TRIAKIS CORPORATION

The Use of a Virtual System Simulator and Executable Specifications to Enhance Software Validation, Verification, and Safety Assurance

Final Report

for the

NASA Office of Safety and Mission Assurance
Software Assurance Research Program

Research Initiative 583

Ted Bennett, Principal Investigator

Triakis Corporation
Ted.Bennett@Triakis.com

Paul Wennberg, Principal Investigator

Triakis Corporation
Paul.Wennberg@Triakis.com

Table of Contents

Abstract

The root causes of the majority of software defects discovered during the integration test phase of an embedded system development project have been attributed to errors in understanding and implementing requirements. The independence that typically exits between the system and software development processes provides ample opportunity for the introduction of these types of faults. This research project has shown a viable method of verifying object software using the same tests created to verify an executable specification-based system design from which the software is developed. If the object software passes the same tests used to verify the system design, it can be said that the software has correctly implemented all of the known system requirements. This method enables the discovery of functional faults prior to the system integration test phase of a project. Previous research has shown that finding software faults early in the development cycle not only improves software assurance, but also reduces software development expense and time.

1. Introduction

The root causes of the majority of software defects discovered during the integration test phase of an embedded system development project have been attributed to errors in understanding and implementing requirements. These may be the system and/or the software requirements. We believe that this is largely a result of the independence that exists between the requirements development and the software development processes.

In contemplating ways to meaningfully connect the two processes, we realized that one viable way of doing so would be at the verification test level. In other words, create one set of tests that can be used to verify both the system design and the executable software itself.

In order to do this, the system design must be developed in a form in which it is completely testable. Executable specifications (ES) fill this need since they support testability while providing a means of expressing requirements in an unambiguous form.

Maintaining test consistency from system verification through software verification assures that software correctly implements all known and tested system requirements. This report shows how system/software test consistency can be maintained by combining the ideas of embedded system design & test using ES', and a virtual environment simulator capable of running and testing embedded executable software.

The research effort described in this report was conducted under a grant from the NASA Office of Safety and Mission Assurance – Software Assurance Research Program. Co-Principal Investigator, Paul Wennberg developed the simulation tool used in this investigation.

Our core tenet is that maintaining test consistency from system verification through SW verification assures that SW correctly implements all known system requirements. Hence, the primary objective of our project is to evaluate the viability of maintaining test consistency between system-level, and software-level verification.

2. Background

2.1. Root causes of software faults

In 1992, Dr. Robyn Lutz conducted an analysis for the Jet Propulsion Laboratory (JPL) to determine the root causes of the 387 software defects discovered during the integration test phase of the Voyager and Galileo spacecraft development efforts. The software controlling these spacecraft is distributed among several embedded computers with roughly 18,000 and 22,000 lines of source code respectively. Lutz reported that the programming faults discovered on the two projects are distributed as shown in Figure 1.

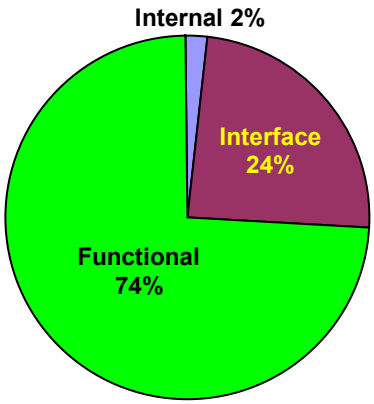

Figure 1. Combined fault distribution

The fault classifications given in Figure 1 are defined as Functional faults, Interface faults, and Internal faults. Functional faults comprise the following three subclasses:

a. Operating faults: omission of, or unnecessary operations;
b. Conditional faults: incorrect condition or limit values; and
c. Behavioral faults: incorrect behavior, not conforming to requirements.

Interface faults are those related to interactions with other system components such as transfer of data or control. Internal faults are defined as coding faults internal to a software module

Examining the root causes, Lutz reported that functional faults were predominantly caused by errors in understanding and implementing the requirements while interface faults principally resulted from inadequate communication between development teams (rather than within individual teams) [1].

The conclusions of the JPL report point to the need for improved focus in the following areas:

- Interfaces between the software and the system domains
- Identification of safety-critical hazards early in the requirements analysis
- Use of formal [and unambiguous] specification techniques…
- Promotion of informal communication among teams
- Keeping development & test teams apprised of changes to requirements
- Inclusion of requirements for "defensive design"

The JPL report findings are echoed in reports of numerous other researchers. For example, Thompson asserts that it is imperative for an embedded software system requirements specification to rigorously capture the interfaces and communication between the software and its embedding environment [2]. More than 50% of all system failures can be traced back to the requirement specifications [1] [3] [4]. Specifying correct software behavior can be as difficult and error-prone as writing the software itself [4] [5].

Consider a few of the plethora of requirements-related opportunities where problems may be introduced:

- Assumptions and ambiguities in the interpretation of customer descriptions of desired system behavior
- The difficulty in fully understanding the real-world environment in which the system will interact
- The difficulty in anticipating all of the possible modes and states that the system may encounter
- The difficulty in thoroughly validating and verifying the requirements
- Capturing an accurate, unambiguous representation of the requirements in a written document
- Misinterpretation of requirements by the system & software designers
- The difficulty in verifying that the design has correctly implemented the requirements

We generally cannot know at the onset of a project if we have accurately modeled the real-world system behavior. As a project advances, however, so does our understanding of the system. Additional faults may be introduced as a result of inadequate communication of system model refinements to the software development teams as a project progresses. In order to be more effective at creating software with a high level of assurance, it is clear that we must reduce the number of errors attributable to misunderstanding & misimplementing requirements, as well as improve communication between development teams.

One way to improve our ability to develop and test system designs in a more realistic manner is through the creation of unambiguous executable requirements specifications. In order to promote a broad understanding of the system requirements among development teams, there must exist an easy means of distributing the system design in a form that may be used by the software developers to verify that their code correctly implements the specified system functionality. Further, when system updates are generated, they must be distributed to the development teams so that software created prior to the update may be retested to verify that it meets the revised system design requirements.

