INTEGRATION TESTING OF THE AMRF

William G. Rippey
March 1992

U.S. Department of Commerce
Technology Administration
National Institute of Standards and Technology (NIST)

ABSTRACT

This report presents recommendations that aim to improve the effectiveness and efficiency of developmental testing of large manufacturing systems. This type of testing can require up to half of the total development effort of large systems. [14] The recommendations apply to distributed systems in general and are not limited to manufacturing systems.

The recommendations are based on testing principles that were learned and applied during the development of the Automated Manufacturing Research Facility (AMRF). Use of these integration principles and guidelines for testing will help manage complexity involved in the development of manufacturing systems like the AMRF. Developers can produce a higher quality system, in less time, if they prepare for the complexity through modular subsystem design and good practices of development and testing. In terms of human resources, preparation can reduce time and effort needed to get the system working, reduce stress on the developers, and increase satisfaction with the results.

The AMRF is a major U.S. national laboratory for research in automated manufacturing. It consists of automated workstations and the control rooms and computer equipment necessary to operate them. The facility was developed by NIST as a testbed where scientists and engineers from industry, academia, and the federal government work together on projects of mutual interest. Their research concentrates on the interfaces and measurement techniques needed for successful computer integrated manufacturing.

The material will be of interest primarily to people involved in detailed design and implementation of computer integrated manufacturing systems. Others who may benefit are people responsible for performing, directing, and scheduling system testing.

INTRODUCTION

NIST is investigating the issues of integrating many discrete components into complex manufacturing systems. This report deals with the initial development, from 1982 to 1987, of NIST's Automated Manufacturing Research Facility (AMRF).

There are four major sections in this report:

- Brief Description of the AMRF
- Issues of Integration Testing - discussion of what is meant by the term integration testing, and a description of some AMRF testing concepts.
- Recommendations for Integration Testing - these recommendations form the conclusion of the report.
- An Example of AMRF Integration Testing - this illustrates some of the design recommendations.

Brief Description of the AMRF

The AMRF is a research testbed that models automated production of small batches of machined parts. In 1987 manufacturing tasks were performed by six groups of equipment components, called workstations. Three workstations machined parts by using robots to tend numerically controlled (NC) machine tools. A two-robot workstation deburred parts produced by the machining workstations. The last process in part manufacture was a robotically tended inspection workstation that used a coordinate measuring machine (CMM) and surface roughness sensor system to inspect parts. A materials handling workstation transported part blanks, using a wire guided cart, from a central storage to each of the AMRF workstations. The cart also transported finished parts and NC tooling. Data preparation and NC programming is done by a variety of methods. [6]

To reduce complexity the AMRF hierarchical control architecture modularizes the system by using levels of control. This modularity aids system testing. Command-status interfaces tie AMRF control levels together. The six workstations were coordinated by a cell controller which was at the highest level of the AMRF. Groups of equipment components --robots, machine tools, and fixtures--were coordinated by workstation controllers.

AMRF manufacturing data is distributed throughout the factory both physically and logically. The information that must be exchanged between processes is managed by the Integrated Manufacturing Data Administration System (IMDAS) [6] .

ISSUES OF INTEGRATION TESTING

What is Integration Testing?

Integration testing is the step-wise testing of a system that comprises discrete, interconnected subsystems. Interactions between the subsystems are rigorously tested under controlled conditions that reduce the complexity of system debugging. Its goal is to confirm that a system can function according to design requirements. In progressing toward this goal, the testing must detect system errors, and assign corrections or changes to specific subsystems.

The quality and efficiency of developmental testing has a great effect on:

- system reliability
- how soon a new system becomes operational and useful, i.e. how much time is required for testing.
- how soon an existing system can be revised for more advanced system capabilities or for addition of new subsystems
- the amount of human effort required for system development
- the sense of satisfaction the project staff gain from system development.

A new system that is tested in progressive stages increases in complexity at each stage. Effective system testing constrains the increases in complexity of system operation. Failure to manage testing complexity results in the system becoming too complex to test or operate.

