Cruisin' and Chillin': Testing the Java-Based Distributed Ground Data System "Chill" with CruiseControl

Kathryn F. Sturdevant
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-1147
Kathryn.F.Sturdevant@jpl.nasa.gov

Abstract—This paper describes the design of the development test environment for the Mission Data Processing and Control Subsystem (MPCS), code-named "Chill." MPCS Chill is currently in development to support the Mars Science Laboratory (MSL), scheduled for launch in 2009. Chill is a Linux-based ground data system which includes both telemetry and command functions. The development test configuration consists of five levels: unit testing, end-to-end testing, user interface testing, external interface testing, and installation/deployment testing. This paper will focus primarily on the automation of the lowest two levels, unit and end-to-end testing.[1][2]

MPCS Chill's continuous integration process is provided by its adaptation of CruiseControl, which is an open source framework for a continuous build process. CruiseControl is configured on the dedicated build machine, a Linux workstation, which is the target platform. Chill has configured CruiseControl into two project loops: the first fetches the latest version of software from a central repository, builds it, and performs unit tests (JUnit). The second loop runs scripted end-to-end tests, which are performed on the results of the first build. Results are reported via email notification and a web interface provides the details of the current and previous builds for each loop.

The evolution of test definition is: requirements feed into design, design leads to use cases, and tests are derived from use cases, thus leading to the mapping of tests to requirements. Unit tests operate on internal components, while end-to-end tests operate at a higher level of abstraction and therefore can be traced to requirements. The evolutionary process ensures that we are testing to requirements.

MPCS Chill presents a model for testing Java-based distributed ground systems in a semi-automated manner. The MPCS model is highly applicable to other projects looking to automate their testing in addition to achieving continuous integration.

1

[1] 1-4244-0525-4/07/$20.00 ©2007 IEEE.

[2] IEEEAC paper #1470, Version 3, Updated December 14, 2006

TABLE OF CONTENTS

1. INTRODUCTION

MPCS Chill program set is the ground software to support the test and flight environments for MSL, which include the flight software workstation test set, the spacecraft testbeds, the Assembly Test and Launch Operations (ATLO) environment, and the operations environment. It has both uplink and downlink components, thus enabling both command and telemetry functions. MPCS/Chill processes the flight data via telemetry packet extraction, decommutation, channel and event processing, and product generation using XML-based telemetry dictionaries. MPCS is developed in Java and runs on Linux workstations.

This paper will discuss the MPCS development testing methodology. For more information on MPCS development, please see the corresponding IEEE paper on MPCS Development [1]. Additional test teams exist to perform integration and user acceptance testing. Their roles will be defined but the details are outside of the scope of this paper.

MPCS Chill is currently in development and therefore MPCS Chill testing is also in development. This paper is an effort to capture our current testing methodology as implemented and as planned to date. MPCS Chill invokes an Agile development methodology, so flexibility is a high priority. This methodology allows MPCS Chill to adapt to evolving requirements as it integrates with flight software test set environment, followed later with the delivery to the spacecraft testbed environment.

The MPCS development team consists of 5 developers at 3.25 total time (only two developers are full time). The development time from start to installation on an MSL testbed was 2 years. MPCS was integrated with the flight software testset in November 2006.

The MPCS applications are independent applications with their own JVM communicating via a JMS Bus. The MPCS Chill Architecture is diagrammed in Figure 1. The main application components are a downlink processor, an uplink processor, a database, and a monitor utility. Some tools can be run both via command line and graphical user interface (GUI) as appropriate. MPCS Chill is being developed with the intent that "all" errors are detected and reported. The processes are designed so that they can be tested separately for performance.

Because MPCS Chill is being developed in parallel with MSL Flight Software (FSW), real MSL test data does not yet exist. Simulation data is provided by the Mars Exploration Rover (MER) project, and is being used for testing. MER data is a close, but not exact, match to MSL data.

This paper describes:

(1) Test Design Philosophy

(2) The methodology of Unit Test Design

(3) The methodology of End-to-End Test Design

(4) Other Test Types: XML Validation, Graphical User Interface, External Interface, Deployment, Integration and User Acceptance Testing

(5) Development/Test automated flow

(6) How the project uses CVS and CruiseControl

(7) Issues and status to date

(8) Future plans

The project makes use of Concurrent Versions System (CVS) for source management of the development code, test code, and data sets.

