Testing for Imperfect Integration of Legacy Software Components

David Flater
National Institute of Standards and Technology
100 Bureau Drive, Stop 8260
Gaithersburg, MD 20899-8260 U.S.A.
+1 301 975 3350
dflater@nist.gov

Abstract

In the manufacturing domain, few new distributed systems are built ground-up; most contain wrapped legacy components. While the legacy components themselves are already well-tested, imperfect integration can introduce subtle faults that are outside the prime target area of generic integration and system tests. One might postulate that focused testing for integration faults could improve the yield of detected faults when used as part of a balanced integration and system test effort. We define such a testing strategy and describe a trial application to a prototype control system. The results suggest that focused testing does *not* add significant value over traditional black-box testing.

Keywords

Component, integration, legacy, system, testing

1 Introduction

Integration technologies such as the Common Object Request Broker Architecture (CORBA) [16] and the Component Object Model[*] (COM) [13] have changed the way that software systems for manufacturing and other domains are built. Components that were originally deployed in different places and times are now being wrapped with object-oriented interfaces and made to interact with one another. Data exchange is being replaced by data sharing [12]. This has created a new category of problems for software testers, who must find not only component faults, but also integration faults such as unintended interactions between components and misunderstood interface semantics. While overt functional faults are detected by current testing practices, imperfect integration between legacy components or between legacy components and the interfaces with which they are wrapped can introduce subtle faults that are outside the prime target area of generic functional tests. One might presume that a change in tactics is needed to target the new class of faults – but is it really?

Since working around imperfect integration is a chronic expense, any reduction in imperfect integration could result in dramatic savings. In this paper we will examine the taxonomy of integration conflicts, describe how to construct tests aimed specifically at detecting faults induced by those conflicts, and discuss an application of those tests to a prototype control system.

[*] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

2 Related Work

Black-Box Testing by Boris Beizer provides ample background on traditional testing techniques [2]. Another relevant book by the same author, *Integration and System Testing*, is still forthcoming as of October, 1999.

Some of the worst legacy system integration faults are analogous to faults that can occur in newly developed software that lacks conceptual integrity. Brooks has much to say on this topic in *The Mythical Man-Month* [3], focusing on prevention.

The concepts of components and connectors in a system are formalized in architecture description languages (ADLs). Recent work involving ADLs includes formalization of dependencies between components [19] and operations on connectors [7].

Zhenyi Jin and Jeff Offutt have defined a coupling-based testing technique [11] and coupling-based coverage criteria for integration testing [10]. These focus on white box functional testing of connections, for example, testing the expected values of variables before and after an interaction between components, and on structural coverage.

Numerous formal languages and simulation tools exist for avoiding concurrency-related problems in distributed systems; these are chiefly used in the networking, telecommunication, and real-time disciplines, where finite state approaches are relatively common [8,9].

Finally, there are efforts ongoing, such as those of the Testing Distributed Systems project at Purdue [18] and the Test Special Interest Group of the Object Management Group [15], to build tools and define interfaces for monitoring and debugging the kind of distributed system that concerns us.

3 Taxonomy of Integration Conflicts with Suggested Test Cases

Our approach to defining a "focused" testing strategy for integration conflicts is to (1) explore the taxonomy of integration conflicts; (2) enumerate the techniques used to resolve the conflicts; (3) enumerate the faults that these techniques could potentially introduce; and (4) determine test cases to detect the resulting failures.

We have built this taxonomy out of our own experiences and those of people we surveyed. The categories are not all mutually exclusive, but each one at least provides a unique view of the conflict. Inevitably, readers will have experienced other kinds of conflicts that we have not anticipated, which may suggest additional test cases.

3.1 Connection

3.1.1 Conflict Description

A connection conflict occurs when the interfaces exposed by the components are architecturally different. Examples:

- We have data flowing through a pipeline but we need to integrate a component that only reads and writes to files.
- We have an ODBC (Open Database Connectivity) client but the database server is web-based.
- We need to pipe data into an application that only provides a graphical user interface (GUI) for data entry.

3.1.2 Resolution

In trivial cases, such as the pipes-versus-files conflict, it is possible to resolve the conflict using a scripting language. Output redirection can be used as a pipe-to-file adapter, and any program that can copy a file to standard output can be used as a file-to-pipe adapter. We can think of these programmatic expedients formally as transformations on connectors [7].