Complexity impedes peoples' ability to understand system operation, and has a direct effect on system reliability. These four factors determine the complexity of any system: number of elements, attributes of the elements, interactions between elements, inherent degree of organization. [10]

The first factor means that when there are more elements in a system, there is more complexity. 'Attributes' are properties or possible states of an element. 'Interactions' are relationships between elements in which they exchange outputs and process them. 'Organization' describes the extent to which predetermined rules, which guide interactions or describe attributes, exist. The overall complexity of a system is determined by the relationship among the four factors.

Testing and the System Development Life-Cycle

The development of a system can be divided into four phases: design, implementation, testing and maintenance, shown in Figure 1. Examples of techniques to ensure and improve the quality of the system at each phase are shown below the boxes. The recommendations in this report apply to the design and testing phases.

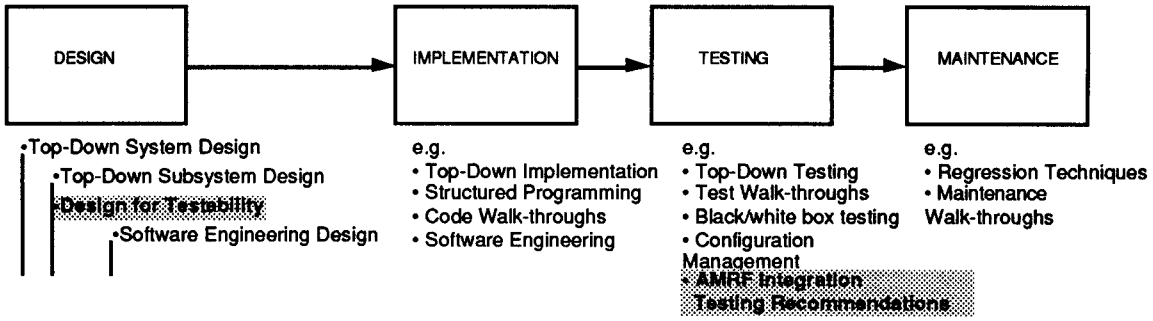

Figure 1. Steps in the life-cycle of software/hardware systems. The bullets denote techniques that can be used to improve quality of the system. The shaded areas are addressed by this report.

Adapted from reference [13].

AMRF Integration Testing

In the AMRF each subsystem is a distributed hardware, software, or hardware-and-software system; e.g. robots, machine tools, communications systems, database systems, intelligent sensors, controllers. The goal of AMRF integration testing is to achieve progressively advanced levels of operation of the automated factory.

Control System Configurations for Integration Testing Configuration is the internal logical structure and operating mode of a controller or control system. During integration testing, AMRF control systems were tested in different configurations for the following reasons:

- to permit step-wise development of controllers. Controllers were tested incrementally, to limit complexity, by integrating untested sub-processes one-by-one into a rigorously tested configuration.
- to simplify integration testing. Complexity of the integrated system can be decreased by reducing complexity in individual controllers.
- to support concurrent development of subsystems. It was often desirable to test interactions between two separate subsystems before either subsystem was complete. Selected modules were run to exercise interface functions: other subsystem functions were not performed for the test. In the AMRF, the most frequent tests run in this fashion were communications tests and IMDAS access tests.
- to aid troubleshooting. If a controller failed during an integration test, its internal configuration was simplified or diagnostics were added, and the test was rerun.
- to conduct a system test when a subsystem failed. The objective was to operate and test an integrated system that contained a failed controller. If a control system component failed, its functionality was emulated by human interaction or by use of a simplified software module.

The number of possible configurations of a controller contributes directly to testing complexity by increasing the number of attributes of the elements in the system. Multiple

configurations provide the advantage of flexibility, if each possible configuration is thoroughly tested and operators are skilled in manipulating the controller.

Table 1 shows the controller configurations used for testing and for operation. A control system operating unattended, fully integrated, is in all five modes on the left. Testing progresses from step-wise modes to integrated modes.

