Distributed Testing of an Equipment-Level Interface Specification

John Horst, Thomas Kramer, Keith Stouffer, Joseph Falco,
Hui-Min Huang, Frederick Proctor, and Albert Wavering
National Institute of Standards and Technology (NIST),
Gaithersburg, Maryland, USA.
E-mail: {john.horst, thomas.kramer, keith.stouffer, joseph.falco,
hui-min.huang, frederick.proctor, albert.wavering}@nist.gov

Abstract— **A test suite for a key interface within a dimensional measuring system (coordinate measuring machine or CMM) is presented. The test suite consists of test procedures, test definitions, and various testing utilities. A real-time, distributed test utilizing the test suite has been performed and is described.**

Keywords— **conformance test, coordinate measuring machine, distributed testing, interface specifications, metrology, object-oriented, real-time systems, test suite, validation test**

I. Introduction

Software testing has become critical to software quality. However, interface specification development efforts often treat testing as non-essential. We argue for the early and rigorous application of testing to specification development and implementation, prior to the standardization process. The goal of interface specification development is to create a widely accepted common (or neutral) language at the interface, and the goal of the common language is system interoperability.

We will be looking at the following systems on either side of a two-way interface: application software on one side and a real-time logic and motion control device on the other. The device is a coordinate measuring machine (CMM) consisting of device control software (for logic and motion control), motion control hardware, sensors, and actuators. The application software performs many operations including interface to the CMM operator, interpreting inspection programs in higher level languages, analyzing sensor data, and estimating high level features from low-level sensed data.

NIST has been working with those responsible for a particular interface specification, developing and implementing testing procedures and tools. This document describes what those procedures and tools are, what tasks they perform, and what remaining work needs to be done.

Real-time and object-oriented issues such as remote testing, controller-to-device interface specifications and testing, and the role of object-oriented languages in the specification and test suite will be discussed.

II. CMM Device-Level Specification

A CMM equipment-level interface specification currently under development has received a broad level of industry support [1], called the "I++ DME-Interface" specification. It consists of requirements for communications protocol, software execution paradigm, and the syntax and semantics for command and response across the interface. For example, communications protocol might be TCP/IP or direct serial link. Software execution paradigms might include cyclic execution of commands and responses, preemptive execution, or a combination of both. Syntax may require ASCII text format and the type and ordering of allowable characters. Semantics will specify the precise meaning of commands and responses. This includes definition of the allowable error types and what each error type means.

The audience of the specification is the technical implementors who will be creating interpreters for application software or device software that either convert from the native language to the common or *vice-versa*. It is expected that users will not be involved in many of the details of the specification. However, users may need to interact with the CMM system directly, not just through the application software, and the interface under consideration is between the application software and the CMM. Functional requirements also are essential inputs from the users.

III. Test Suite

We define the total collection of all testing-related entities as the "test suite." The entire test suite consists of the following elements, most of which are under varying stages of development at NIST.

- test types
  - validation tests
  - conformance tests
    - CMM implementation tests
    - application implementation tests
    - cross testing (simultaneous testing of two implementations)
- testing utilities
  - test cases
    - inspection plans
    - test artifacts
  - common test software
    - application simulator
    - response simulator

  - command and response classes
  - analysis tools and metrics
- testing procedure

The test suite ostensibly checks to make sure an implementation is written according to the specification. However, the test suite is ultimately meant to improve the specification. As we conduct the tests, we will invariably find that certain aspects of the specification need adjustment. As we perform and analyze tests, we will also discover that the test suite itself needs to be improved. So, testing is iterative with the implementation development, the specification, and the test suite itself.

Each of these elements in the test suite will now be described.

IV. Types of Tests

Conformance testing is the effort to determine the level of compliance of implementations of the specification to the specification itself. Validation testing is the attempt to guarantee that all the functionality needed on the interface is explicit in the specification.

Validation testing should be done early in the specification development process, so that all necessary functionality is being integrated into the specification. In the same manner, conformance testing should begin early in the development process and be used to test every new implementation of the specification. The validation test will not have such continued use. For a good overview of conformance testing see [2].

A. Validation Tests

Validation testing is necessary to assure the developers that all the functionality required by users of the interface is sufficiently expressible in commands and responses detailed in the specification. The specification under discussion [1] has not, as far as we know, undergone any formal validation tests for functionality. Functionality "coverage" has been accomplished informally (as thought experiments).