In harder cases it may be necessary to wrap a component with a new interface and translation logic; this wrapping is also a kind of integration and can produce new conflicts. In the worst case it is necessary to alter the component itself to give it a new interface because no transformation of the existing connector is feasible.

3.1.3 Potential for Faults

Even the trivial pipe-to-file resolution can introduce faults related to file name collisions, full disk conditions, network problems (for networked file systems), and unintended interactions with other programs or people modifying the file system. Some nontrivial adaptations create the potential for semantic bugs, and most will alter the timing of transactions, possibly triggering timing-related failures.

3.1.4 Test Cases

Test for file name collisions, full disk conditions, network problems, concurrency problems and other timing-sensitive faults by stress-testing (i.e., testing under heavy load) the relevant interface in combination with any other entities, external or internal, that would compete for the same resource. Complex adapters should additionally be subjected to coverage-based testing (i.e., testing to cover some domain of inputs or behaviors) to check each translation that the adapters make.

3.2 Control

3.2.1 Conflict Description

A control conflict occurs when the components embody incompatible assumptions about the flow of control in the system. Examples:

- Two components each expect to be the "main" component, and neither will respond to the other's requests. We call this case "too many leaders."
- Two components each expect to be "pure servers," and neither will make requests of the other. We call this case "no leader."

3.2.2 Resolution

In the "too many leaders" case, one of the "leaders" must be converted into a "follower." This can be more or less difficult depending on the architecture of the component. Often it is necessary to build a wrapper that synthesizes meaningless transactions in order to get the component to perform requested actions.

The "no leader" case is easier. One need only supply the missing "main" component.

3.2.3 Potential for Faults

Converting a "leader" into a "follower" is an inherently fragile operation because it violates the component's assumption that it does, in fact, control the entire system. When a special case arises, the component may take actions, such as initiating interactions with subordinates, that are not anticipated by the integrator. There are also likely to be complications in the production of synthesized transactions. Since it believes that it is the main program, the component will be diligent in rejecting transactions that are inconsistent with its view of the system state, even though that view may be completely synthetic and irrelevant.

3.2.4 Test Cases

For "leaders" that have been converted into "followers," test special cases and long or repeated scenarios that could potentially build up state in the converted component. For newly built "main" components, conduct standard unit, integration, and system tests as you would with any new component.

3.3 Design

3.3.1 Conflict Description

A design conflict occurs when the components were built to satisfy incompatible design requirements. Examples:

- One component is real-time, another is not.
- One component is secure, another is not.
- One component is fault-tolerant, another is not.

If the design requirement in question is also a requirement of the integrated system, then the problem is that one of the components fails to meet that requirement. If the design requirement is not a requirement of the integrated system, then the problem is that one of the components may respond to a timing/security/failure condition that is irrelevant in the integrated system context.

3.3.2 Resolution

Design requirements are often easy to eliminate but usually very hard to retrofit. The design requirements in our examples can be eliminated as follows without modifying the components:

- Real-time deadlines can all be set to the latest possible timestamp.
- Security classifications can all be set to the most public and unprotected policy.
- Failure responses can be turned off or ignored.

But retrofitting them without modifying the components is fragile at best:

- The component can be left sufficiently under-utilized that deadlines will be met.
- The component can be wrapped with a security preprocessor that validates or rejects every transaction.
- The component can be wrapped with a script that restarts it if it crashes.

3.3.3 Potential for Faults

In the first case, the most likely failure is an inappropriate response to a condition that is not relevant to the integrated system. In the second case, the wrapped component is likely to fail to meet the new requirements under stress because there is essentially no control over its actual performance:

- A wrapper cannot force a component to complete its work on-time.
- A wrapper cannot prevent a component from making internal state changes that violate security policy.
- A wrapper cannot force a component to respond intelligently to failure conditions.

If the integrator went to the effort of modifying the internals of the component to retrofit the new requirement, then any kind of software fault may have been introduced.

3.3.4 Test Cases

Stress-test the affected component with respect to the relevant design requirements. If a design requirement has been relaxed, test extreme cases to ensure that an inappropriate response does not occur. If a design requirement has been added, ensure that the requirement continues to be met under stress. If a wrapper has been used, test cases that explore the separation of functionality between the wrapper and the wrapped component. Otherwise, conduct standard unit, integration, and system testing with the modified component.