TABLE 1
Control System Configurations for Integration Testing

Integrated Operation Modes	Step-wise Development Modes
Integrated	Stand-alone
AMRF IMDAS	Local database
Lower level connected	No lower level subsystems connected
Minimal or no diagnostics	Diagnostics or trace on
No operator interaction	Operator interaction

Other configuration factors
Environment , change in operator performance.

The first three step-wise modes in Table 1 require the emulation of processes that interact with a controller: specifically, the higher level controller, IMDAS, and lower level controllers. Thus the emulation modules become temporary parts of the system. Figure 2 shows the interfaces involved.

AMRF testing policy requires the stand-alone capability for all AMRF subsystems, in which the higher level controller is emulated. Operator utilities produce command information inputs, and display the status information outputs.

AMRF controllers must be able to switch from accessing AMRF IMDAS information to using their own local database. There are three uses for this configuration change:

- Use of local data is a usual step in individual system development. The first tests are with simplified data and data access protocols.
- By using previously stored data, manufacturing operations can be performed even when communications and/or the IMDAS are not available.
- Complexity of integration testing can be decreased. The subsystem operation is simplified and more reliable in local mode. System-wide, there is reduction of Network traffic and IMDAS computer loading.

In the no lower level subsystems configuration command-status interactions and sometimes the physical tasks of lower level subsystems (e.g. robots, machine tools, fixtures, automatic guided vehicles, automated storage and retrieval systems) are emulated or simulated. Integration testing benefits from increased subsystem reliability and faster subsystem response when physical tasks are not performed. This mode is also used in stepwise subsystem development. The use of emulated lower level subsystems for testing is currently being emphasized in the testbed of the Manufacturing Systems Integration project. [11]

