Traceability With System Requirements

Each system requirement allocated to software must map to one or more software requirement. Tools, such as the StP Requirements Table Editor, capture requirement traceability information, from which traceability tables are generated. Show below is the system to software requirements traceability table.

Source Rqt ID	Requirement Table	Requirement ID	Short Description
DISP-1254	EICAS	E-01	DisplayTAT

The StP Life Cycle Desktop also generates software to system requirements traceability tables showing the software requirements allocated to the system requirements. Derived software requirements will not trace to system requirements. Traceability analysis will determine the completeness of the mapping of system requirements to software.

4.3 Modeling Requirements

One major problem with recording requirements as text is the difficulty of analyzing them for completeness, consistency, and correctness. This problem is reduced with a tools-based modeling approach. The modeling language should be easy to learn and capture all information required by the requirement process.

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 case study, the requirements modeling language is the UML Use Case Model notation, which consists of a set of use cases and their sequence diagrams. The Use Case Model notation is very easy to learn and with the following restrictions the models can represent both high-level system and high-level software requirements for safety-critical systems. The restrictions are (1) implementation details such as sub-systems, modules, and internal objects must be excluded from scenarios, and (2) the models must be extended to capture information for requirements-based test case generation.

The allocation of system requirements to software will be simplified if the system requirements have been represented with use case models. The system use case model is modified to include system architectural design details. New software use case models must be created to map to the derived requirements.

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 it is defined and placed in the baseline.

In the case study, StP/UML was used to develop the use case model.

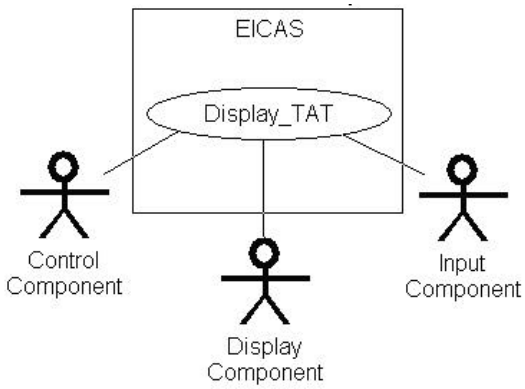
4.3.1 Use Case Diagram

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 categories. Categories are software architectural design components.

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

The use case diagram contains the following symbols:

- Use case - oval
- Actor – stick figure
- Communication link – line
- System – box



A **use case** (ex. Display_TAT) represents a high-level feature (function) and should map to one or more text requirements.

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

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 the communication link will be modeled in the sequence diagram.

The **system** (ex. EICAS) names the component and establishes the boundary between the internal and external features. In this example the system name represents a software component which is a non-trivial, nearly independent, and replaceable part of the software.

4.3.2 Sequence Diagram

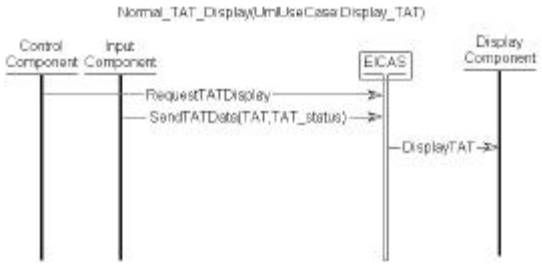
Sequence diagrams model how the actors interact with the system to accomplish the required feature.

The operation of a use case is described with one or more sequence diagram. 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.

The diagram contains the following symbols:

- System object –vertical box
- Actor –vertical line
- Input event – horizontal line from an actor
- Output event – horizontal line to an actor



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

The **actors** (ex. ControlComponent) 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. RequestTATDisplay) 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. DisplayTAT) 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.

4.3.3 Requirement Model Traceability

Each software requirement must map to one or more use cases. In the case study, StP/UML creates a link between use cases and text requirements recorded in DOORS or the StP Requirements Table. Requirements traceability tables are generated from the links.

Use Case	Requirement Table	Requirement ID	Short Description
Display_TAT	EICAS	E-01	DisplayTAT

4.4 Modeling requirements for testability

Testing safety-critical applications has two objectives: demonstration that the software satisfies its requirements and demonstration that errors, which could lead to unacceptable failures, have been removed.

Requirements-based testing has been found to be effective at revealing errors when test cases include both valid and invalid tests.

Two types of test cases can be determined from the requirements. They are feature test cases and operational load test cases.

Feature testing demonstrates the reliability of individual features (use cases) of the software. When executed, features test cases will 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 parameters 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.