The goal is to build an integrated development and test

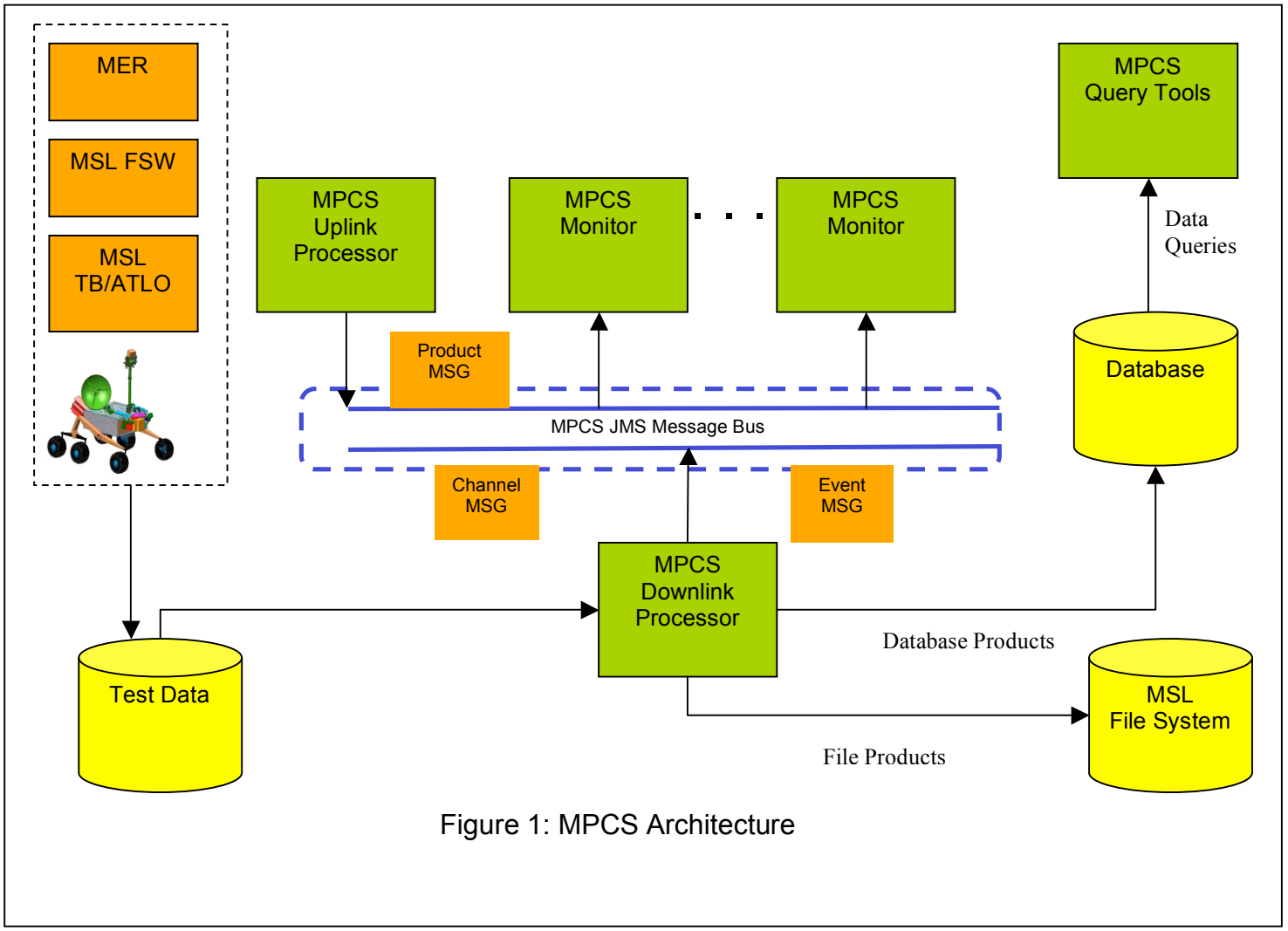

Figure 1: MPCS Architecture

environment with the aid of automation in an effort to reduce development and integration concerns by building and testing often, thus insuring the regularity and consistency of testing.

2. TEST DESIGN PHILOSOPHY

MPCS uses an Agile development model so unit tests are developed alongside the software and therefore built and tested along with each development build. End-to-End tests are built alongside the software, but have their own location in CVS and their own build cycle. They test the most recent build of the software. Building tests along with the development insures that the software and its corresponding tests remain consistent between code changes and between developers. If something gets out of synch, the build will catch the inconsistency by failing the test and all of the developers will be notified. The problem can be fixed quickly and the builds will continue. All test cases built during the course of development will be folded into the automated regression test suite to be used for verifying and validating the final release.