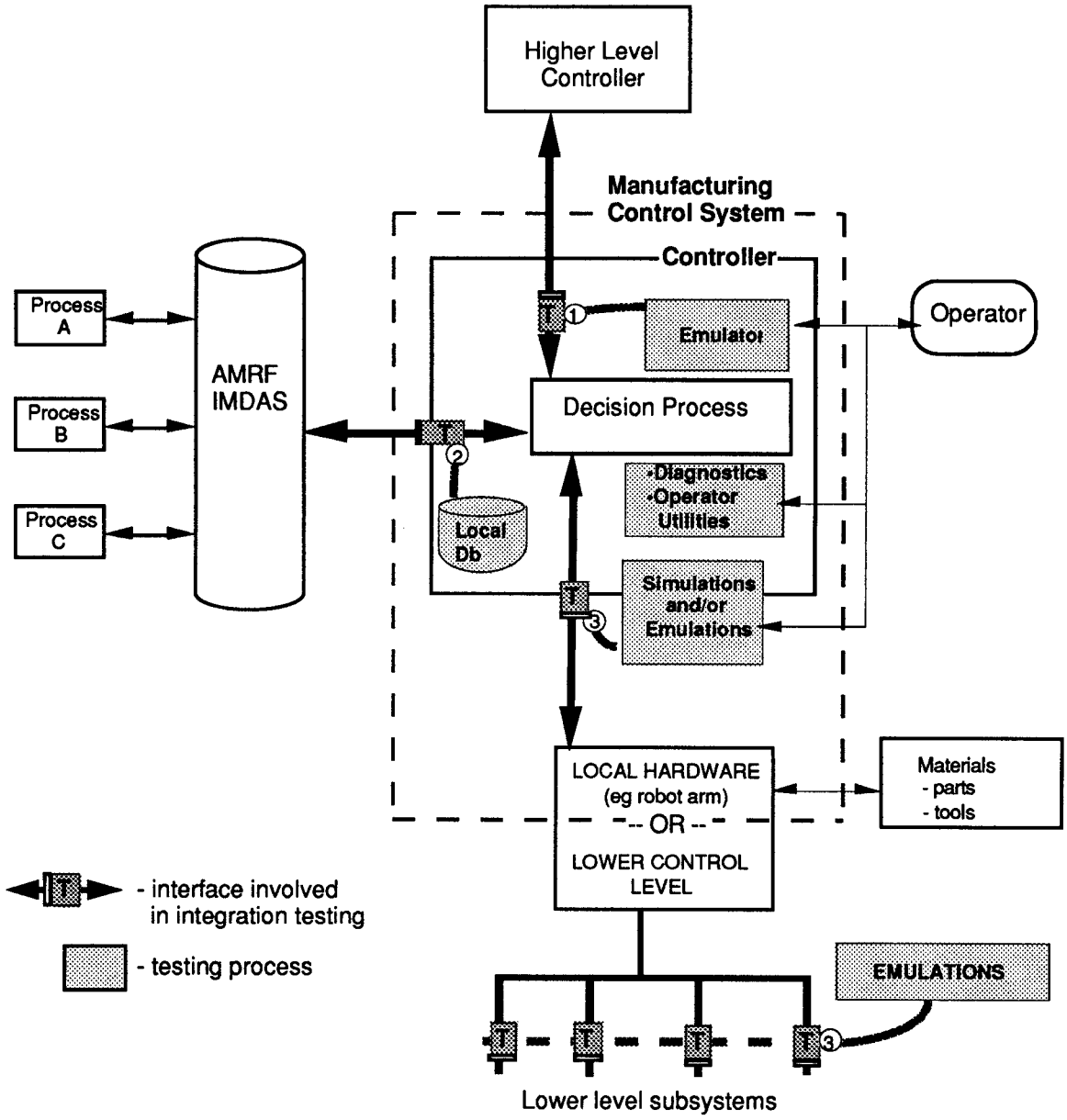

Figure 2. Manufacturing control model showing configurations for integration testing. Testing modes that involve the labeled interfaces are: 1) Stand-alone 2) Local database 3) No lower level subsystems.

Diagnostic testing is the use of extra software routines to capture information during process operation. The use of diagnostics changes the configuration and can affect the operation of a system. Adding or removing diagnostic software statements in source code, and turning procedures on and off, can affect code execution paths, timing, and the behavior of the subsystem at interfaces to other processes. Some of these changes in system performance may not be detected in stand-alone testing.

A change in one process that can produce an apparent change in the behavior of another is an environmental configuration change. A common change in the AMRF was to change to newer versions of operating systems or compilers. Other examples of environmental configuration changes for a controller "C" are: a changed version of controller "H" that issues commands to "C", a changed version of controller "L" that receives commands from "C", and a change to IMDAS.

Phases of AMRF Integration Testing The testing of a large system is done in phases to manage complexity and to permit concurrent development of the subsystems. AMRF testing is done in four phases: interface testing; standalone subsystem testing; step-wise integration testing; and regression testing.

Configuration control is important in proceeding from one phase to the next. Complexity is reduced when tested conditions and data are carried forward to the next phase. The ability to isolate changes in the system as testing progresses is a key to achieving repeatable system tests.

Interface testing exercises the subsystem functions that interact with other subsystems. Typical AMRF functions are communications protocols, and exchange of command and status information. Selected modules of subsystems are run without involving the processing of control programs and control inputs. Interface testing simplifies the system testing by simplifying subsystem operation. It also permits system testing to be started before all subsystems are complete.

Standalone subsystem testing is performed in two phases. First, a controller is run while all interfacing subsystems are emulated. The objective is to thoroughly test internal functions of the controller and the interfaces between control modules and external functions. Decision making and processing of control information by the controller is tested. Next, in the hierarchical systems of the AMRF, lower-level subsystems and their interfaces to the controller are added. The goal is to aggregate complete control systems; i.e. a controller plus all of its lower-level hardware or control processes. The higher level interface to the controller is exercised via operator utilities.

The goal of step-wise integration testing is to connect the control systems that were tested in standalone mode by replacing operator utilities at high level interfaces with the actual controller. If the global data system was not previously tested, it is included at this stage.

System regression testing is required when subsystems are changed, usually to improve them via operating system upgrades, partial redesign, or when they are replaced with a new implementation with the same functionality.