2.2. Virtual environments: A promising approach long overdue.

In discussing the path to improving software engineering for safety, Lutz asserts that driven by the enthusiasm of industrial users, the use of virtual environment (VE) simulations to help design, test, and certify safety-critical systems is on the horizon. For software engineers, virtual environments offer a powerful means of integration and

systems testing. Lutz also points out how executable specifications enable the systems engineer to exercise specified safety requirements to ensure that they match the intent and the reality. Further, they allow the exploration of assumptions and help elicit latent requirements that may affect safety [6].

We assert that the use of executable specifications in the system design process affords a unique opportunity to connect the system and software development processes. Designing a complete system (or subsystem) using executable specifications allows the system verification to be performed empirically, with less reliance on analysis.

If the tests created to verify the system design could be used to verify the software developed to implement the functionality specified in the executable specifications, then the two processes can be effectively connected. When the software passes the system verification tests, then it has correctly implemented the known, documented, and tested requirements.

3. Methodology

The methodology we used is straightforward, adhering to these four essential steps:

a. Create a simplified ES-based system-level simulation of a suitable host system. This system-level simulation must comprise multiple ES'; including one embedded controller ES from which a hardware and software design are developed.

b. Write a suite of tests to verify and validate (V&V) the system design.

c. Simulate the controller hardware and write control software based on the functionality specified in the controller ES. We refer to the controller hardware simulation loaded with the executable object software as a detailed executable, or DE.

d. Substitute the DE for the controller ES in the system-level simulation and rerun all of the unmodified system tests developed in step 'b'. Debug the software and evaluate the test results.

3.1. Create ES-based simulator

To pursue our objectives, we created a simplified simulation of Shuttle Remote Manipulator System (Robotic Arm), or RMS, as a framework for implementing our ideas. The RMS simulator design is loosely based on descriptions of the real-world counterpart found at the NASA shuttle reference web site [7], a NASA news reference web site [8], and a published space shuttle operator's manual [9].

Our approach to executable specification-based (ES) simulators differs from other ES tools in several ways, but most importantly in the way ES parts are bounded. Every ES part in our system has a one-to-one correlation with a real-world counterpart. Each ES part is bounded exactly like its real-world counterpart i.e., all inputs and outputs are the same as those of the parts they simulate.

At the ES level, communication protocols, message traffic, data throughput, display formats, human factors, control algorithms, interaction between electrical, hydraulic, mechanical systems and all other system-related issues can be conceived, tested and validated. Because parts in this VE are bounded and interconnected analogously to real-world hardware, we refer to our system-level simulations as virtual system integration laboratories, or VSILs.

Figure 2 shows a simplified system-level diagram of the ES-based shuttle RMS that was used as

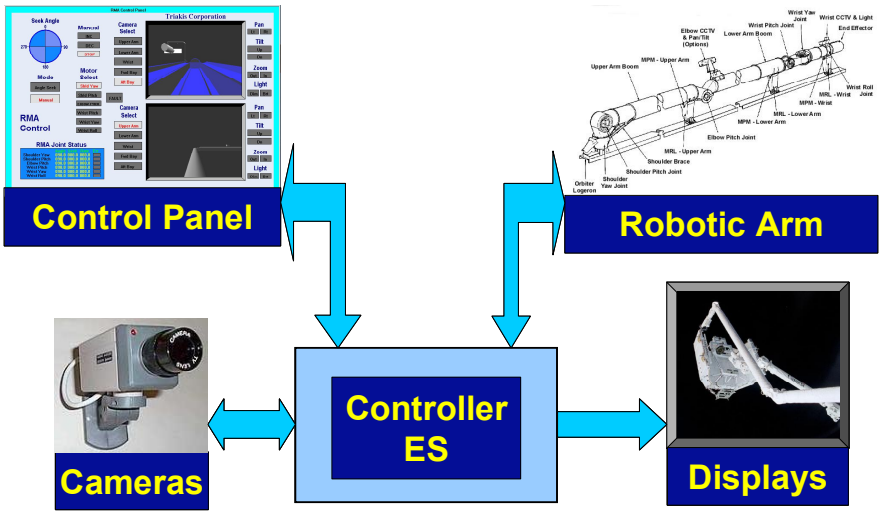

Figure 2. ES-based simplified shuttle RMS system-level diagram

a test vehicle for our research. The system essentially consists of an operator control panel, the remote manipulator arm (RMA) itself, five video cameras, two video display monitors, and an embedded controller.

Our control panel is a simplified version of the real-world panel. It enables the selection and control of each RMA joint on an individual basis. Each joint may be operated individually in manual or angle-seek mode. Manual mode allows the operator to start and stop the joint motor at will while angle-seek mode allows the operator to input a destination angle and the controller will activate the joint motor automatically until the joint reaches the desired angle. The angles of all six RMA joints are displayed on the control panel.

Video from any of the five cameras may be selected for display on the two video display monitors. Three of the cameras are located on the RMA upper arm, lower arm, and wrist while the other two are positioned at the fore and aft ends of the shuttle cargo bay. Camera video is digitized, compressed and sent to the controller part via the AFDX data bus. Figure 3 shows an image of our simulated RMS operator control panel.