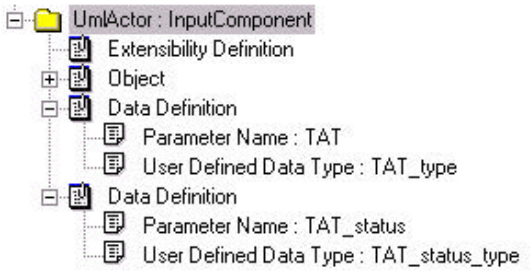
Operational load test cases focus on the operational load on the software. Load test procedures are created for each operational mode of the system to test the average and maximum load. An Operational Profile, which is a picture or model of expected field use of the software product, can be calculated for each operational mode.

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 the feature and operational load test cases.

Requirements-based test cases and operational profiles can be automatically generated from "test-ready" use case models with Validator/Req. The test-ready extensions are described in the following sections.

4.4.1 Data Parameter Definitions

The parameters associated with input events on sequence diagrams must be fully defined. In order to develop test cases for the requirements the testers must be able to create valid and invalid sample values for each input argument. Each data definition must have a data type which may be user-defined, pre-defined, or fixed. User-defined data types must also be defined completely.



In Validator/Req, the data names are attached to the source actor.

4.4.2 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 a valid subdomain and an invalid subdomain. For valid subdomains, developers enter valid sample values, or boundary definitions or regular expressions. For invalid subdomains, developers establish a criteria for the invalid samples such as: out_of_type, below_bounds, above_bounds, out_of_bounds, not_in_list, and abnormal.

In the case study the Data Type Editor of Validator/Req is used to define sample test values.

Type	Basic Type Attributes		Integer and Real Type Attributes		
Name	Type Class	Invalid Subdom.	Min Value	Max Value	Resolution
TAT_type	real	OutOfBounds	-80.0	100.0	0.125
TAT_status_type	string	None			

Testers, creating test cases manually, can use the information to determine the sample test values, and then combine the samples together into test cases. With some simple rules an organization can enforce a standard for sample data value selection and test case creation.

Validator/Req uses the data type definitions to automatically generate sample test values for all of the data definitions.

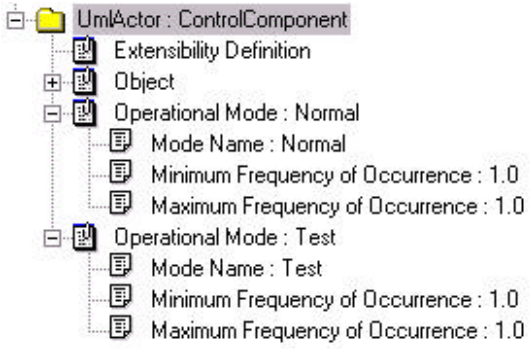
4.4.3 Logic Definitions

Logic definitions (constraints) correspond to externally observable conditions or business rules. Constraints are defined on the input events. In a sequence diagram, a constraint must evaluate to true before the scenario can proceed beyond the input event.

StP/UML uses the logic definitions to generate a set of test cases that "exercise" the scenario when all of the logic conditions evaluate to true; and a set of test cases that "do not exercise" the scenario when at least one of the logic conditions evaluates to false.

4.4.4 Actor Operational Mode

Actor mode information is used to calculate the operational profile. For each mode of operation, the developer determines the minimum and maximum concurrent occurrences of each actor over a pre-determined period of time.

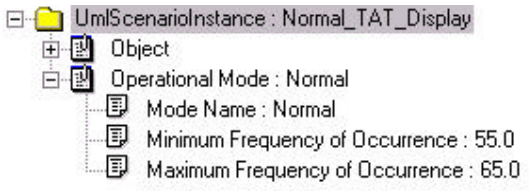


In the case study, StP/UML was used to capture the operational mode information for each actor.

4.4.5 Scenario Operational Mode

Scenario mode information is also used to calculate the operational profile. For each mode of operation, the developer determines the minimum and maximum concurrent occurrences

of each scenario actor over a pre-determined period of time.



In the case study, StP/UML was used to capture the operational mode information for each scenario.

4.5 Reviewing and Accepting Requirements

The most probable defects in requirements are well known. Requirement defects can be group as missing, wrong, or extra. Wrong requirements result from ambiguous words and phrases, incomplete statements, inconsistent features, untestable features, untraceable features, and undesirable design constraints.