3.4 Model

3.4.1 Conflict Description

A model conflict occurs when components use incompatible factorings of the same conceptual domain. Examples:

- Vector graphics versus bitmapped graphics.
- Constructive solid geometry CAD (Computer-Aided Design) versus boundary representation CAD.

3.4.2 Resolution

Exchanged information must be subjected to a destructive translation in order to get the receiving system to produce a similar image. Even so, it is only a similar image; the semantics of the original data are lost.

3.4.3 Potential for Faults

Since there is no semantic interchange between the systems, it is unwise to use the received data in any rigorous computations. Preferably, the received data would only be used for presentation, and any faults would only degrade the quality of the presentation without inducing software failures. But in practice, there may not be any alternative to using the data, so related failures will occur.

3.4.4 Test Cases

Test any scenarios in which the translated information is passed farther downstream and/or is used in subsequent computations. Analyze the flow of information in the system to determine all possible impacts of the data and test each significant case.

3.5 Scope

3.5.1 Conflict Description

A conflict of scope occurs when a concept that is important to the world view of one component is not realized by another. Examples:

- (From the PC world) A 3-button mouse needs to operate using a protocol originally designed for a 2-button mouse.
- One component versions documents while another considers versioning to be out-of-scope.
- One component requires an approval for every work item while another does not track approvals at all.

3.5.2 Resolution

If the scope of a component is broader than is required for the integrated system, the unwanted functionality can be ignored or the unwanted fields can be filled with synthesized values. If the scope of a component is narrower than is required for the integrated system, the missing functionality must be provided by a wrapper or somehow embedded within existing structures. (This is what happens with 2½-button serial mice – middle button events are encoded as movements that are zero in both axes.)

3.5.3 Potential for Faults

Again, there is the danger that a component will reject transactions because of synthesized values having no relevance to the system as a whole. Added-on functionality may be faulty, and embedding data within existing structures can create semantic faults and/or conflicts with other components that are trying to use the same fields for a different purpose (maybe even the purpose for which they were put there in the first place).

3.5.4 Test Cases

If synthesized values have been used to adapt a component to a system with a narrower scope, test special cases in which the synthetic values could have an impact. If the scope has been widened by embedding data in existing data structures, trace possible uses of the affected fields, particularly conflicting uses, and test those scenarios. If a wrapper has been used to extend the scope, test cases that explore the separation of functionality between the wrapper and the wrapped component.

3.6 Semantic

3.6.1 Conflict Description

A semantic conflict occurs when the meaning of some entity is different to different components. Examples:

- One component interprets a given message as a command to perform an action while another component interprets it as a notification that the action has been performed.
- Components assume different units for untagged numerical values.
- One component's diagnostic dump is interpreted by another component as flight data. (Although this did occur during the Ariane 5 disaster, it was not the cause of the failure [1].)

3.6.2 Resolution

The problematic entities must be translated into those entities that will best preserve the original semantics when they are interpreted by the receiving system. In cases where no perfect translation exists, the integrator must work around or persuade the users to live with the "lowest common denominator" semantics.

3.6.3 Potential for Faults

If the integrator is not cognizant of all of the semantic disagreements between the components, direct mappings will likely be made between entities that are not quite equivalent. The components will happily interpret each other's data in ways that are valid according to their own specifications but which are completely wrong in the system context. These faults are similar to those that can occur in newly developed software when there is a lack of conceptual integrity. Brooks called these "the most pernicious and subtle" of all bugs [4].

When the components being integrated are legacies from different sources, there is no conceptual integrity except that which

- occurs naturally among like-minded people;
- results from subscription to a common standard, specification, or reference;
- is imposed by the integrator's mapping.

3.6.4 Test Cases

Identify and document scenarios which demonstrate the intended semantics of all system-level concepts that are relevant to the integrated components. Execute the scenarios and verify that the usage of the concepts was consistent with the system-level interpretation.

3.7 Syntactic

3.7.1 Conflict Description

A syntactic conflict occurs when components use different representations for the identical concept. Examples:

- Different file or message formats.
- Different scripting languages.
- Different protocols.

3.7.2 Resolution

Simply translate one syntax to the other.

3.7.3 Potential for Faults

In a word, mistranslation. A mistranslation may result in valid or invalid syntax. If the resulting syntax is invalid, one would expect to see fairly obvious failures. But if the resulting syntax is valid, then the mistranslation generates a semantic fault.