B. Conformance Tests

A useful family of conformance tests for the specification might consist of three classes of tests. The first two are conformance tests and the last is an interoperability test.

1) A suite of test files containing specification compliant strings are placed on a socket by the common application utility (see Section V-A.3) and sent to a CMM simulator or a real CMM that has an implementation of the specification. The test will automatically (or manually) compare the real output to the expected output and log the results of this comparison for analysis. The test files will command the CMM to perform various inspection tasks on a common test artifact. Test files include erroneous commands to test for proper response to such inputs. This is done for n CMM systems. A "soft" real-time, distributed, cross-continental test was performed on July 13, 2001 and is described in Section VIII.

2) A suite of test response files is generated for use by the (common) CMM utility software (see Section V-A.4). The response files will include a variety of rules for producing various kinds of responses depending on the input received by the CMM utility. The rules will look at the commands, command sequence, and random events to determine the appropriate response for each test. The response files will contain rules for producing legal and illegal responses for both semantics and syntax to fully exercise the application level implementation. Response files coupled with the CMM utility will be employed with all m application software packages. High level test inspection plans may also be necessary and helpful. The CMM utility will automatically (or manually) compare the real output of the application software (subsequent commands) to the expected output and log the results of this comparison for analysis. The goal in testing application implementations is to minimize (or avoid altogether) any required testing modules in the application software.

3) A suite of test files consisting of high level inspection plans are input to an implementation of the specification by an application software package that performs inspection plan execution and sends it to a CMM simulator or a real CMM that also has an implementation of the specification. The output of the application software will also be sent into the common receiver utility, and comparisons and results will be logged for analysis. The test files (again including errors) will command the CMM to perform various inspection tasks on a common test artifact. This test will be repeated for the $m \times n$ combinations of m inspection plan execution software packages and n CMM systems. An analysis utility will compare the real output (subsequent commands) of the high level inspection program to the expected output and log the results of this comparison for analysis.

The goal of performing these tests is to verify the correctness of implementations, produce a complete and unambiguous specification, and facilitate interoperability.

V. Testing Utilities

Successful design and implementation of the validation and conformance tests requires that there be a mix of testing utilities to support these tests. The general idea is to have the same (common) utilities used by everyone conducting tests. This minimizes variations in the meaning of the test results. If errors in the test utilities occur, all testing participants should experience them and the common testing utilities can be fixed. Testing utilities include the following:

- test cases
  - test artifacts
  - inspection plans
- common test software
  - application simulator
  - response simulator

- analysis tools and metrics

We need to test implementations on both sides of the interface. Interestingly, such tests are not symmetric. Tests for implementations on the CMM side simply require a sufficient set of test cases, each consisting of a test artifact paired with a test inspection plan for that artifact. However, the testing of an implementation on the application side is more subtle and challenging as is described in Section IV-B.

We have not created application software testing utilities as yet, except that we have a CMM response utility that parses specification compliant commands and very roughly simulates CMM-like responses. Simulation involves simply appropriate delays for "move" and "measure" commands. We now describe tools for testing CMM implementations of the specification.

A. Test Cases

A single test case consists of a test artifact paired with a test inspection plan for that artifact. We want these test cases to provide sufficient coverage of the types of commands allowed in the specification. We use these test cases to sufficiently exercise the commands from the specification as realized in each implementation under test.

A.1 Test Artifacts

Common test artifacts are needed to minimize the sources of error in the various implementations. We need to be able to ensure that any problem in execution of a test inspection plan is not attributable to the test artifact. If we restrict all tests to specific and common test artifacts, we stand a better chance of efficiently debugging problems in the various implementations of a specification and ensuring that the implementation conforms to the specification.

A.2 Inspection Plans

Though currently not the case, we expect that the test inspection plans will consist of files of both high level (*e.g.*, the Dimensional Measuring Interface Standard (DMIS) [3]) commands and files consisting of lists of specification compliant strings written by hand. We must have the latter in order to successfully introduce errors into the list of commands. A subset of these files will simply contain one command per file. Log files consisting of the correct responses will exist for each file. The inspection plans should not require any artifact other than the test artifact(s). We want the files to test both syntax errors and semantical errors. The latter would include errors in execution, for example, the probe being sent to measure a non-existent point or the probe encountering an obstacle prior to an approach point in a "measure" command or in a "move" command. Some of the files we wrote, after debugging, were error-free, some were filled with intentional errors, and some had only a smattering of intentional errors.