There are essentially five levels of tests that will be described in this paper, namely:

- Unit Test
- End-to-End Test
- Graphical User Interface Test
- External Interface Test
- Deployment Test

The goal is to implement each test in the lowest level possible to test it appropriately. If a test can be adequately resolved by a Unit test, then it will be built at that level. If not, then an end-to-end test, on up until it is suitably demonstrated. Emphasis will be placed on explaining unit and End-To-End tests as those are more geared toward automation.

Unit tests are the "white box" tests that the developers create to test the internal components down to the lowest level. Unit tests are created alongside the application as modules are identified for testing. Since the development task is in Java, these are implemented as JUnit tests, which can be executed both manually as part of development and automatically as part of each software build. JUnit classes are part of the standard Java library set used by MPCS.

End-to-End testing can be thought of as the "black box" tests, which further validate the components, but at a higher level of abstraction. End-to-End tests were identified based on the requirements: each requirement was analyzed and how to test it was determined. Logical groupings of those requirements defined the End-To-End tests. Some of the simpler tests could technically be identified as "unit" tests, as they relied on only one module. However, since they are to be distinguished from JUnit tests (which are at a lower level of abstraction), and they can be automated and run with each build, they fit more logically with the End-To-End tests. Most End-To-End tests involve more than one MPCS module.

The remaining tests are all manual tests, including the GUI testing since an automated GUI test tool is not anticipated at this time. User interface testing verifies the use cases, the anticipated use of the software, and ensures that the human-computer interactions are "friendly" and robust. External interface tests verify that data format and specifications are properly implemented. Deployment tests verify that an MPCS workstation is properly configured with the correct versions of MPCS and third party software (TPS).

In general, test cases and procedures are initially developed and executed manually. The tests will be added to the automated regression test suites as appropriate and when possible to verify and validate the software development build or software release.

Test cases and procedures are developed based on software requirements allocated for each major development phase, anomaly fixes or new software change requests. Test case development is a continuing process so the test suite will continue to grow over time and involve both requirements testing and tests initially based on anomalous results or behavior. Bug reports are filed for test failures and new tests will be developed to address those failures and ensure they are repeat tested as part of a test suite. Both testers and developers generate bug reports for failures.

The approaches to designing and implementing the unit and End-To-End tests are described below, with summary information of what User Interface, External Interface and Deployment testing will involve.

3. JUNIT TESTS

MPCS employs the Agile development method that entails software development and test as a closed-loop process. Ideally, Unit Tests that are needed to test the lowest level functionality of a product under test will be developed even before the product is built. Since the MPCS software suites are mostly java-based components, the JUnit technology is utilized for test case construction. The software codes for these Unit Tests are kept in the same CM repository as the product under test so that they can be executed every time the product under test is re-built to ensure system stability. Periodic team code reviews ensure that the Unit test cases have covered the critical aspects of the module under review. The goal of the MPCS unit tests is to cover the following:

- Component and data interfaces, validating a "reasonably comprehensive" range of inputs and outputs
    - The term "reasonably comprehensive" includes in-range and out-of-range values where applicable. For example, where valid numeric inputs range from -X to X, the minimal results to be tested include:
        - val < -X
        - val = -X
        - -X < val < 0
        - val = 0
        - 0 < val < X
        - val = X
        - val > X
- Where component behavior is configurable (e.g. through XML configuration) execute and validate component behavior, modifying each configuration parameter
- Test failure modes
- If there are multiple ways to build the same product, build product each way and compare
- Specific unit tests for any user-discovered bugs

Unit tests are not required for simple setters/getters or where there is no logic behind the execution of the method. Code inspections are intended to catch this type of error.

XML Validation will be discussed in Section 5 "Other Tests."

Unit tests should not exercise a significant portion of the system - those should be performed in the End-To-End tests that will be described in the next section.

The unit tests, implemented as JUnit tests, can be though of as the "white box" tests—testing the code from the inside, as opposed to external interfaces. The testing is on the module level—testing the components that make up the executables, as opposed to the executables themselves, which are tested in "End-To-End" tests. The JUnit tests are built by the developers alongside their operational code, and often before the operational code. This improves the quality of the code by exposing issues related to internal code interfaces early on, while maximizing the coverage of testing.

Additionally, the JUnit tests are the entry point for code reviews—they give a good view of what the module does and how it interfaces with other modules.