3.7.4 Test Cases

Perform coverage-based testing of the syntactic elements in the source representation. Perform additional tests of cases where the mapping to the target representation is nontrivial, such as when the correct target representation depends on context.

3.8 Unknown

The preceding enumeration includes several techniques that can be used to detect imperfect integration even if the integration conflicts that have been addressed in the system are unknown:

- Test for correct usage of system-level concepts as described in Section 3.6.
- Stress-test the system with respect to overall volume as well as system-level design requirements.
- Test scenarios in which there is high propagation of information through the system.

4 Application to Prototype Control System

4.1 Background

For a test subject, we obtained a prototype control system from another division in our own organization. This system has been used in the past to demonstrate the use of the Real-Time Control Systems (RCS) Library [17] to control a coordinate measuring machine remotely. A real coordinate measuring machine can be used or a simulation can be substituted. For practical reasons, we used only the simulation.

The prototype system is known to produce excellent results when used in practice with a real coordinate measuring machine, even though resources are not available to make a production-quality system. Moreover, the specific integration that we tested was the lowest priority for the prototype system's development. Thus, the results of our testing are only relevant to our hypothesis regarding test methods, and are not significant for evaluating the quality or value of the software.

The following paragraph describes our understanding of the system at the time that we identified test cases. The testing exercise revealed some inaccuracies in our understanding that we will discuss in the analysis.

> The front end receives Dimensional Measuring Interface Standard (DMIS) [6] programs that are normally generated by another component that we have left out of scope. Using the Real-Time Control Systems (RCS) Library, it processes these into Coordinate Measuring Machine (CMM) "canonical commands." The "canonical commands" are then reduced to a simple, *ad hoc* command language that is sent over a socket. The commands are received and interpreted by a Graphic Simulation Language (GSL) program running inside of Deneb/ENVISION [5] on a different host. ENVISION simulates a CMM performing the specified commands and sends simulated inspection data back over the socket. The inspection data are read back by the front end and subsequently analyzed in an operation that we have left out of scope.

4.2 Preliminaries

In order to verify that the installation of the experimental ("sandbox") copy of the demo was successful, we first attempted to run the standard demo scenario without modification. Unfortunately, the installation was not successful the first time. We encountered a series of failures resulting from improper configuration management. With our limited resources, instead of modifying the demo system code to pursue a proper system configuration, these were resolved by recompiling the demo and by identifying specific computer workstations on which the demo was known to work.

Through discussion with the regular operator, we established that the front end understands only a subset of the DMIS language. Clients are expected to restrict their DMIS input to the subset that is understood. This is due to the original purpose of the system: to experiment with certain architectural issues, not to be a complete system. In addition, GSL has broader functionality than what can be encoded in the DMIS subset. There is the potential for conflicts of scope both upstream and downstream, but since this is a known limitation of the system, we must restrict the scope of our testing to the subset of DMIS understood by the front end to obtain results of practical value. We used the DMIS programs from the existing demo as a guide to the testable scope.

We also established *a priori* that a control conflict exists between the front end and ENVISION. In the demo, ENVISION starts up as a standalone program. An operator must interact with it to place it into "server mode" to accept a connection from the front end. The simulation will only idle in this mode for a few minutes before spontaneously reverting to standalone mode, requiring another operator intervention, but this time-out behavior was intentionally introduced to conserve critical system resources.

Finally, we observed that the demo scenario is not calibrated against the CMM simulation as delivered. We elected to ignore this problem.

4.3 Black-Box Testing

Traditional black-box testing treats the implementation under test as opaque; tests are selected based only on coverage of the functional specification or the feasible domain of input. Coverage-based, black-box testing is "targetless" in the sense that it is not designed to detect any *particular* class of error more than any other. On the other hand, because the tester is not looking for any particular class of error, the process is inherently more sensitive to functional errors, which cause hard-to-ignore failures, than to semantic ones, which might require more analysis.

The following abstract test cases were formulated after the preliminaries but before any testing was conducted. The executable test cases were created in sequence after testing was begun, so limitations discovered in earlier test cases would be avoided in later ones. The approach to identifying test cases was to obtain coverage of the domains of parameters to the DMIS commands used in the existing demo scenario. Coverage of DMIS statements *per se* cannot be meaningfully explored since the permissible statements are defined by the existing demo scenario.