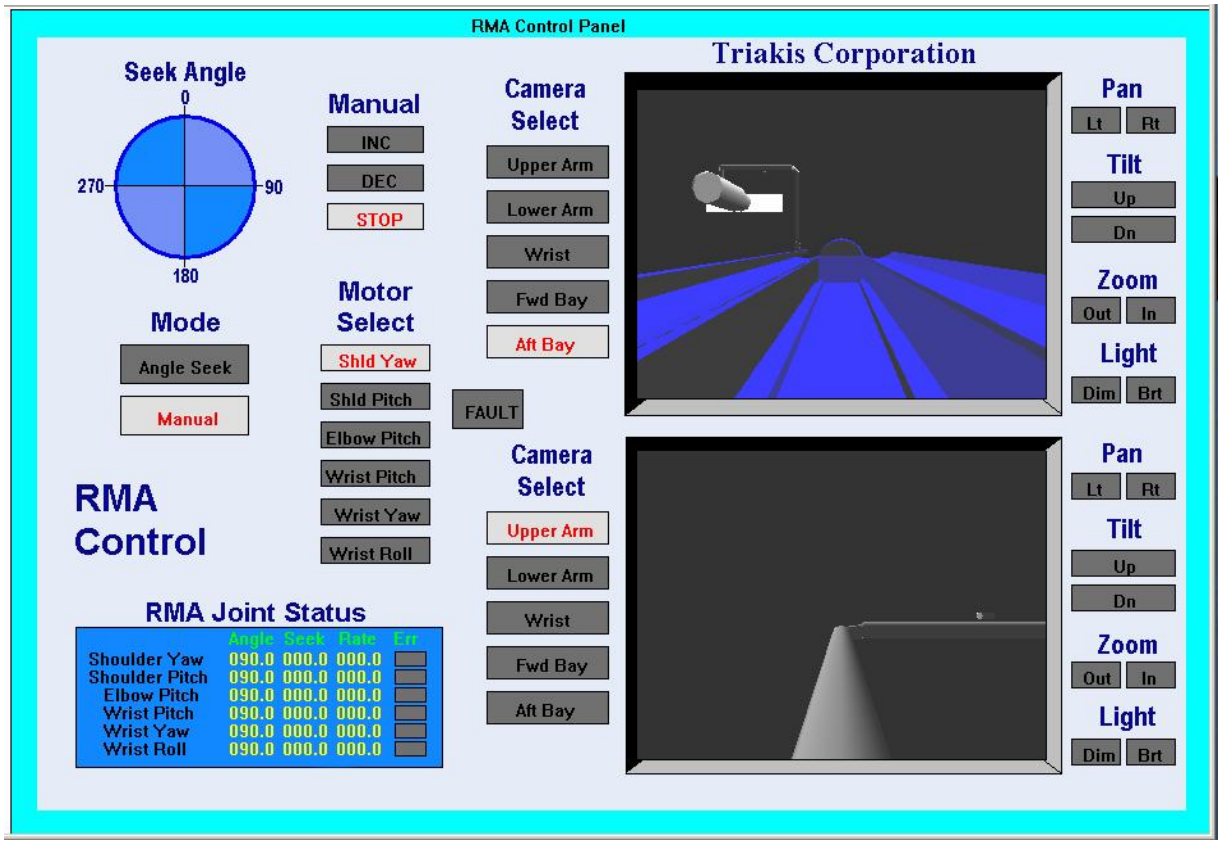

Figure 3. Simulated operator control panel ES

The simulated RMA ES has six motor-controlled joints like its real-world counterpart. All commands to the joint motors are issued from the controller part via a simulated avionics full-duplex Ethernet (AFDX) bus. Commands are converted to motor drive voltages on interface cards located at each joint.

Each motor is connected through a reduction gearbox to both a shaft encoder, and the joint itself. The shaft encoder output is digitized by the interface card and converted to an AFDX signal for transmission to the controller part. An image of a satellite payload is attached to the RMA end effector as a visual aid.

Figure 4 shows the SRMS system block diagram. The RMS Computer implements the controller ES part shown in Figure 2. The RMS Computer communicates with the cameras and the RMA via AFDX serial communications buses. Communications with the control panel are accomplished via a modified serial peripheral interface (SPI)

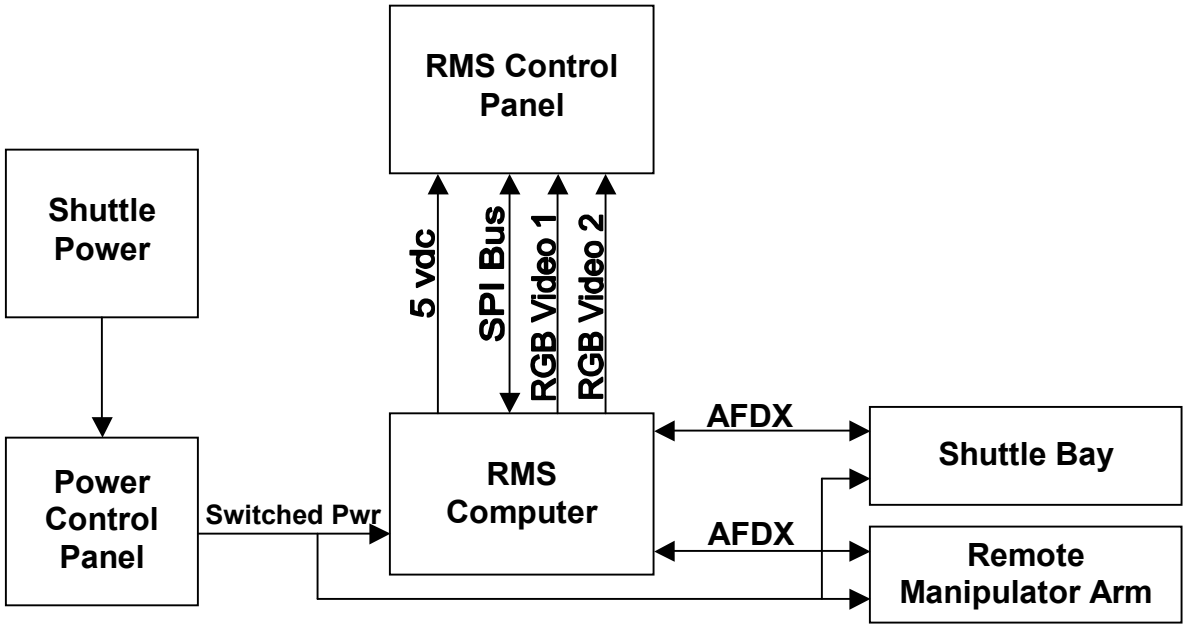

Figure 4. RMS Control Computer ES

industry standard data bus. Digitized camera video is converted to RGB video and sent to the control panel for presentation on two video display monitors.