The error-free files included:

1. Some similar to what a list of specification compliant messages coming from a DMIS interpreter might look like when DMIS code was being interpreted for inspecting a simple feature such as a cylinder or a plane.
2. Some that simply tested part of the specification by including all messages of a certain type (such as all messages for getting and setting parameter values).

The error-filled files were of two types:

1. Sets of command messages with various syntax errors. Files of this sort may contain an error on every line.
2. Sets of command messages that cause execution errors. This type of file necessarily contains about two-thirds correct commands, since most execution errors can only occur when a particular machine state has been reached, and it usually takes two or three correct commands to reach a desired state.

For each command file, we wrote a corresponding trace file. The trace file is similar to what would be expected to be in a log file prepared by the application utility. The file consists of pairs of lines, the first being a line from the corresponding command file, and the second being the responses that would be expected from a system executing the command. A real log file, however, would have only one response per line and would not be so nicely ordered, since queueing is used (resulting in "done" messages arriving much later). Also, in the hand-written file, all probed points are at the exact nominal location, whereas in a real file, it would be surprising if any point were exactly at the nominal location.

We wrote a test program (specification compliant command file) for the test artifact that was used in an international demo with LK Metrology, Ltd based on an earlier specification for the same interface [4] (called the "CMM-driver" specification). The demo is described in Section VIII. D. Smith of LK Metrology, Ltd. added substantially to this program. In connection with the test program we wrote C++ software for translating goal points in a specification compliant command file. This was needed since the test artifact might be located on any part of the table of the CMM doing the inspection.

A.3 Application utility software

An application utility has been fully developed for distributed testing of the CMM-driver specification defined in [4] and it is in the process of conversion to the I++ DME-Interface specification. The application utility is a graphical user interface (GUI)-based, object-oriented program that runs on a personal computer (PC) platform and was developed in Visual C++. The commands are sent over a TCP/IP socket to a specification compliant controller. The controller receives the commands, executes them, and returns the appropriate response back to the application utility via the socket. The application utility creates a time-stamped log file of commands sent and received over the socket and performs a validation test on the returned responses to determine if the CMM controller is compliant with the specification. The application utility and a small set of test command files (inspection plans) were delivered to several CMM vendors for evaluation and testing of their specification compliant controllers.

The user executes the application utility through a GUI. The user first selects which type of file to run and the name of the file to run. The application utility can run either of two types of files, a low-level command file or a DMIS file. When a DMIS file is selected, the file is run though an interpreter that converts the DMIS command to the appropriate specification compliant low-level command(s). Only a subset of DMIS commands are supported at this time. The user then selects the name of the log file where the time-stamped data will be recorded. The user specifies the host name of the controller. The host name can be entered as either a fully qualified host name or IP address. The user then specifies the port number. The default port number is 1294 as specified in [4]. When the user pushes the "Connect to CMM Controller" button, a non-blocking TCP/IP socket is created between the application utility and the CMM controller on the specified port. Once the application is connected to the controller, the user can either enter a command manually, single step through the program file that was selected, or run the entire file. A status window displays the current status of the executing program, including what command was just sent or received and any error conditions that exist.

A.4 CMM utility software

The CMM utility software will eventually allow an application software vendor to test his or her implementation of the specification. The idea is to have a set of files consisting of response rules that given a specific event will output a specific response or sequence of responses. The event that triggers the response may be a specific command, or a specific command sequence, or the tick of a clock, or some random event. The utility will examine the nature of the subsequent commands from the application, looking for expected command sequences.

We may also want to create high-level test inspection plans, *e.g.* in DMIS, for use with the CMM utility test software. However, the problem here is as follows. First, we cannot force the generation of certain low-level commands (such as "get parameter") with the detailing of a high-level command. The application developer, in his or her test measurement routines, may never test certain low-level commands. Certain response rules will never fire because the events needed to fire those rules never occur. This will be a complete test of an implementation only if those particular commands are never needed by the application, which is doubtful. Secondly, the particular low-level commands (and the order of execution of those commands) accompanying a particular high-level command will be to some degree the unique choice of the application software developer. On the other hand, certain high level commands, *e.g.*, to "measure a point" or to "go to a point" will of necessity require a low-level "measure" and/or a "move." This issue needs to be debated and resolved as the metrology industry moves along the specification development and testing process.