The JUnit tests were implemented in Java in Eclipse IDE platform. This allowed the team to take advantage of JUnit libraries, including superclass handling for unit tests, and the ability to "thread" unit tests, meaning to run them all together.

When identifying JUnit tests, the developers used these categories as guidelines: logical testing, data processing, message handling, and coverage of inclusive requirements. The unit tests are located in a "test" subdirectory for each module. Thus, they are part of the same CVS tree.

The software build and the JUnit test execution are paired in the same build file. First, the software is compiled and then the JUnit tests are performed. The details of the CVS trees and CruiseControl build loop are described in Section 6.

4. END-TO-END TESTS

End-to-End tests execute a "thread" of control through multiple modules. These tests can be thought of as "black box" or data-flow tests, which can verify data interface agreements. End-to-End tests deal with the input and output of modules, the executables as a whole, and often require testing the interfaces between modules, and the data flow in and out of one, or between two or more.

Due to performance considerations, End-to-End tests are designed to test many requirements with every set of data processed so as to make the best use of processing time. The requirements are grouped logically into tests. Some tests will address only one requirement while it might be appropriate for another to test cover more than 50.

The methodology used to define the requirements is discussed in this section along with the different categorizations of End-To-End tests that evolved from this analysis.

Designing to Requirements

The End-To-End tests were designed based on a thorough review of the MPCS Phase 2 requirements. The review began with 276 requirements. The requirements were reviewed for:

- text
- testability
- truthfulness compared to implementation

The entire team was solicited for requirement text clarification and implementation feedback.

The requirements were sorted based on these criteria:

- subsystem vs. system level tests
- automated vs. manual testing

After review, some were designated to be text only (not a requirement—an artifact of the way the requirements were fed into our database utility). Other requirements were reassigned to later project phases. The test team identified

92 requirements for rewriting, which included changes to text (for clarification), removing duplicate requirements, or splitting requirements that had been incorrectly paired into one. In general, the system level tests required manual testing, so although the split between subsystem and system needed to be identified, the definitive split for this phase is automated vs. manual testing. The initial split of the requirements between automated and manual is roughly 50/50, with slightly more in automation and more expected to migrate over time.

Cross-check with Developers' Lists of Input/Ouput

The developers defined early End-To-End test cases, which identified input and output tests to be performed on each chill application. These lists will be compared with tests generated based on requirements and can supplement the tests to a further level of detail.

Basic End-To-End Test

The basic End-To-End tests involve a single application and test the existence, usage, help, and version. These could technically be considered unit tests since they involve one application, but because they are at a higher level of abstraction from JUnit tests, and can be automated, they have been logically grouped with the End-To-End tests. Additionally, standalone performance tests for each module would be considered basic.

Building an End-To-End Test

The requirements reviewed above were placed in logical groupings, which were then translated into tests. These tests can be though of as the positive tests, in a sense, as they identify how the system is to be run. An additional type of test was determined necessary, and those were nicknamed "boundary tests."

Boundary Testing

Boundary tests focused on misuse and abuse of the applications, such as choosing parameter extremes or entering command line options differently than anticipated.

All of these End-To-End tests were being developed with the intent of scripting. The focus for this phase was to run them on the command line, and capturing the command line text for later conversion to a script. Note also that the goal is to identify a subset of these End-To-End tests to be run nightly against the latest build of the day. Not all of these tests will be appropriate for a nightly build, so some will be reserved for a separate periodic test of the system.

5. OTHER TESTS

Other pertinent tests necessary for a thorough verification and validation of the system are summarized in this section.

XML Validation

There are two steps in the MPCS XML validation process. The first is to run a validation utility on the file to check the XML format. The second step is to write utilities with knowledge of our data formats that can review the files at a higher level of abstraction. The third possible step, verifying that the configuration makes sense, would require too much knowledge engineering and development for the size of this task. This level of validation will therefore be performed via testing.

Graphical User Interface Tests

The Graphical User Interface (GUI) tests have roots in two areas. First, they are the End-To-End tests that were derived from requirements and were determined to be unsuitable for scripting. Second, they are derived from the use case scenarios: how an actor/operator would use the system. The MPCS GUI components will be tested manually as there is no plan for employing an automated GUI test tool in the near future. User case scenarios are usually created as a result of meeting with end users. A use case scenario is a description of how an end user will operate elements of the system. They are documented on the MPCS Wiki. The MPCS test engineer will perform the GUI tests by reviewing the steps described in each use case, and verifying that the software GUI performs as described. Where it is applicable, user interface tests should also be traced to the MPCS software requirements.