3.2. Write suite of tests to V&V system design

Next, we created a suite of tests to exercise the system design in a manner closely mirroring real-world scenarios. Table 1 shows an example of a test written to verify that an RMA joint moves at the specified slew rate under manual control.

The positive joint velocity test shown in Table 1 begins by printing the test name using a pointer to the name of the RMA joint whose motor control is being tested. The next command activates the corresponding "Motor Select" button on the control panel part. Following a half second delay the "Manual Mode" button on the control panel is selected. After another brief delay the test name, "Positive Velocity" is printed to the log file.

Next, the control panel "INC" (increment) button is activated and then, after a short delay to allow for both the motor speed to stabilize and the joint velocity to be read by the controller and displayed on the control panel, the velocity of the low speed (output) gearbox

Table 1. Verification test for joint velocity

```
fprintf(resfp, "Testing %s Control\n\n", motorStr[mot]);

RMS_Ctrl_Panel->SetMotor(mot);
eq.Delay(0.5);
RMS_Ctrl_Panel->SetMode(MANUAL_MODE);
eq.Delay(0.5);

fprintf(resfp, "\nPositive Velocity\n\n");

RMS_Ctrl_Panel->SetCommand(INC);
eq.Delay(2.0);
actual_vel = gears[mot]->GetLowSpeedVelocity() * 180.0 / pi;
display_vel = RMS_Ctrl_Panel->GetMotorData(mot, 2);
expected_vel = (600.0 / 500.0) * (1.0 / 60.0) * 360.0;

fprintf(resfp, "  Actual value: %lf\n", actual_vel);
fprintf(resfp, "  Display value: %lf\n", display_vel);
fprintf(resfp, "  Expected value: %lf\n", expected_vel);
err_range = 2.0;
fprintf(resfp, "  Error range: +- %lf\n", err_range);
fprintf(resfp, "Verify Actual Value\n");
if(fabs(actual_vel - expected_vel) < err_range)
  fprintf(resfp, "**** PASSED\n");
else
  fprintf(resfp, "**** FAILED\n");
fprintf(resfp, "Verify Display Value\n");
if(fabs(display_vel - expected_vel) < err_range)
  fprintf(resfp, "**** PASSED\n");
else
  fprintf(resfp, "**** FAILED\n");
```

shaft is read. The value of the joint velocity being displayed on the control panel is read and then both velocity numbers are compared with the expected value within the allowable tolerance. The remainder of the test determines the pass/fail status of both the actual and displayed joint velocities, prints the results to the log file, and activates the control panel button to stop the joint motor. Control of all RMA joints are tested in both manual and angle-seek modes in a similar manner.

Each camera part has motor-controlled pan, tilt, & zoom capability so that each camera may be pointed in the desired direction. This is facilitated through camera control buttons on the RMS control panel and tested in the same way as the RMA joint responses are tested.

Table 2 shows a small segment of the test results log file generated during a system-level test. The segment shown gives the results for the RMA shoulder yaw control positive velocity and positive angle seek test sets. Each set verifies the following two functions:

a. The joint behaves as specified, and
b. The displayed value correctly indicates the actual joint angle and velocity.

Table 2. System test results segment

```
Testing Shoulder Yaw Control


Positive Velocity

  Actual value: 6.963920
  Display value: 7.000000
  Expected value: 7.200000
  Error range: +- 2.000000
Verify Actual Value
**** PASSED
Verify Display Value
**** PASSED

Positive Angle Seek

  Actual angle: 95.353984
  Display angle: 95.000000
  Expected angle: 96.587127
  Error range: +- 2.000000
Verify Actual Angle
**** PASSED
Verify Display Angle
**** PASSED
```

A total of 131 tests have been written to partially verify our system design. While the focus of the tests is to verify the functionality specified in the embedded RMS Computer ES, much of the rest of the system is also verified as a result of this environment-driven, end-to-end approach to testing. For example, testing the joint control behavior also exercises basic functionality of the motor, gearbox, angle encoder, and interface control parts.

A number of tests have been written to verify the specified behavior of the RMS Computer in response to subsystem part failures. These tests induce the various motor controller parts to report fault status to the RMS Computer and verify that the correct control panel fault indicator is lit along with the master fault lamp. The tests then clear the fault and verify that the computer extinguishes the associated control panel indicator.

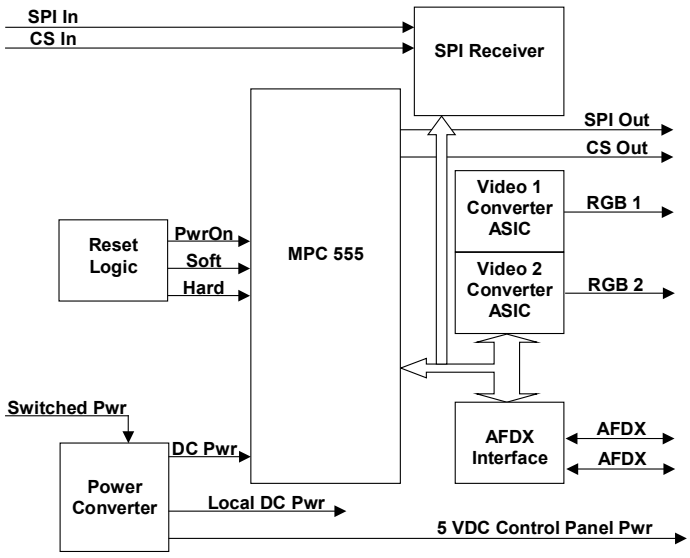

Figure 5. RMS Computer Block Diagram

3.3. Design & simulate the controller hardware, write the control software

Having verified our ES-level RMS simulation with a suite of functional tests, we proceeded with creating a hardware design for the embedded control computer. We chose the PowerPC-based MPC 555 microcontroller as the CPU since we had previously developed (and V&V'd) the instruction-level simulation of this part for another project.

To the MPC 555 we added the following salient elements:

a. An AFDX interface part,
b. An SPI bus receiver part, and
c. Two video ASIC parts that convert compressed digital camera video into RGB signals to drive the control panel video displays.