Designing Manufacturing Subsystems for System Testability

Modular subsystem structure is essential for effective system integration testing. There are three steps for incorporating testability needs in subsystem design.

1) Modularize the subsystem to accommodate the four phases of system testing. Specific requirements are stated in recommendations 1-3. Verify that the design is correct by planning a scenario of the four phases of system testing. Some checks on the design are: Can interfacing functions be run independently of control processes? Can information exchanged between functions be examined and/or manipulated? Have interfaces to operator utilities been designed?

2) Incorporate the features described in recommendations 4 and 5.

3) Generate sets of information to be used in system testing. See also reference [4]. Step 3 is a cooperative effort with staff from other subsystem projects. This is not strictly a design task, but it involves the same personnel and should be done immediately after design is complete, and before implementation is begun. This step contributes to the overall understanding of system operation and of the communications and control protocols between subsystems. The early establishment of test data contributes to effective configuration control throughout the four testing phases.

RECOMMENDATIONS FOR INTEGRATION TESTING

The theme of these recommendations is to isolate and constrain system complexity. Two measures of success in efficient testing are: tests that can be run with repeatable results, and the ability of developers to predict test results.

Design Principles and Features

Some measures to control testing complexity affect the design of system components. The key is to anticipate system test methods. Subsystems operating in an integrated environment must have certain configuration change capabilities to support stepwise development and testing.

1) Incorporate into subsystem design the ability to select the testing configurations listed in Table 1. Don't wait until after modules are coded and then patch in changes to accommodate testing.

2) Modularize controller functions that are linked to integration interfaces. Data exchanged between modules should pass through buffers that are accessible to operator utilities. This structure, plus the ability to manipulate buffer data, allows configuration changes to be made more efficiently, and tests of different configurations will be more repeatable. See the example.

3) Put configuration information used for subsystem initialization in data files, not in source code. It must not be necessary to compile a different version of source code to reconfigure a system. The operator procedure for startup should be to select and set all configuration parameters in the file, then load the subsystem software. These initialization files must be covered by configuration management.

4) These operator capabilities for controllers are useful testing features:

- Pause the controller--examine status information--resume, with minimal operator action needed.
- Pause--manually or automatically manipulate outputs--then resume.
- Switch between local and remote database systems dynamically (i.e., without reloading, restarting, or recompiling software).
- Manually initiate module tasks, including external communications
- Assess controller status and activity, especially with a continuous display that does not require keystrokes to invoke.
- Initialize and activate subsystems using automated procedures (rather than step-by-step manual actions).
- Change controller configuration via operator interface while the controller is running. This is the most useful and versatile technique for changing configuration.

5) Controller developers must provide tested software utilities that support the configuration changes listed in Table 1. These utilities emulate or simulate the interactions of other controllers and subsystems.

Integration Testing Procedures

- Before integration testing begins -

6) Test subsystems in stand-alone mode, to verify their start-up and performance.

7) Stand-alone testing must exercise subsystem integration interfaces and protocols. That is, emulated input information should be expressed in the format specified by the integration interface--stand-alone testing using internal representations of interface information is not adequate. See the testing example at the end of the report. The integration buffers are B1 and B2, the internal buffer is B3.

8) Confirm that the performance of emulations of other subsystems used in standalone testing conform to interface specifications. Note: the deliberate generation of

bad interface information is a useful technique for testing subsystem error detection and correction.

9) Specify the design of interfaces with documents that are based on a well-written narrative description of subsystem interactions and information formats. Program listings of procedures and/or data definitions can be used to supplement the narrative, but should not be the sole mechanism for conveying interface information to people.

10) If a tested subsystem is modified (hardware or software) retest it in stand-alone mode before it is included in further integration tests. Also retest when initialization data changes, even if source code did not change.

11) Consider choosing a project-neutral test coordinator to coordinate test activities, centralize configuration control, and provide impartial judgment during troubleshooting.

12) Provide for and enforce central configuration control for subsystem versions of software, hardware, and initialization data. Developers often cannot "see" effects of changes to their subsystems on system operation.