The current CMM utility includes a command parsing engine coupled with both a real CMM and a simple CMM simulator. NIST developed CMM-driver specification compliant message parsing software, which was integrated into the CMM utility software.

The CMM utility is expected to interact with the application implementation in terms of receiving and handling specification compliant CMM control commands from any application and generating appropriate responses. The commands are to be coded as ASCII character strings and sent through a communication socket with a mutually agreed port number.

This CMM utility software provides the following command reception functions:

- Read text strings from the pre-designated communication socket.
- Interpret the strings according to the specification and either extract for command information or determine error severity and report the errors.
- Manage the commands, *i.e.*, either place them in a queue or abort them.

Both specifications detail the syntax and semantics for error responses. For example, the CMM simulator produces errors such as "parameter out of range" and "illegal probe type." In order to be able to execute these error responses and to be able to verify the correctness of the command reception, the software provides the following CMM simulation functions:

- Retrieve the commands from the queue. Execute them using a state machine model.
- Simulate the machine behavior, in low fidelity, to provide feedback for command execution.
- Report errors if execution fails.

The CMM utility executes at a uniform rate. Its first function is to read the socket for command strings. A parser processes the command string into command serial number, command name, and command parameters. Each segment must conform to the format stated in the specification. When reading a command name, the parser reads until either a left parenthesis or end-of-command character is recognized. Blank spaces in between the characters are allowed but ignored. The command parameter parser is a generic one that handles all the commands, hence, it accommodates variable length commands. For example, a "home" command contains no parameter and a "move axis" command can contain one, two, or three axis parametric values.

The testing utilities also contain a set of common command classes for each of the commands (the common command and response classes are described in Section V-B). Once the parser verifies the command name, the corresponding command class will be used to store the associated parameters. Methods relating to each command are predefined in each command class. A variable-length linked list stores the valid input command class instances in the order that they are received. Currently, if the CMM utility receives an "abort" command, it will clear the queue.

At the end of the parsing, the CMM utility sends a confirmation signal back to the application. The confirmation conveys a message that the input is either a valid or erroneous command. The parser contains a function to determine the severity of the errors.

We partition the CMM utility software such that users can either take the parser and integrate only that into their own machine controllers or also use the execution and simulation modules that are a part of the CMM utility software.

The execution of each of the commands has been implemented using a finite state machine (FSM) model. Prior to execution of the FSM, a command is retrieved from the command queue. Upon execution of the FSM, a new-command flag is verified and, if true, a new instance of the command class is generated and populated with the information that is specific to the command. It then checks for certain common errors in the command string, such as "CMM not initialized" and "invalid command parameters." After the errors are checked out, the FSM calls the simulator for the command. A new instance of the response class is generated and the attributes updated throughout the execution.

B. *Command and Response Classes*

An important part of the testing utilities is a common set of command and response classes. Only the CMM utility uses this common set of definitions for command and response. Using these (or similar) command and status classes in implementations of a specification will reduce development and debug time and will streamline the testing and analysis process. NIST has developed a set of command and response classes which are currently being used within the CMM utility software along with a specification compliant version of the NIST DMIS interpreter. These were written for the earlier specification [4]. The command and response classes are written in C++. Accompanying these classes are C++ files defining various methods for each of these classes. In order to simplify the class structure, the actual command and response classes are derived classes from command and response base classes, respectively. The primary function of these classes is to provide a common set of data structures for passing data and generating command and response strings. Also defined are classes for handling data types and errors defined within the specification. The data classes contain the necessary logic for formatting the data per the specification when a command or response string is being generated.

Command and response classes for the I++ DME-Interface specification are discussed in Section VII. The object model for the I++ DME-Interface specification is a required part of the specification.

C. *Analysis Tools and Metrics*

In order to better utilize conformance testing, quantitative metrics are essential. This will give all implementors measurable incentive to persevere until their implementation receives high marks in all tests. It will also help as a presentation tool to management in all the organizations involved in testing, to quantify progress with simple graphs. Several spreadsheets have been developed. Some preliminary metrics for measuring CMM implementations tests have been developed.

The data analysis tool currently extracts communication performance information from the data log files that both the application and CMM utilities generate. The analysis tool generates a performance report that consists of two parts, for each individual command and for the command file. For each individual command, the analysis tool indicates whether the command has been handled at every stage and within user specified timing ranges, and whether an error condition has occurred. The command performance for the following critical stages were listed:

1. command sent, whether and when (relative to the execution starting time)
2. command receipt
3. command acknowledgement, signal sending
4. command acknowledgement, signal receiving
5. command complete, signal sending
6. command complete, signal receiving.