The MPC 555 incorporates RAM and ROM integral to the part for implementing the controller requirements

developed for our project. Combined with a bit of "glue" logic, these sub-elements are interconnected to create a controller part, bounded exactly like the controller ES, and capable of executing PowerPC object software. Figure 5 gives a block diagram of the RMS Computer hardware design.

Software was written in C language to implement the functional behavior specified in the embedded controller ES. With the source code compiled and loaded into the hardware simulation, this part, referred to as a DE, is the functional equivalent to the ES part from which it was developed. All that remains is to substitute the DE for the ES controller part in the system simulation and run the system-level verification tests.

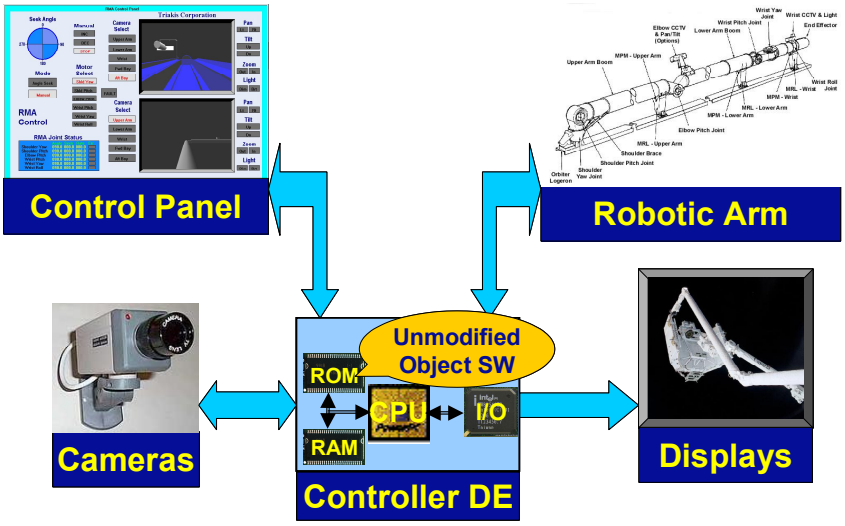

Figure 6. Testing software in system-level simulation

3.4. Testing software in the system simulator

Figure 6 depicts the shuttle RMS system-level diagram with the DE substituted for the ES. Having replaced the ES controller with the DE part in the system simulation, we then reran the same suite of verification tests developed for the ES-based simulation.

Not surprisingly, the software failed to pass all of the system-level tests on the initial run. Table 3 shows the test results from the same test sequence that produced those shown in Table 2, but of the initial test run on the software in contrast to the ES.

Table 3. Initial SW test results segment

```
Testing Shoulder Yaw Control


Positive Velocity

  Actual value: 7.149900
  Display value: 1.000000
  Expected value: 7.200000
  Error range: +- 2.000000
Verify Actual Value
**** PASSED
Verify Display Value
**** FAILED

Positive Angle Seek

  Actual angle: 95.600196
  Display angle: 95.000000
  Expected angle: 97.083018
  Error range: +- 2.000000
Verify Actual Angle
**** PASSED
Verify Display Angle
**** FAILED
```

The results in Table 3 indicate that software control of the RMA produces the correct joint angle, but fails to produce the correct display data. Investigation of the problem revealed a timing discrepancy between the software implementation and the specified implementation of the joint angle data being sent to the control panel display. This problem was resolved by modifying the software to update the display more rapidly and verified by rerunning the system tests and confirming that a "pass" result was logged. The debugging process continued until the software passed all of the system-level tests created to verify the ES controller functionality.

4. Results

Our investigation began with one primary (A), and three secondary (B, C, & D) objectives. The results of our effort are presented in the following paragraphs, ordered by objective. All documentation and data produced during this project effort may be found on the NASA IV&V Facility's SARP Results web site [10]. Tests and test results are documented in the System Test Design Document (SARP-I583-205) and in the System Test Document (SARP-I583-302). The ES- & DE-based simulators and much of the source software are also available at this web site.

4.1. Objective A

Evaluate the viability of maintaining test consistency following substitution of a detailed hardware simulation running actual embedded software (DE) for its ES counterpart in an IcoSim VSIL environment.

Being our primary objective, we devoted a major portion of our activities to achieving test consistency. After building the simulator and defining the system behavior in the RMS Computer executable specification part, we were able to resolve a large portion of the system issues prior to developing tests for the requirements. Using the interactive capability of our virtual RMS Control Panel, we were able to track down problems with individual system parts, problems with inter-component communications messages and timing, and problems with the specified RMA control algorithms.

When satisfied with the behavior of our system, we wrote a suite of 64 requirements tests to verify it. This, of course, uncovered more deficiencies. However, by taking advantage of one really useful quality of developing a system in a pure virtual environment, we generally had a great deal of flexibility about where to attack the problem. We could chose to change an aspect of a subsystem part, the behavior specified in the ES (i.e. the requirement), or the test itself depending on what made the most sense from a system designer viewpoint.

When the ES-based system passed all 64 tests, we turned our attention to developing the PowerPC-based RMS Computer part and the control SW to implement the ES requirements. After reaching the point where the DE would run in the virtual system in place of the ES, we debugged the software using the 64 previously developed requirements tests. The net result is that the software passed all 64 unmodified tests – clearly meeting objective A. Following this initial success, we extended the total number of tests to 131 to investigate other objectives, and resolved some further embedded SW faults until it passed the same additional tests as the ES.

By achieving this objective, we have demonstrated a viable means of unequivocally verifying that software has correctly implemented all of the known, documented, and tested system-level requirements from which it was developed. Further, this has been accomplished with a relatively modest effort and without the use of system integration hardware equipment.

4.2. Objective B

Explore the potential that this process has for reducing V&V time, uncovering functional system & SW implementation deficiencies, and directing the creation of additional tests and/or design changes.