Test case 1 Coordinate extrema, GOTO (the DMIS command for unguarded moves)
Test case 2 Coordinate extrema, PTMEAS (the DMIS command for probing)
Test case 3 Vectors for PTMEAS (setting the direction of probing)
Test case 4 Feed rates
Test case 5 Length units
Test case 6 Angle units
Test case 7 Feed rate units

4.4 Testing for Imperfect Integration

Our experimental, "focused" testing strategy is directed by the taxonomy of integration conflicts. Although we narrowed the scope of the system considerably, the following types of conflicts still appear to be relevant:

- Connection (getting ENVISION to receive commands on a socket)
- Control (too many leaders)
- Scope (narrowed from DMIS and GSL to the scope of the DMIS parser and the *ad hoc* control language)
- Semantic (understanding of CMM controller state and behavior, units, usage of vectors, interpretation of *ad hoc* control language)
- Syntactic (DMIS to *ad hoc* language by several steps)

The following abstract test cases were formulated after the preliminaries and after the formulation of the black-box test cases but before any formal testing was conducted. The executable test cases were created in sequence after black-box testing was complete, so limitations discovered in earlier test cases would be avoided in later ones. As with the black-box testing, it was difficult to conceive of test cases that would not unfairly bias the yield by detecting the problems documented in Preliminaries. We considered a test case "fair" if the expected failure modes were not identical to one of the known problems:

- ENVISION will refuse to accept connections from the front end if the front end does not connect to it within a few minutes of being placed in "server mode."
- Interpretation of DMIS language is incomplete.

Using the guidelines in Section 3, we first identified one fair test case for each category of conflict known to be relevant to the system, then filled out the quota with alternatives. Significantly, we rejected the important semantic tests on unit conversions and on the directionality of vectors because they would have been redundant with black-box test cases previously identified.

Test case 1
Category: Connection
Description: Execute a scenario containing many short movements while loading the network link between the two components.
Expected failure modes: stress-related failures.

Test case 2
Category: Control
Description: In the middle of a scenario, set the feed rate (FEDRAT/MESVEL) very slow and send a movement command (PTMEAS) that should take very long to execute.
Expected failure modes: incorrect time-outs, feed rate faster than requested.

Test case 3
Category: Scope
Description: Using SAVE and RECALL, send a long DMIS program that switches coordinate systems several times between movements. (These translations are beyond the scope of the *ad hoc* protocol, as are units conversions.)
Expected failure modes: lost or garbled state.

Test case 4
Category: Semantic
Description: Test the semantics of movement and position with special cases. For example, GOTO the position already occupied.
Expected failure modes: rejection of apparently legal commands, unexpected movements or failures to move.

Test case 5
Category: Syntactic
Description: Exercise syntactic variations on numbers and whitespace in commands. (Most other admissible syntax tests in the very limited scope are made redundant by the black-box testing.)
Expected failure modes: incorrect translation.

Test case 6
Category: Scope

Description: Send a DMIS program containing many boilerplate commands, followed by a normal inspection scenario.
Expected failure modes: stress-related failures, incorrect impacts from boilerplate commands.

Test case 7
Category: Semantic
Description: Test the semantics of probing and collisions with special cases.
Expected failure modes: rejection of apparently legal commands, unexpected collision behavior.

4.5 Results

To conserve space, subtests in which no failures were observed are not detailed here.

4.5.1 Black-Box Tests

Test case 1 Coordinate extrema, GOTO

When specified coordinates exceeded the physical range of the CMM, CMM components "floated" out to the specified coordinates anyway, even passing through the table and the floor when necessary. As the moves became longer, the simulation began "freezing" for increasing periods of time after each move. Upon further investigation, we found that a front-end component logging to a window called EMOVESIMMAIN was active during these "freezes." In an iconified window, it was logging a long series of micro-movements that would, in their summation, equate to the long movement that had been previously commanded. When this operation completed, the next command was issued and ENVISION continued normally.

To enable the test to proceed, all long moves except the final one were commented out. The final move, which used coordinates on the order of 10^{15}, was successfully received by ENVISION. The destination point was drawn, but ENVISION then reverted to console-interactive mode and made no further progress. The front end still read the status as executing. No error message was observed.

When ENVISION was killed to stop the test, the front end reported feedback from ENVISION which was erroneous.

Test case 2 Coordinate extrema, PTMEAS

ENVISION performed the PTMEAS commands at the same velocity as it performed GOTOs. However, the rate used by EMOVESIMMAIN was considerably slower, so the "freezes" were considerably worse than in Test 1 and less subtests could be executed in the available time. Otherwise, the change from GOTO to PTMEAS did not produce different results.