The summary part of the performance report totals the numbers of commands that are properly and not properly handled, including commands not received and those taking too long to respond.

VI. Testing Procedures

Validation (functional) testing and conformance testing of any specification are both essential. Neither formal validation testing nor formal conformance testing has yet been done on either specification. The implementation and testing that has been done on the CMM-driver specification, however, has provided informal validation and conformance testing. Informal testing should continue and formal testing should begin. It will be useful if an organization consisting of NIST and vendors from both sides of the interface is formed to collaborate in testing. A procedure or process is needed to guide the conformance testing to achieve testing efficiency and high quality implementations.

A. *Validation Testing Procedures*

Informal validation occurs naturally as part of implementation. For example, when an application vendor writes the software that generates commands provided in the specification, if the vendor discovers that there is no command to perform a required action, the vendor raises an issue against the specification.

Validation testing is also a side-effect of conformance testing. If a vendor attempting to build software that conforms to the specification finds that it is unreasonable or impossible to make the software conform, the vendor raises an issue against the specification.

It will be useful to devise a procedure for collecting issues discovered during implementation and conformance testing and considering whether and how the specification might be changed. The Standard Improvement Request process used by the DMIS National Committee provides a model of how this might be done.

Formal validation testing is conducting by identifying functional scenarios that should be supported by a specification and trying to carry out each scenario using the specification. The total set of functional scenarios should cover the entire range of functionality expected to be supported by the specification. This set should be agreed on by the community interested in the specification.

Scenarios might be given in natural language, such as "inspect hole A and plane B on the part whose design is in CAD file C," or they might be given in inspection programs written in DMIS or some other high-level language. Manual use scenarios should also be included, since the specification needs to support manual use.

The role of NIST in validation testing has not yet been determined, but it is likely to include helping devise validation tests and conducting validation testing.

As part of validation testing, it is essential to conduct interoperability testing. In interoperability testing, each application tries working with several CMM vendors and each CMM vendor tries working with several applications. At the stage where software on side A of the interface has been made to work with only one vendor's software on side B of the interface, it is nearly certain that the side A software will not work with a second vendor's side B software. The failures are most likely to reveal that one side or both are not actually conforming. The failures, however, are also likely to reveal ambiguities or other inadequacies in the specification and inadequacies of conformance tests. These should be reported when discovered.

B. Conformance Testing Procedure

A conformance testing procedure is meant to test the compliance of an implementation to the specification. Metrics and analysis are needed to determine the degree of compliance. Conformance tests may be conducted privately by a vendor or they may be conducted by a testing service set up to do conformance testing and issue certificates (or other assurances) of conformance. At one extreme, a conformance testing service might make detailed results of all conformance tests publicly available. At the other extreme, a conformance testing service might make nothing public except certificates of conformance. Test results and trends may be performed privately or at some independent site. The degree of openness is determined by what users and vendors prefer. A side effect of the conformance testing process is to effect an improved interface specification, which can ultimately lead to quantifiable CMM system interoperability results.

The test files and procedures used by a conformance testing service should be publicly available. Ideally, a vendor will have run all the tests and adjusted the software so that it passes all the tests before submitting it to a conformance testing service. Large users might conduct conformance tests themselves. It is expected that particularly interoperability testing will be performed in a distributed manner. Potential commercial implementations of the specification need to be involved in the testing. Care must be taken to insure that testing results are not made available to unwanted parties.

The role of NIST in conformance testing is to devise testing files and procedures that may be used by testing services, vendors and users.

C. Collaborative Testing

It will be useful to form an organization of representatives from NIST and metrology systems vendors (on both sides of the interface) to do informal validation and conformance testing. We assume here that this organization also is responsible for development and modification of the specification; if that is not the case, procedures for changing the specification will be less direct than described below.

We envision two or three phases for this procedure. Phases one and two are differentiated by an expanding scope of commands. Phase three (not yet fully defined) expands to more devices accompanied by a formalization of the specification into a standard. Metrics that will be the measure of testing success are discussed in V-C.