We see software V&V time being reduced through direct reuse of the tests created to verify the system executable specification. Traditional methods employed for system and software development require the creation of separate tests at the system and software levels that, in essence, are targeted at verifying the same behavior. Duplicate tests not only waste time to develop and debug, but also create opportunities for the introduction of interpretation and other types of errors. In the VSIL environment we use, the only additional software verification tests that need be created are for hardware-specific (e.g. built-in-test, etc.) software functions.

V&V time is also reduced through the 100% availability of the system test environment on the same computer that the SW development takes place. As we wrote selected parts of the PowerPC code, we were able to test them immediately without the need to write stubs and without the need of supporting test equipment. Conventional development methods often require the use of external test equipment and/or the scheduling of integration test facility time for software verification.

During the course of our project we encountered and remedied numerous faults distributed among the categories of system design (system specification-level), simulator design, and software design. For example, when testing the angle-seek control mode in the ES we realized that the algorithm drove the selected joint only in the clock-wise direction rather than moving the direction required to take the shortest distance. This case was clearly a specification issue solved by redesigning the algorithm programmed in the ES.

Developing the system at the concept level with no constraints necessitating the use of pre-developed parts, afforded us the greatest flexibility in resolving problems as we encountered them. In one case we wanted to enhance the information displayed on the control panel during angle-seek control mode by lighting the “INC,” “DEC,” or “STOP” manual buttons according to the motion state of the RMA joint. Implementing this in the control computer ES required an upgrade to the control panel message processing capability to accommodate the new message formats. This upgrade paved the way for an easy message extension required to add fault reporting to the control panel display.

Adding fault reporting to the control panel part meant that subsystems would be required to report their status to the RMS computer. When status reporting was added to various subsystem parts it became apparent that it would be most effective to send a status word with every message from the part – so all messages from the parts to the computer ES had to be upgraded. To support testing this feature, we added the ability to change the part status under test control in order to verify that a part fault can be detected and reported to the control panel operator in a specified time. Of course all of these changes led to changing the control software written to execute in the RMS computer DE. One unexpected consequence of adding the extra control panel messages to the target PowerPC software was that the MPC-555's integrated SPI communication channel did not handle multiple messages as conveniently as we had expected. Coding the software to send one message to the control panel per event cycle (rather than from 1 to 3 as the ES does) solved the problem thereby allowing the SW to pass the ES tests.

It is worth noting that testing the fault reporting capability uncovered a problem with the simulator itself. The robotic arm has two data acquisition modules that send strain gauge data to the computer. The ES and the DE both failed the fault-reporting test for the upper arm strain gauge data module. While we have left this problem intact to illustrate a point, our investigation revealed that for some reason the upper arm data module is not responding to any commands or queries from the computer. The part itself functions correctly since it is also used for the lower arm strain gauge and passes the same tests.

In meeting objective B then, we have shown how testing in this VSIL environment potentially reduces V&V time, uncovers system & SW implementation deficiencies, reveals simulator problems and directs the creation of additional tests & design changes.

4.3. Objective C

Evaluate the extent to which the Triakis concept of executable specifications (ES') achieves unambiguous communication of system requirements thereby reducing interpretation-induced errors.

We purposefully chose the C computer language for ES development because of the following qualities:

a. It is inherently unambiguous,
b. It is general purpose,
c. It is widely accepted, and
d. It integrates well with our simulator (written in C++).

All ES languages share the common goals of eliminating ambiguity and enabling direct testability of system requirements. The general-purpose nature of C imposes few constraints on the system designer – creating what some might consider to be enough rope with which to "hang one's self." Were the ES to stand alone, we might be inclined to agree. However, in the context of operating within a VSIL, the entire system imposes constraints on the designer. Within those constraints, the designer is afforded a great deal of flexibility in how to develop the behavior of the part being specified.

During the process of developing the MPC555 source code from the RMS Computer ES, we discovered no ambiguities. In other words there was never any question about the intent of the system designer with respect to the behavior specified in the ES. This is not surprising because the C language is inherently unambiguous, and not terribly revealing since the same people developed both the specification and the control software.

What we did find, though, was that shortcuts taken in the creation of key system parts (due to resource constraints) caused ambiguities in signal & message timing between system elements. One of the powerful features about the IcoSim ES is that it allows the designer to create very complex behavior that executes in virtually no time at all. Tongue-in-cheek aside, a complex algorithm that might take a great deal of real-world processing time can be computed virtually instantaneously in an ES. In the development of our simulator, none of the parts have been made with timing delays representative of real-world data transmission bandwidths.

This was not a problem in the ES-only simulator but when we began to test the MPC555 control software in the DE-based simulator, we discovered some timing-related problems in the communications interface with the RMS Control Panel. The integrated SPI bus within the simulated MPC555 was implemented with a higher fidelity than the ES SPI bus. Consequently we had to make some changes to the control software to compensate for the differing characteristics embodied in the higher fidelity simulation of the real-world part. Were this an actual development project, improving the SPI bus fidelity within the control panel and RMS computer ES' would have been preferable to "coding around" the problem in the software.

In summary, while our concept of ES' certainly appeared to communicate the behavioral intent of the system design in an unambiguous manner, it may be desirable to adopt some rules compelling the designer to realistically model signal and data bus timing.

4.4. Objective D

Evaluate methods of gathering metric data on dynamic aspects of a software program only possible (or at least a lot easier) in a virtual environment.

In addition to running the same system requirements tests developed for the ES-based simulator, the test script for the control software was extended to gather the below SW execution metrics of interest. None of the metrics gathered required instrumentation of the code itself. The metric data itself can be found in the System Test Document that is part of the document repository for this project available on the NASA IV&V Facility SARP Results website at [10].

a. **Software Path Coverage:** The software path coverage report shows all conditional jump (or call) assembly code instructions and identifies whether a condition caused a jump or no jump (execution fell through to the following instruction) during execution. Typically, test development would not be considered complete until all code paths are exercised.
b. **Software Code Coverage:** The software code coverage report calculates the percentage of code executed during the test run. The 131 tests we ran covered only 27% of the code written. On an actual development project, tests would continue to be written until 100% coverage was achieved.
c. **Software Interrupt:** We made this report by automatically recording the time at entry and exit of the periodic processing routine. The measured time spent in the in this routing was used to compute the percent of reserve processing time available. For this project that number turned out to be about 33.4%.
d. **Software Execution Marker:** The software execution marker report provides a look at where the processing time is spent. This is done by counting the number of times that individual instructions are executed over a set period of time. This, in effect, forms a histogram of where processing time is spent during program execution.
e. **Software Subroutine & Interrupt Call:** This report logs all subroutine and interrupts made during program execution. These data are useful for analyzing and debugging program control flow.