Test case 3 Vectors for PTMEAS

The front-end parser printed an error and entered an infinite loop when WKPLAN (the working plane) was changed inside of a MEAS block (the DMIS program structure in which probing is performed). WKPLAN was moved to the top of the file to fix this. Then the parser had the same problem when it arrived at a PTMEAS without an explicit vector. This subtest was commented out. The test then ran to completion, but some of the steps in the DMIS program were skipped without any warnings or errors. Repeated runs of the test showed different skipped steps.

Test case 4 Feed rates

Test 4 contained subtests for GOTO and PTMEAS related feed rates. The parser printed errors and looped when MESVEL commands (the commands used to set the feed rate) appeared inside a MEAS block. The test was refactored to enable the feed rate to be changed only between MEAS blocks. It was then possible to begin executing the GOTO subtest.

Changes in feed rate affected EMOVESIMMAIN but the rate used by ENVISION was unchanged. When the feed rate became relatively large, EMOVESIMMAIN overshot its target and kept going toward infinity, effectively hanging the test. The GOTO subtest was called complete and commented out.

The PTMEAS subtest ran into problems as only 4 of the 6 probes needed to characterize a cylinder were actually executed (skipped steps again) which caused the fitting algorithm to fail. Making the probes cover a greater distance did not fix the problem. The subtest was called complete except to verify that the overshoot problem also existed for PTMEAS, which it did.

Test case 5 Length units

Changes to the UNITS statement in the DMIS program caused related canonical commands to be issued in the front end but the behavior of ENVISION was unaffected. Neither was a units-changing command issued to ENVISION, nor were lengths converted by the front end before being transmitted.

Test case 6 Angle units

ANGDEC (angles in degree decimal form) functioned normally.

ANGDMS (angles in degrees, minutes, seconds form) caused the parser to issue an error and loop.

ANGRAD (angles in radian form) caused no error messages, but the simulation went berserk – Cartesian coordinates ceased to function normally.

Test case 7 Feed rate units

As before, ENVISION was unaffected by feed rate changes. EMOVESIMMAIN responded correctly to feed rate changes, but overshot its target after the DEFALT rate (sic – spelling is as specified) was selected.

4.5.2 Focused Tests

Test case 1 Connection

Testing prior to network loading showed many skipped steps. However, no change was observed when the network was loaded.

Test case 2 Control

A temporary syntax error in DMIS file #4 resulted in a nonsense parser error being issued for file #5.

As before, ENVISION was unaffected by feed rate changes, and setting the rate to DEFALT resulted in EMOVESIMMAIN overshooting and hanging the test.

Test case 3 Scope

Testing was blocked temporarily while a web server that is accessed by the front end became unavailable.

The parser issued an error and looped when a RECALL command (recall a SAVEd coordinate system) was issued inside of a MEAS block. This was fixed and no other failures were observed.

Test case 4 Semantic

Usage of FROM (define the home position) and GOHOME (return to the home position) syntax in DMIS file #3 resulted in a nonsense parser error being issued for file #4. That syntax was removed and other than skipped moves the test then ran without additional failures.

Test case 5 Syntactic

No significant failures were observed.

Test case 6 Scope

Every ROTATE command (rotate the coordinate system) caused MEASPLMAIN (the log window of a front-end component) to print "invalid transform – better stop," but the behavior of the ENVISION simulation seemed to remain correct. The front end was found to have a limit of 199 features. After the number of features was reduced, the test ran to completion without additional failures.

Test case 7 Semantic

The test ran into trouble due to skipped commands. Additional GOTO commands were added as a workaround to avoid the skipping.

As had been observed previously, the probe passed through the simulated part without stopping. By monitoring the front-end window containing feedback from ENVISION, we verified that no meaningful probe-related feedback was being generated. We then replaced the ENVISION model with an equivalent model that had collision detection enabled. The probe still passed through the simulated part without stopping, but probe-related feedback did appear in the window. Successful probes returned coordinates and three additional parameters while "missed" probes appeared as a repeat of the previous line of feedback. We found that the subtest in which the probe only just contacted the part produced feedback indicative of a missed probe, even though ENVISION changed the color of the screen to show that it had detected a collision. The subtest in which the probe entered the part did produce probe feedback, but it was highly inaccurate.