C.1 Phase One Testing

Phase one scope would include some of the following commands:

- two degree of freedom (DOF) probe head and three DOF move of CMM arm (five DOF total)
- measuring in same five DOF
- abort
- get and set parameters

Phase one will exclude error correction and recovery, scanning, rotary motion, and probe calibration. Among other benefits, this will help to debug the conformance testing procedure itself. Phase one testing will consist of tests on the various CMMs and/or their simulators as well as tests on third party inspection program execution software, independently and together. Various high level languages, *e.g.*, DMIS, inspection plans will be required, files of command strings in the format of the emerging specification, and response rules files. Software tools agreed to for use by all participants must be employed in testing to minimize variability.

C.2 Phase Two Testing

The scope of phase two will include error correction and recovery, scanning, rotary motion, and probe calibration as well as the entire set of commands in the specification.

C.3 Conformance Testing Procedure for Both Phases

1. Design (or select or modify) and agree upon an appropriate artifact or artifacts.
2. Develop and achieve consensus on appropriate high level inspection programs and specification compliant command files for each chosen artifact, and response rules for the CMM utility software.
3. Develop, collect, and publish on the web any necessary files (*i.e.*, DMIS files, specification command files,and response rules) along with file format specification (allowing automated output file validation) that will be the standard baseline for testing. Specification-compliant command files will be used in testing because we will not be able to encode in certain high level languages certain behaviors we can perform with CMM driver commands. Furthermore, high level to low level language interpreters will not, in general, be able to produce certain important types of errors.
4. Develop and publish on the web performance analysis tools consisting of various metrics of performance and reporting facilities. These tools will be used to quantify the success of the test results. All participants will agree to these metrics before testing begins.
5. Using the specification compliant inspection programs and application utility software, vendors and users will implement the specification, conduct tests, and do performance analysis.
6. Using the NIST test suite, third party inspection program software vendors and third party software users will conduct tests and do performance analysis.
7. Using high level inspection plans and test artifacts, application software vendors, application software users, CMM vendors, and CMM users will conduct tests on an integrated system and collect data. These tests may be done remotely in a distributed manner, assuming that high level programs require no real-time interaction with the CMM from command to command.
8. Performance results will be collected, stored, and analyzed using agreed performance metrics. Results of this analysis will be made known only to participants, their managers, and any oversight committee.
9. An issues log will be maintained and all participants will log issues resulting from the testing. Regular conference calls following testing sessions will be centered on the resolution of these issues.
10. Modify the specification as needed.
11. Modify the common utility testing software as needed.
12. If the specification is stable or a previously determined time period has expired, conclude and move to next phase. If the specification is not stable, go back to the first step in this procedure.

VII. Real-Time and Object-Oriented Issues

The equipment-level interface is currently a "soft" real-time interface in the sense that both the CMM-driver and I++ DME-Interface specifications are so written that absolute or relative times are not required either between adjacent commands or command and response. This allowed us without reservation to test a CMM implementation of the specification over the Internet. Of course, motion commands like "measure" and "move" were the key commands that would require true real-time operation, but as long as "measure" and "move" commands have final speeds equal to zero, there is no real-time problem. However, if non-zero final speeds of the probe head are allowed, the system is then dependent on the timing of the subsequent command(s) and responses. However, this problem can be resolved if commands are allowed to be queued in the CMM implementation. The I++ DME-Interface specification allows for such queueing, and it implicitly requires it of the CMM implementation. Therefore, assuming that all application implementations expect command queueing in the CMM implementation, the application will send subsequent commands upon receipt of an "OK" response only *versus* "OK" plus "done" (or "error"). The CMM implementation can then look ahead and wait to execute all adjacent "move" commands together which will allow non-zero final velocities for some of the "move" commands.

In the broader sense of real-time, the specification must be developed, implemented, and tested successfully by a broad majority of vendors and users within a certain time frame that might be defined as the technology change window, *i.e.*, the specification cannot be obsolete by the time it appears in new products.

A. The Device Interface

Within the CMM is the "hard" real-time device interface, between the controller on the one hand and the motors and sensors on the other. This interface has been talked about as a candidate for standardization. NIST has done substantial research on making this device interface open, if not common. We developed and reported on a component-based, object-oriented software controller system that outputs motor voltages and read raw sensor values in "hard" real-time [5].

There are several challenges for the development of a device level specification for a CMM. Proprietary device interfaces would have to change at some cost to

CMM vendors. It may be hard to get broad agreement across the industry on certain parameters on the interface. Because of the danger of probe and equipment damage, with CMMs and probes being so costly (often in the $100,000 range), liability is a problem. There is perhaps an even greater liability problem having to do with the assurance of measurement accuracy and many vendors claim that only with a proprietary interface definition can measurement accuracy be ensured.