This is just a representative example of the types of metrics that can be gathered in the simulator. There are a great deal more possible metrics that can be gathered on the dynamically executing software. We are interested in hearing about useful types of metrics that other researchers may see value in collecting.

4.5. Discussion

Our concept of executable specifications merits some further discussion. First, it might be useful to explain that we have not created a new system specification language, modeling language, system on a chip (SoC) design language, etc. Nor have we created or used a new formal test method on this project. We have, however, created a new way to render a system-level design that is:

a. Fully testable,
b. Unambiguously specified,
c. Subsystem implementation independent,
d. Highly representative (behaviorally & interface-wise) of its real-world counterpart, and
e. Capable of part substitutions that support running and testing of unmodified executable target SW.

The ES-based VSIL is a very effective tool for rapid prototyping of system designs with complex or simple behavioral models, and single or multiple part types (e.g. electrical, mechanical, hydraulic, pneumatic, etc.). Although ES part behavior is written in C, it is not so burdensome a task as might first be imagined. The creator of an ES part need not worry about embedded operating systems, execution speed, I/O or other resource availability, processor initialization, hardware testing, etc. At the ES-level, there are no processor-imposed or other HW implementation-specific constraints on the execution speed requirements of algorithm and other computations. Further, one may have as many simultaneous processes and interrupting signals occurring as desired. It is the implementation team's job to decide on a specific hardware & software design to host the specified behavior.

Since our ES concept has not been developed into a formal method, we have developed a set of guidelines to which we generally adhere when creating an ES. Our guidelines include, for example, the following:

- Proper behavior of power events (e.g. Power-on delay, cold/warm start) must be specified
- Use multiple, independent state machines when possible to implement functional behavior. State transitions may be triggered by time delays, signal changes, other state machine transitions, etc.
- Timing values, analog measurements, etc. should be specified in ranges
- Timing values, analog measurements, etc. should be dynamically adjustable under test control
- Et al.

It should be apparent by now that our ES concept is a system design & requirements specification tool and not intended to support the specification of general system requirements prior to design. Our ES idea compliments rather than supplants tools like S^3L (State-System Specification Language), UML (Unified Modeling Language), DESML (Dynamic Emergent System Modeling Language), CDL (Component Description Language), et al. In general, these classes of specification and modeling languages operate at a higher abstraction level than does our ES tool. After developing the system specifications & models required, they could then be transferred into a testable system design as we have described herein. It may be beneficial to develop translation utilities that can facilitate the process of migrating the specifications and models developed with one or more of these formal tools, into C code that may be used within our ES parts. Tools such as Simulink (part of the MathWorks toolkit) already provide C code generators that may be used for this purpose.

When developing the embedded software to implement the system behavior specified in the target controller ES, the programmer must consider the constraints imposed by the HW design (e.g. processor(s) used, supporting interface chips, etc.), the RTOS, built-in-test (BIT) software, etc. While some of the code written for the ES may be directly reused, it will likely be necessary to rewrite much of the software to optimize execution efficiency for integration with the controller HW design, for integration with the chosen RTOS, to manage system resources, to add BIT, etc. With the ES code as an unambiguous guide, however, writing the control software is generally a straightforward process. Upon completion of this effort, verification with the system-level tests will ensure that the software has correctly implemented the specified behavior.

5. Implications

The results presented in report carry the potential of having a significant, positive impact on both software assurance and the cost of developing embedded software. However, since the primary tangible result of our research is the discovery of requirements-related software faults prior to the integration test phase of a project, we will restrict the scope of our discussion in this report to that of the economic benefits.

5.1. Economics of fault-finding

Estimates of the cost to find and correct software faults at each of the principal stages of a project have been publicized and widely referenced since 1976 when Boehm first published his study [11] on the subject. Cost numbers vary widely depending on the type of application for which the software is being developed but the common thread they all exhibit is the substantial increase in project costs of carrying problems from one development stage to the next.

A report released in May 2002 by the National Institute of Standards & Technology (NIST) [12] contains a fairly thorough (and sobering) analysis concluding that inadequate software testing costs the United States an estimated *$59.5 billion annually*. The 309-page NIST report is a well-considered treatise on the economic impact of inadequate software testing. While the numbers are extrapolated from software developed for the financial services and transportation applications (CAD, CAM, etc.) sectors, the message applies even more significantly to industries engaged in developing software for safety critical applications such as aerospace, medical, defense, automotive, etc.

NASA recently sponsored a study to evaluate the economic benefit of conducting Independent Validation & Verification (IV&V) during the development of safety-critical embedded systems. IV&V is a proven means of increasing software assurance and, as this study concluded, reducing project costs by uncovering defects early in the development cycle. In the four embedded aerospace projects studied, incorporating IV&V during the development yielded a return on investment (ROI) from 124% to 493% [13]. Table 4 shows the relative cost to repair factors – considered to be conservative estimates for embedded systems – the first four columns of which were used in support of this study.

The relative cost to repair table tells us that when a requirements error passes undetected or unresolved into the design phase, it will cost five times more to correct than were it to have been corrected in the phase in which it was introduced. Correspondingly, it will cost ten times more to repair in the code phase, 50 times more in the test phase, 130 times more in the integration phase, and 368 times more when repaired during the operational phase. Table 4 also gives the cost multipliers for problems introduced in the design, code, test, and integration phases of the development cycle [14].