13) Design integration tests before the test is begun. Elements of a good Test Design are:

- Purpose of the test: what will we find out by trying the test; by successfully completing the test?
- Definition of the system to be tested, configuration of the subsystems, and description of the environment. Use drawings or sketches in addition to narrative.
- A functional scenario, including description of initial conditions
- A technical scenario: how functions are done and what interfaces are involved.
- Criteria to use in determining a test's success. These form a prediction of the results of a successful test. Criteria include technical results as well as operational procedures (e.g. the criterion "The subsystem performs well with a non-expert operator.") Quantify criteria when possible, e.g., the robot will process and perform four different commands, the workstation controller will perform three different types of IMDAS transactions.

14) Use a Test Plan to describe tests that are a subset of a given Test Design. In this way, ambitious, longer-range Test Designs can be generated that do not cover details of individual tests. Smaller scope, step-wise tests can be run without generating a new Test Design each time. A Test Plan can reference specific sections of the Design, and note exceptions if applicable.

15) Use a standard form for Test Design and Test Results documents. The formats and titles should be easily recognizable. Make forms easy to read and to complete.

16) Perform upgrades of subsystem computer operating systems or compiler versions after reliable system operation has been achieved. Do not make changes when some progress has been made, but more testing is needed. Notify all subsystem staff before changes are made.

17) Experts on interface protocols and system design should publish examples of transactions or data encodings for use by other project staff. It is especially important to illustrate transactions that affect the most subsystems: in the AMRF these were Network and IMDAS transactions.

18) Generate tools to produce and manage diagnostic trace information. Use "save" files to preserve data in case it is needed later. Develop, document, and use procedures for cleaning up and reclaiming file space after tests.

- Procedures for Integration Testing -

19) All subsystem modules that will run in integration testing must first be successfully run in the stand-alone test.

20) Activation procedures for subsystems and the integrated system must be in written form or directed by interactive software utilities--not remembered by operators. Include activation procedures in configuration control.

21) The following documentation must be on-hand during tests:

- software listings
- hardware wiring diagrams and manuals
- interface specifications (not just well documented software listings)
- a test plan or test design
- standard initialization and startup procedures
- an integration testing policy (recommendations 1 through 30).

22) Dedicate shared resources such as multi-user computers to integration testing, at least in the early stages, so that baseline performance can be assessed.

- Regression testing (retesting of a previously tested system)

23) Retest a subsystem if any of its modules is changed.

24) Retest the system when the configuration of any subsystem has changed, including the apparently harmless removal of diagnostics.

25) Provide utilities for each subsystem for examining interface information for inputs and outputs. Note: continuous display of important system status parameters, without need for keystrokes, is a very desirable capability.

26) Do not train operators of subsystems or the system during integration tests.

27) Do not change operators during critical testing phases. Operator performance may change, and a new operator may not be aware of the latest changes in procedures and environmental conditions.

28) Develop, test and use procedures for orderly system shutdowns. This includes orderly process exits and file space cleanup. Failure to do so can produce conditions that can corrupt subsequent tests.

29) The test coordinator should not operate a subsystem during an integration test.

- After Testing -

30) Issue reports of test results to the participants. This is their feedback; to know if the test was successful, to learn how other subsystems performed, what the overall problems were, what progress was made, and what additional testing may be necessary.

Human Factors

31) To help communication among test personnel, develop scenarios of system operation, system test procedures and desired results of system operation. These scenarios are elements of good Test Designs.

32) Plan milestones and goals in steps. Early and intermediate successes help build confidence and morale. There are some differences between integration testing and testing of small projects regarding personal factors. Integration testing is a team activity in which individuals' satisfaction is linked strongly to cooperative success. Also, results of the cooperative testing are visible to members of the team. In smaller project development intermediate results may only be known to that project's staff.

33) Visual displays

- Make operator interfaces instructive, not terse. Eliminate distracting information on visual displays.
- Don't leave displays blank during process execution. Use positive feedback to indicate a process's state, such as "Process Begun", "Requesting Data", "Loading Part". Also see [12], p. 98, 213, 223, 308, 311, 312, 325.