The subtest in which we attempted to probe the table showed that collisions with the table were not detected at all.

4.6 Analysis

The classification of failures as "legitimately detected" failures versus failures that were related to known demo limitations was subjective. Semantic inconsistencies between the designer and the user led to some difficulty. We assumed a system specification in order to be able to conduct the testing; however, failures occurred which would have been classified as demo limitations had we known to ask about such limitations during the preliminaries. Since we did not, they were validly detected by the tests (in the context of our assumed specification).

To help with this dilemma, we used the following rules of thumb. If a given DMIS command caused the front end to emit a parser error, we excused the failure as a parser limitation; but if there was no parser error and the command was not correctly executed, we counted it towards the "score." Also, if a given misbehavior of the simulation was obviously something that its users had chosen to ignore, we classified it as a limit; but if the behavior was masked in the original demo scenario, we considered it a legitimately detected failure.

The tables below summarize the limitations and failures that were detected by the tests. Limitations are denoted by L*n*, failures by F*n*. Failures that were not directly related to the test scenario, but were observed nonetheless, are flagged by an asterisk.

Table 1: Black-Box Testing

Test 1	
Simulated movement exceeds physical range	L4
Collisions are ignored	L2
ENVISION "freezing"	L5,6
Odd behavior after numerical subtest	F1
Erroneous feedback after ENVISION killed	F2*
Test 2	
ENVISION "freezing"	L5,6
Probing done at same rate as GOTO	L5
Test 3	
WKPLAN can't change on the fly	L3
Must specify vector in PTMEAS	L1
Skipped steps	F3
Test 4	
Can't set MESVEL inside MEAS block	L3
ENVISION ignores feed rate	L5
Front-end sim overshot target, kept going	F4
Skipped steps	F3
Test 5	
Length units ignored	F5
Test 6	
Can't use ANGDMS	L1
ANGRAD causes berserk behavior	F6
Test 7	
ENVISION ignores feed rate	L5
Front-end sim overshot target, kept going	F4

Table 2: Focused Testing

Test 1	
Skipped steps	F3
Test 2	
ENVISION feed rate not changed	L5
Overshoot after set default rate	F4
Misleading parse error message	F7*
Test 3	
Blocked by web server failure	L7
Can't RECALL inside of a MEAS block	L3
Test 4	
Misleading parse error message	F7
Skipped steps	F3
Test 5 no failures	
Test 6	
Limit 199 features	L3
Test 7	
Ignored collisions	L2
Bad output from ENVISION simulated probe	F8

Table 3: Key to Failure Classes

1	Control linkage broken by numerical overflow
2	Loss of connection to ENVISION generates erroneous feedback to front end
3	Combinations of short moves cause some to be skipped
4	Front end simulation moves fail to terminate if feed rate is large
5	Length units are ignored
6	ANGRAD breaks Cartesian coordinates
7	Parser error message identifies wrong input file and wrong command
8	Bad output from ENVISION simulated probe

Table 4: Key to Limit Classes

1	Front end parser is limited
2	No collision detection
3	Reasonable DMIS interpretation or characterization limit
4	No bounds on movement
5	Rate of movement in ENVISION not controlled
6	ENVISION controls are dead while blocking on input from front end
7	Dependent on web server, availability not assured

4.7 Discussion

Many of the identified failures and limitations can be classified as imperfect integration of some sort. However, most of them were detected by black-box testing.

The biggest problem for the testing process was the discovery of the front-end component EMOVESIM and the role that it played. Our investigation of the freezing behavior in the first test showed that the long movements themselves were causing no problem for ENVISION; ENVISION was merely blocking while waiting on input from the front end. The next command would not be issued by the front end until EMOVESIM finished simulating the same movements, which it did very slowly.

We discussed the relevance of this front-end simulation with the operator and learned that in fact, the measurement data received by the front end comes from EMOVESIM. The data returned by ENVISION are not used. This would have posed a severe problem had we defined the scope of our testing to include the subsequent operations on the returned data.

With regards to ignored collisions and simulated movement exceeding the physical range, it was again resource limitations which led to limited functionality. Feedback from Envision to control was not implemented at all; therefore, collisions in Envision do not affect the control. Coordination consistency between the two subsystems was not implemented either, so the probe can fly wild. Collision detection was turned off by default because the concept of collision in Envision was not quite equivalent to the concept of probe touch that was needed in the inspection context; it was being used for an unintended purpose.