However, a standard device interface would allow best-in-class controllers paired with best-in-class CMMs which offer hope to increase performance and lower cost of total systems as has happened in several other well-known technology areas. Additionally, the liability for measurement accuracy and damage problems can be shared with some effort to define appropriate liability boundaries. Furthermore, the device interface is relatively a simple interface involving voltages to motors, position and speed of the CMM arm, and a probe tip "touch" event precisely synchronized with the position of the CMM arm when the event occurred.

To develop a test suite for the device interface, remote testing must be abandoned. There would be a need during tests to define as metrics the performance of the machine motion under different controllers. As before, standard artifacts and inspection plans would be essential for successful testing.

B. Object-Oriented Issues

The issue of objects has been in the forefront of development in both specifications. The I++ DME-Interface specification has chosen to make objects (in the form of Unified Modelling Language (UML) models) as part of the specification, whereas the CMM-driver specification makes no such requirement. However, we argue for a middle ground where the object models are part of the testing and implementation tool set that allow implementors to facilitate implementation, testing, and integration. This will both facilitate implementation development (and subsequently interoperability) and will allow implementors to choose non-object oriented implementations while still complying with the specification.

VIII. Distributed Real-Time Tests

A ("soft") real-time, distributed, and cross-continental demonstration was conducted on July 13, 2001 in which a CMM-driver specification compliant file was used as input for sending commands to LK Metrology in Darby, United Kingdom (UK) from the National Institute of Standards and Technologies (NIST) in Gaithersburg, Maryland, USA, where the commands were used for inspecting the test artifact. A user-controlled pan/tilt/zoom camera was also integrated into the environment with a web-based video server to allow the inspection program to be viewed at either location. The demonstration used only one inspection plan test file.

This demonstration helped us improve NIST testing utilities and showed that distributed testing is not only possible but efficient and beneficial to any future testing events.

IX. Future Work

Depending on the needs of the CMM industry, we may create and perform explicit validation tests. Additionally, more precise metrics need to be defined in concert with the CMM community of users and vendors. Our current set of metrics (Section V-C) is preliminary only.

The current set of testing utilities is focused on the CMM-driver specification detailed in [4]. Based on a recent study and comparison of the two specifications [6], it is clear that the two specifications have more similarities than differences. It currently appears that the community of CMM system users and vendors is moving to integrate the two specifications into one while using the I++ DME-Interface specification [1] as the baseline. NIST is presently performing the modifications necessary to make the current testing utilities compatible with the I++ DME-Interface specification.

Perhaps the most important single item lacking in our test suite is to develop a comprehensive response simulator for use in application implementation testing. This may also be the most challenging part of test suite development. We also need to discern whether to couple our response simulator (which includes the response rules) with high-level test inspection plans, *e.g.* in DMIS, for use with the CMM utility test software. Also, whether test result logging code needs to be compiled with application level implementations. The questions and problems associated with this decision are discussed in Section V-A.4.

References

[1] Hans-Martin Biedenbach [Audi], Josef Brunner [BMW], Kai Gläsner [DaimlerChrysler], Günter Moritz [Messtechnik Wetzlar], Jörg Pfeifle [DaimlerChrysler], and Josef Resch [Zeiss IMT], "I++ DME-Interface," Release 0.99, 2002.

[2] Martha Gray, Alan Goldfine, Lynne Rosenthal, and Lisa Carnahan, "Conformance Testing," *http://www.itl.nist.gov/div897/ctg/conformProject.shtml*, 2000.

[3] Consortium for Advanced Manufacturing - International, *Dimensional Measuring Interface Standard*, Revision 3.0, ANSI/CAM-I 101-1995, CAM-I, Arlington, Texas, 1995.

[4] David Smith [LK Metrology], Lutz Karras [Zeiss IMT], Michel Penlae [Xygent], and William Wilcox [Wilcox Associates], "CMM-driver Specification," Release 1.9, 2001.

[5] John Horst, "Architecture, Design Methodology, and Component-Based Tools for a Real-Time Inspection System," Newport Beach, CA, 2000, 3rd International Symposium on Object-Oriented, Real-Time, Distributed Computing (ISORC).

[6] Thomas Kramer and John Horst, "A comparison of the CMM-driver Specification Release #1.9 with the I++DME-Interface Release 0.9," 2001.