34) When applicable, identify and document a common high-level methodology to be used in system and subsystem design by all personnel . This provides a common ground that simplifies interpersonal communications and produces more consistent designs of subsystems and interfaces. The example in the AMRF is the concept of hierarchical control and characteristics of all command-status interfaces.

AN EXAMPLE OF AMRF INTEGRATION TESTING

This example shows the flexibility that should be designed into controllers to allow the configuration changes required for step-wise development and for integration testing. The configuration change is between integrated and stand-alone modes. The most important design principle is to modularize controller functions. The controller configuration is then changed by controlling module execution and manipulating data that is exchanged between modules, rather that by compiling different versions of software.

Figure 3 shows the internal modules and information buffers of a controller. In integrated operation a Network mailbox conveys commands from a higher level controller. The Common Memory Protocol (the AMRF communications paradigm) module copies the information into an internal buffer. The conversion module processes the standard AMRF format information into an internal representation of commands and parameters that is used directly by the controller's decision process.

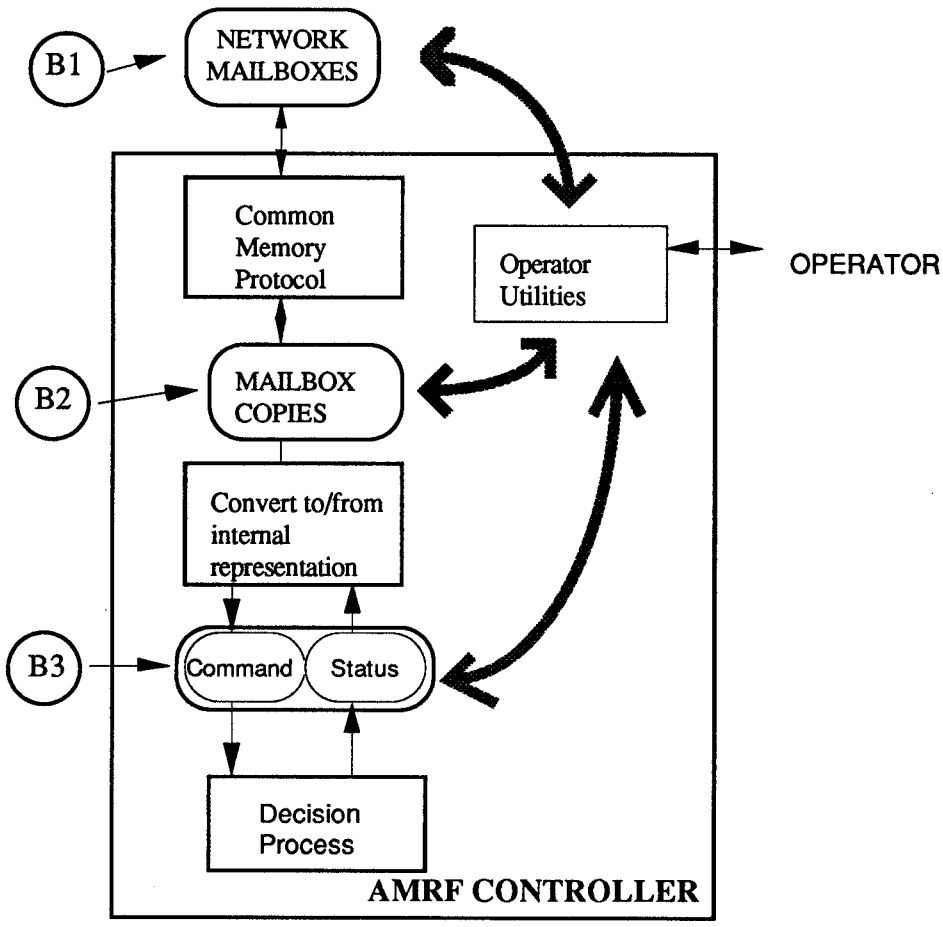

Figure 3. Processing modules and information buffers of a controller.