The method for finding the most probable defects in requirements is the Joint Review. It's important to get reviewers from multiple sources, and the reviewers should be the stakeholders. System stakeholders are:

- Buyers (acquirers) – they acquire the product
- Builders – they specify, design and code the product
- Checkers – they verify and validate the product
- Marketers – they market and sell the product
- Field engineers – they support the product
- Users – they use the product

As the stakeholders approve the requirements, they are added to the baseline.

4.6 Documenting Requirements

From baselines, tools generate the following requirement process documents:

- Verification reports
- Contractual documents
- Requirements test cases
- Traceability tables
- Operational profiles
- Joint review reports

When an incremental development method is used the documents will be incomplete until very late in the project.

4.6.1 Generating Verification Reports

Verification reports provide information about the test-readiness of the requirement model. The report will check for the following:

- Is the use case model complete
- Do all use cases map to requirements
- Are all data arguments defined
- Are all data types defined
- Are all mode occurrences defined for actors and scenarios

4.6.2 Generating Requirements Documents

Software Requirement Specifications (SRS) are generated from text requirements and the use case model.

The case study uses DocExpress in conjunction with StP to provide developers with the capability to create Word documents from pre-defined templates containing links to the StP database. The SRS is "built" by DocExpress only when necessary.

4.6.3 Generating Requirements Traceability

Traceability must be documented between the system and software requirements. Two tables are generated: System → Software and Software → System.

Requirement ID	Short Description	Source Document	Source Rqt ID
E-01	DisplayTAT	SCD	DISP-1254

The traceability between the software requirements and the use cases must also be documented. Two tables are generated: Software Requirements → Software Model and Software Model → Software Requirements.

Requirement ID	Short Description	Use Case
E-01	DisplayTAT	Display_TAT

4.6.4 Generating Test Cases

Validator/Req generates a minimal set of test cases from the test ready use cases. A three-step process is used

(1) extract test information
(2) generate samples for each data input
(3) combine the samples into test cases

When a use case is "test-ready", the test case information is extracted from the StP database and sent to the test case generator.

The test case design process is started with the generation of sample input values. Each input event has a valid value “did_occur” and an invalid value “did_not_occur”. 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.

```
TAT Valid     TAT_status Valid
 values           values

   -80.0           valid
   -79.875         invalid
     0.0           npr
    10.0
    99.875
   100.0
```

The invalid samples may be below-bounds, above-bounds, out-of-bounds, out-of-type, not-in-list, and abnormal. Invalid subdomains are optional.

```
TAT Invalid     TAT_status Invalid
 values              values

   -80.125           none
   100.125
```

(3) 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. In the remaining test cases, the individual input dataitems are “probed” by combining each of its samples with the reference samples of all the other input dataitems.

```
Test Cases

1.      10.0        valid    screening
2.      -80.0       invalid
3.      100.0       npr

4.      –80.0       valid    debug
5.      0.0         valid    probe
6.      99.875      valid
7.      100.0       valid

8.      –80.125     valid    invalid
9.      100.125     valid    probe
…
```

Validator/Req stores the test cases in a database for use in test planning, coverage analysis, and test script generation.

4.6.5 Generating Operational Profiles

An operational profile lists the complete set of operations and their probabilities of occurrence for each mode of operation. Two tables will be generated for each operational mode (1) average operational load and (2) maximum operational load.

Initiator Actor Name	Scenario Name	Scenario Occurrences	Frequency per 1 minute	Probability of Occurrence
Control Component	Normal_TAT_Display	60.000	60.000	1.000
TOTAL			60.000	1.000

Initiator Actor Name	Scenario Name	Scenario Occurrences	Frequency per 1 minute	Probability of Occurrence
Control Component	Normal_TAT_Display	65.000	65.000	1.000
TOTAL			65.000	1.000

Operational profiles assist with the requirements-based testing process in two ways.
(1) Prioritizing feature testing
(2) Generating load test procedures

The test case generation process creates the minimum number of test cases to adequately test a feature, but typically there are not enough resources to execute all of the test cases. The operational profile can be used to identify “high occurrence” and “low occurrence” features. “High occurrence” features should be thoroughly tested with all of the requirements-based test cases. The number of test cases used to test “low occurrence” features can be reduced without risk to software reliability.