Table 4. Relative costs to repair embedded software defects by life-cycle phase

Phase defect intro'd	Phase defect repaired					
	Req's	Design	Code	Test	Integ'n	Ops
Req's	1	5	10	50	130	368
Design		1	2	10	26	64
Code			1	5	13	37
Test				1	3	7
Integ'n					1	3

5.2. Discussion

We have demonstrated the feasibility of maintaining test consistency between the system-level, and the software-level verification efforts. In so doing, the system-level and software-level development processes can now be connected at the verification test level. Requirements-induced errors like 74% of those discovered during the system integration phase of the Voyager & Galileo spacecraft software project, can be discovered and repaired at one or possibly more orders of magnitude lower cost.

Below is a summary list of some of the ways that the methods presented in this report may result in an economic benefit to embedded software development efforts:

a. Discovery of system errors early in the development cycle where it is least costly to correct them.
b. Reduce interpretation induced SW faults due to ambiguities in system requirements.
c. Improve ability for dynamic, noninvasive test of system & SW response to failure conditions.
d. Reduce SW faults caused by breakdown in communication of system Requirements changes.
e. New capacity for empirical SW V&V in cases where analysis was only viable means, for example: realistic fault injection & failure mode testing, complex digital signal processor designs, et al.
f. Provide a highly viable means of assuring automatically generated code and reused software.
g. Complete VSIL also provides useful NASA tool for:
   - Post deployment command testing
   - Post deployment SW change testing
   - Anomaly/mishap analysis & problem solving

A virtual environment simulator running unmodified target software is ideally suited for gathering dynamic software metrics without instrumentation of either the target operating system or the software. Code path coverage, MCDC reports, throughput analysis, timing analysis, and many other helpful reports are readily produced with this simulator.

Creating a system design with the type of ES discussed herein results in a verifiable system architecture that is readily translated into component-, and interface-level designs. When contracting out the development of subsystem software, the system-level verification tests can provide an excellent way to assure that the contractor has developed the software correctly.

Because ES parts may be created with intrinsic failure modes that can be invoked dynamically under test control, the system designer can empirically verify the specified system response to a variety of off-nominal conditions. This ability allows greater latitude in the type and number of tests that can be conducted when compared with what is economically viable in a hardware integration lab.

Given all of the potential benefits discussed, it will be important to gain a reliable measure of the cost of developing the types of simulators discussed in this report. Because of the part-oriented nature of the simulators we use, the cost of creating a simulator for a given project will vary in proportion to the number and complexity of new parts that must be created. Since many new spacecraft designs reuse proven subsystem designs from other spacecraft projects, the cost of developing simulators should diminish somewhat with successive similar applications.

A 1998 paper by Kirk Reinholtz with the NASA JPL discusses JPL's decades-long history with simulators [15]. In it he addresses the benefits JPL has come to realize from using simulators in the development of space flight software. Although the approach to simulation taken at JPL differs in many respects to that used in this report, there is a great deal of common ground in Reinholtz's discussion of issues & benefits.

6. Future Work

In the coming year we will be engaged in a joint research effort with the NASA Johnson Space Center funded by the Office of Safety and Mission Assurance under the Software Assurance Research Program. This effort will focus on evaluating how a high-fidelity virtual environment simulation can be used for thorough empirical assessment of system and software behavior in response to a wide range of enhanced system and component failure conditions.

A second study that we expect to conduct next year will focus on testing dynamically executing spacecraft flight software in a pure virtual environment. We will explore if this is a viable domain for conducting independent software verification & validation.

Follow-on studies are expected to look at the economic and assurance benefits of the method described herein via a partial parallel development effort. This would enable a fairly direct, comparative evaluation of the described "closed loop" software development process with conventional methods. Some (or all) of the expenditure required for such an investigation may be returned through project savings resulting from early fault detection & resolution.

7. References

[1] Lutz, R.R., "Analyzing Software Errors in Safety-Critical, Embedded Systems", Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA. 1994

[2] Thompson, J.M., "A Framework for Static Analysis and Simulation of System-level Inter-component Communication", Masters Thesis, University of Minnesota, 1999

[3] Ellis, A., "Achieving Safety in Complex Control Systems" *Proceedings of the Safety-Critical Systems Symposium,* Springr-Verlag, Brighton, England, 1995, pp. 2-14.

[4] Leveson, N.G., *Safeware - System, Safety and Computers*, Addison Wesley, 1995.

[5] Leveson, N.G., "Software safety: What, why and How" *ACM Computing Surveys*, 1986, 18(2)

[6] Lutz, R.R., "Software Engineering for Safety: A Roadmap", Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA. 1999

[7] NASA Shuttle Reference Manual web site, http://spaceflight.nasa.gov/shuttle/reference/index.html

[8] NSTS 1988 News Reference Manual web site, http://science.ksc.nasa.gov/shuttle/technology/sts-newsref/sts-caws.html#sts-deploy

[9] Joels, Kennedy & Larkin; *The Space Shuttle Operator's Manual (Revised Edition)*, Ballantine books, 1988

[10] NASA IV&V Facility; Software Assurance Research Program Results web site, *The Use of a Virtual System Simulator & Executable Specifications*, http://sarpresults.ivv.nasa.gov/ViewResearch/282/32.jsp

[11] Boehm, B.W., "Software Engineering" *IEEE Transactions on Computer,* 1976, SE-1(4):1226-1241.

[12] Tassey, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards & Technology, 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf

[13] Dabney, J.B., "Computing Direct Return on Investment for Software Independent Verification and Validation", *Software Assurance Research Program Results Web Site,* NASA IV&V Facility, Fairmont, West Virginia, 2003, http://sarpresults.ivv.nasa.gov/DownloadFile/24/13/Phase%201%20Report.doc

[14] Dabney, J.B., "Return on Investment of Independent Verification and Validation Study Preliminary Phase IIB Report", *Software Assurance Research Program Results Web Site,* NASA IV&V Facility, Fairmont, West Virginia, 2003, http://sarpresults.ivv.nasa.gov/ViewResearch/289/24.jsp

[15] Reinholtz, K., "Applying Simulation to the Development of Spacecraft Flight Software", Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA. 1998