In stand-alone configuration an operator generates the command used by the controller. The operator utilities can produce information that is inserted into any of three information buffers, either the Network mailbox B1, or the internal copy buffer B2, or the internal representation buffer B3. The utilities also display and provide ability to change all

information in the buffers, including commands and the status information produced by the controller.

There are four testing activities that benefit from the modular controller structure and the use of buffers for information exchange:

1) Changing controller configuration. In integrated configuration a higher-level controller produces command data in buffer B1. In stand-alone mode the operator utilities would produce the command information--the configuration of internal modules is identical to that of integrated operation. If the absence of the Network dictated that buffer B1 was not physically present, then the operator utilities would generate data for buffer B2.

2) Stepwise controller development can be performed by first testing the decision process via B3, then adding the conversion module by using B2, then adding mailbox communications functions by using B1.

3) An effective diagnostic technique is to isolate the actions of modules by manipulating input data in buffers, running the module, and observing results. Modules that exchange data can also be run in "single-step" mode so testers can trace controller internal performance.

4) Error recovery can be performed via operator intervention in some cases by pausing the controller, clearing the local error and replacing the error status report in buffer B3 with a status report of DONE. The controller can be restarted, it will report its DONE status to the higher control level, and be ready for the next command.

REFERENCES

1. Barbera, A.J., Fitzgerald, M.L., Albus, J.S., "Concepts for a real-time sensory interactive control system architecture", Proceeding for the Fourteenth Southeastern Symposium on System Theory, April 1982, pp. 121-126.

2. Barbera, A.J., Fitzgerald, M.L., informal notes, "Demo suggestions", June 1984.

3. Basili, Victor R., Perricone, Barry T., Software Errors and Complexity: an Empirical Investigation, Communications of the ACM, Volume 27, Number 1, January 1984.

4. Branstad, Martha A., Cherniavsky, John C., Adrion, Richards W., "NBS Special Publication 500-56", U.S. Department of Commerce, National Bureau of Standards, February 1980, reprinted in Structured Testing, Thomas J. McCabe, ed., IEEE Computer Society, 1983: see page 84.

5. Furlani, C.M. et al., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Proceedings of Summer Computer Simulation Conference, Vancouver, B.C., July 1983.

6. Libes, Don, Barkmeyer, E., "Integrated Manufacturing Data Administration System - An Overview", Vol. 1, No. 1, International Journal of Computer Integrated Manufacturing, January, 1988.

7. Nanzetta, Philip, Weaver, Asenath, Wellington, Joan, Wood, Linda, Publications of the Center for Manufacturing Engineering Covering the Period January 1978 - December 1988, NISTIR 89-4180.

8. Nanzetta, Philip, Hutchins, Cheryl, Wood, Linda, Publications of the Manufacturing Engineering Laboratory Covering the Period January 1989 - September 1991, NISTIR 4713.

9. Rybczynski, S., et al., AMRF Network Communications, NBSIR 88-3816, June 30, 1988.

10. Schoderbek, Peter P., Schoderbek, Charles G., Kefalas, Asterios G., Management Systems, Conceptual Considerations, Plano Texas, 1985, 383 pp.

11. Senehi, M.K, Barkmeyer, Ed, Luce, Mark, Ray, Steven, Wallace, Evan K., Wallace, Sarah, Manufacturing Systems Integration Initial Architecture Document, NISTIR 4682, September 1991.

12. Smith, Sidney L, Mosier, Jane N., Guidelines for Designing User Interface Software, Bedford, MA., The Mitre Corporation, 1986, 478 pp. NTIS document number AD A177 198.

13. Walsh, Thomas J., "A software reliability study using a complexity measure", p. 111, reprinted in Structured Testing, Thomas J. McCabe, ed., IEEE Computer Society Press, 1983.

14. Wasserman, Anthony I., "Information system design methodology", Journal of the American Society for Information Science, January 1980, reprinted in Tutorial on Software Design Techniques, IEEE Computer Society, 1983, edited by Peter Freeman and Anthony I. Wasserman.