External Interface Tests

External Interface Tests are designed to verify and validate that the data interface specifications are well understood and are implemented correctly by the MPCS program set. These tests can be performed with one of the following methods:

- Use raw input data file (either real or simulated data)
- Install and exercise the external system on the MPCS test bed host computer if possible
- Coordinate with external system to get live data feed

Planning for live external interface testing requires scheduling test time with flight software development.

Deployment Tests

Deployment Tests are designed to validate the MPCS software installation and to verify that the target host computer is configured correctly with all the required third-party software needed to operate the MPCS program set.

In the test environments, the Deployment Test is executed to verify that all the correct software versions are deployed and installed correctly. These tests are (eventually) automated and will be executed by the test engineer before starting

daily tests. The deployment test suite will make use of appropriate tests from the automated thread test suite.

Other Test Teams

While the primary focus of this paper is on the development testing, other test teams exist and perform a vital role. The Integration and Test (I&T) team is currently responsible for assisting the Flight Software Test Set environment integrate with MPCS Chill. When the testbeds come online, the I&T team will transition to support them. The I&T team works closely with the MPCS development team as problems are found and new deliveries are made. The I&T team gets the first crack at real MSL data, which will eventually feed back to the MPCS Chill developers as sample test data. It also is in the best position to do performance testing on the system.

The user acceptance test team will verify and validate MPCS from the project standpoint. This team is appropriately independent of the MPCS Chill development test team. It can further the MPCS Chill testing by evaluating the software in a manner closest to how the users will be running it, for example, one downlink processor running with 20 monitoring utilities.

6. CVS AND CRUISECONTROL

The MPCS Build and Test cycle is diagrammed in Figure 2. The task employs the commercial-off-the-shelf (COTS) utility to help manage task "to do" items and bug reports. The developers modify code based on these items and check them into CVS. The dedicated build machine is a Linux workstation, which is configured to run CruiseControl. Using the CVS hooks provided by CruiseControl, the system is setup to detect changes to the "chill_gds" CVS tree and checkout a new copy, build, and run the JUnit tests after the check-ins are performed. CruiseControl was configured to send email to the development team for broken and fixed builds, while the test and development leads as well as any developers who did CVS checkins also received the email notifications for successful builds.

The MPCS development is in the "chill_gds" tree in CVS along with its corresponding JUnit tests. A secondary CVS tree "chill_gdstest" contains the scripted End-To-End tests and the datasets processed by these End-To-End tests.

There is a secondary test loop for "chill_gdstest," with the current plan being to kick it off as an evening build on the latest "chill_gds" build of the day. The "build" in this case is actually a script, which checks out the latest chill_gdstest tree from CVS and executes all of the tests within it using the executables from the latest chill_gds build. The tests are programmed as shell scripts. The success/failure of each test is reported via a web page and any unsuccessful results are saved and displayable on the web page for analysis. Like the chill_gds CruiseControl loop, the chill_gdstest loop is also configured to report the results via email.

The chill_gds build loop has been successfully deployed for several months of the Phase 2 development cycle. It has enabled us to maintain a working version of the software for a majority of the time, as the developers are notified immediately if they alter the CVS tree in such a way that it no longer builds or passes JUnit tests for a "clean" version of the source tree.

The chill_gdstest loop has been prototyped to show the basic capabilities of the automated test loop and tests basic capabilities such as usage, help, and a simple data processing test.

7. ISSUES AND RESULTS

Software versus End-To-End Test Development

The current automated End-To-End test suite consists of basic tests such as existence, usage, and a simple processing test that verifies output. Though it was planned to have more sophisticated tests running nightly on the latest build of the day, the Phase 2 development was too volatile to make much End-To-End test development feasible. When the system volatility subsided, the task of mapping tests to requirements began, based on the software that existed at the end of the phase.

Requirements Review

During the exercise of mapping requirements to tests, it became clear that the requirements for Phase 2 were not as well worded as they could have been, and they required a fair amount of rewriting to make them clear and testable. Because of this experience, we opted to review the requirements for Phase 3 early on and reviewed them for clarity as well as testability. It is our expectation that tackling this review early on will be beneficial to both the development and test efforts.