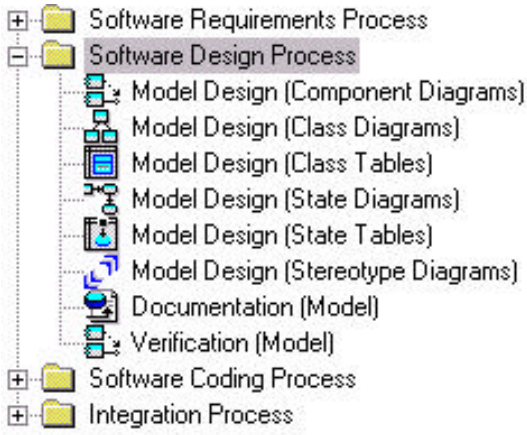
The operational profiles are used to generate test procedures that test the average and maximum operational load on the software. To reduce work

the feature test cases can be reused in the load tests.

5 Software Design Process

Discussion of the Software Design Process is beyond the scope of this paper.

The StP Life Cycle Desktop supports software design using object-oriented design techniques with UML design notations.



6 Software Coding Process

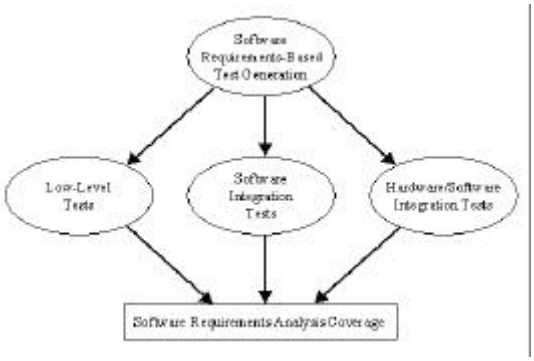
Discussion of the Software Coding Process is beyond the scope of this paper.

The StP Life Cycle Desktop supports code generation in C++, Java, and Ada95 from software designs, as well as links to programming environments.

7 Integration Process

The StP Life Cycle Desktop supports software integration.

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



7.1 Generating Test Scripts

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

Low-level tests use “feature test cases” and are conducted to test individual features prior to component integration.

Feature testers are able to test components of the software as soon as the components are available from development. Low-level testing is not slowed down by delays in the external components, and the software components may be thoroughly tested in a software simulation environment long software integration begins.

Integration tests use “load test procedures” to test the software after component integration.

After software integration, the integration testers test the integration of the software components with full knowledge that individual components have been thoroughly tested. The integration testers focus on testing the load on the software during each operational mode.

7.1.1 Converting Test Cases To Test Scripts

A problem that occurs when implementing requirements-based testing is translating 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 data parameters must be converted from requirements-oriented names into implementation-oriented names.

Tools can automatically generate executable scripts from the sequence diagrams, with the exception of the detailed actions associated with events. In most cases the testers will need to manually create test scripts that correspond to the events. If an event is re-used in the use case model, the test script for that event can be re-used also.

Converting requirement data names into implementation data names is solved by the creation of correlation tables. One set of tables maps requirement data names to low-level data names, and another set of tables maps requirement data names to integration data names. Since Ada was the application programming language, the application data names needed to be Ada variables.

Ada Correlation Table		
TAT	1	TOTAL_AIR_TEMP.data
TAT_status	1	TOTAL_AIR_TEMP.status

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.

Subsystem Correlation Table		
TAT	2	FC_L.wFED_0.tat FC_R.wFED_0.tat
TAT_status	2	FC_L.wFED_0.tatpvb FC_R.wFED_0.tatpvb

7.2 Test Execution

In most safety-critical projects, test execution is performed with proprietary tools.

7.2.1 Executing Low-Level Tests

One possible approach for low-level testing would be to create a simulation environment where test scripts run on a PC while the software under test runs on the target processor with memory shared between the two computers. The test scripts will be able to probe the input data and read the output data.

Shown below is a fragment of an application-level test script.

```
-- Begin TESTID:1
put_line (results_file, “TESTID: 1”);
put_line (results_file, “Reference tests”);

TOTAL_AIR_TEMP.data := 10.0;

TOTAL_AIR_TEMP.status := VALID;

...
```

7.2.2 Executing Subsystem-Level Tests

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

Shown below is a fragment of an integration-level test script.

```
-- Begin TESTID:1
put_line (results_file, “TESTID: 1”);
put_line (results_file, “Reference tests”);

FC_L.wFED_0.tat:= 10.0;
FC_R.wFED_0.tat := 10.0;

FC_L.wFED_0.tatpvb := VALID;
FC_R.wFED_0.tatpvb := VALID;

...
```

8 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 create verifiable requirements models.

9 About the Author

Paul B. Carpenter is the Director or Life Cycle 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. The results of his work were presented at CASE World in 1992 and 1993, and the Software Testing and Reliability Conference in 1993.