5 Conclusion

When used as part of a balanced integration and system test effort, focused testing for integration faults can improve the total yield of detected faults, just as any addition of distinct new test cases can improve the total yield. However, our trial showed a lower yield for the focused tests than for the black-box tests, and more integration-related failures were observed during black-box testing than during focused testing. Significantly, the integration-related failures were readily manifested by tests that did not target them at all. This suggests that traditional black-box testing is still a highly effective approach to use on a system of integrated legacy components, despite differences in the kinds of faults that tend to be present. We did not find evidence to support the theory that black-box testing has a "blind spot" for integration-related faults.

In future work we may conduct similar trials on other systems that we expect to become available, for example, one that integrates a Product Data Manager with a workflow system and/or an Enterprise Resource Planning system, to determine the consistency of the results.

Acknowledgements

This work was conducted as part of the Testability of Interaction-Driven Manufacturing Systems (TIMS) project [14] in the Manufacturing Engineering Laboratory of the National Institute of Standards and Technology. We express our thanks to Hui-Min Huang and other members of the Intelligent Systems Division for providing and supporting the software that we used as a test subject.

References

"By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not, from this page. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites."

[1] ARIANE 5 Flight 501 Failure, Report by the Inquiry Board. <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>, 1996.

[2] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[3] F. P. Brooks. *The Mythical Man-Month*, 20th Anniversary Edition. Addison-Wesley, 1995.

[4] Brooks, p. 142.

[5] Deneb Robotics, Inc. <http://www.deneb.com/>, 1999.

[6] DMIS Information Center. <http://www.dmis.com/>, 1999.

[7] D. Garlan. Foundations for Compositional Connectors. International Workshop on the Role of Software Architecture in Testing and Analysis, 1998. Available at <http://www.ics.uci.edu/~djr/rosatea/papers/garlan.pdf>.

[8] ISO 8807:1989, *Information processing systems — Open System Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*. Available from ISO, <http://www.iso.ch/>.

[9] ITU Z.100 – Z.106, *Specification and Description Language (SDL)*. Available from the International Telecommunication Union, <http://www.itu.int/>.

[10] Z. Jin and J. Offutt. Coupling-based Criteria for Integration Testing. *Journal of Software Testing, Verification, and Reliability*, to appear, 1999. Available at <http://www.ise.gmu.edu/faculty/ofut/rsrch/abstracts/couptest.html>.

[11] Z. Jin and J. Offutt. Coupling-based Integration Testing. In *Proc. ICEECS '96*, pp. 10-17, Montreal, 1996. Available at <http://www.ise.gmu.edu/faculty/ofut/rsrch/abstracts/complex.html>.

[12] S. J. Kemmerer, ed. *STEP: The Grand Experience*. NIST Special Publication #939, U.S. Government Printing Office, Washington, D.C., 1999. Available at <http://www.mel.nist.gov/msidlibrary/summary/9920.html>.

[13] Microsoft. About Microsoft COM. <http://www.microsoft.com/com/about.asp>, 1999.

[14] K. C. Morris, D. Flater, D. Libes, and A. Jones. *Testing of Interaction-driven Manufacturing Systems*. NISTIR 6260, 1998. Available at <http://www.mel.nist.gov/msidlibrary/summary/9827.html>. See also the TIMS web page, <http://www.mel.nist.gov/msid/tims/>.

[15] Object Management Group. Distributed Debugging API for ORBs and Services Draft Request for Proposal. <http://www.omg.org/cgi-bin/doc?test/99-08-04>, 1999.

[16] Object Management Group. What Is CORBA? <http://www.omg.org/corba/whatiscorba.html>, 1999.

[17] Real-Time Control Systems Library – Software and Documentation. <http://www.isd.mel.nist.gov/projects/rcs_lib/>, 1999.

[18] B. Sridharan. An Extensible Framework for Monitoring and Controlling CORBA Based Distributed Systems. First ICSE Workshop "Testing Distributed Component-Based Systems," Los Angeles, 1999.

[19] J. A. Stafford, D. J. Richardson, and A. L. Wolf. Architecture-level Dependence Analysis for Software Systems. International Workshop on the Role of Software Architecture in Testing and Analysis, 1998. Available at <http://www.ics.uci.edu/~djr/rosatea/papers/stafford.pdf>.