Nightly Build versus Next Phase Development

Due to the limitations of the CVS branching capability, the project is considering migrating to Subversion. The motivation to branch comes most often at the end of a development phase, when testing or end users are finding problems with the software, while in the meantime, developers are working on the next phase. The advantages (and disadvantages) of a lightweight version control utility were understood at the project inception and this is a known tradeoff. Our method of branching was successful because we opted to limit branch releases to only essential bug fixes so as to limit the amount of duplication necessary to install fixes in both the branch and the trunk. Additionally, branching provided a more stable release to the flight software team as they received only what they anticipated and not potentially new, untested features related to the next phase development. Two branch releases were made for Phase 2.

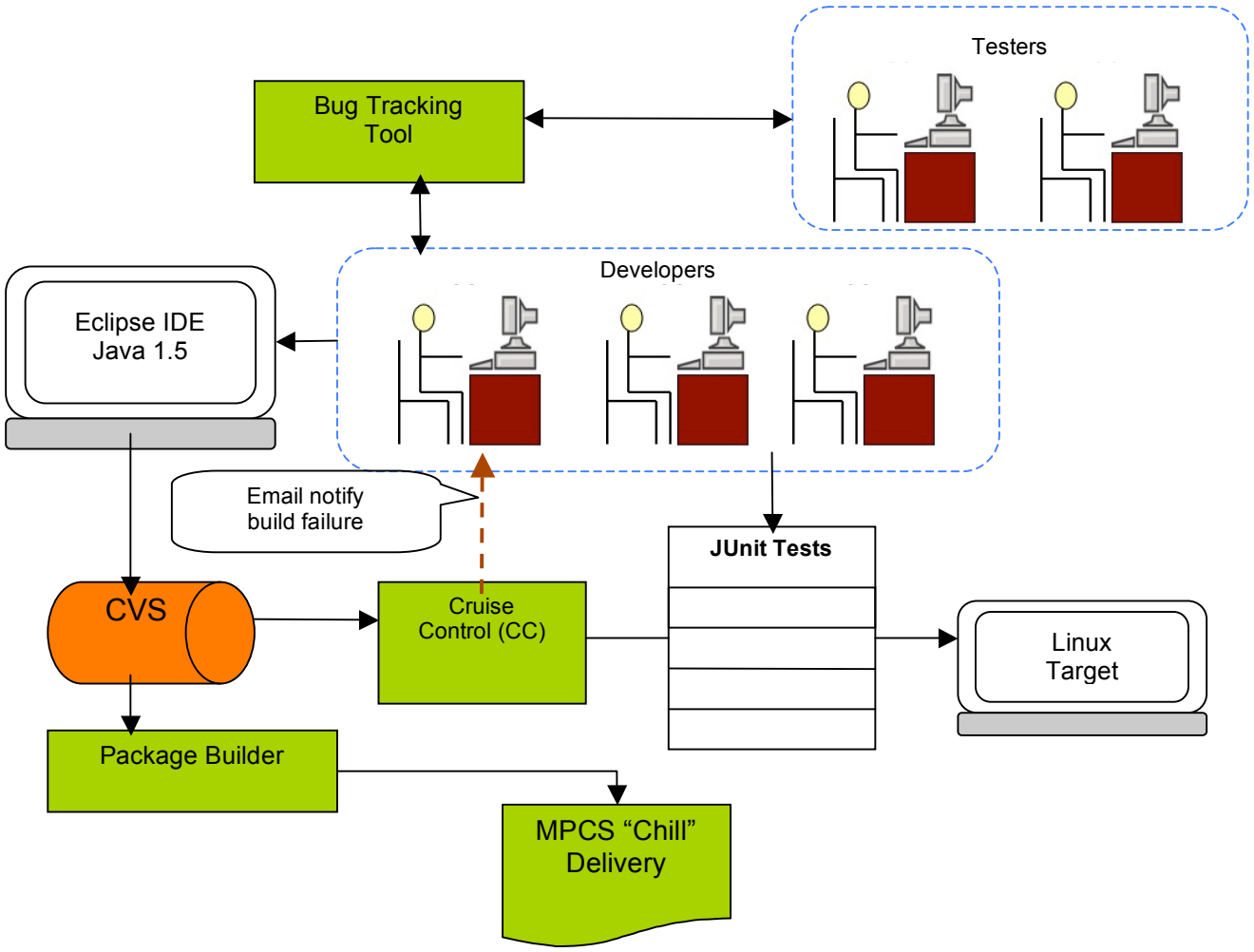

Figure 2: MPCS Automated Build/Test Process

Build Statistics to Date

From June 2006 to December 2006, we had 256 successful builds and 30 failed builds.

Regarding out of synch build conditions, MPCS Chill development has experienced only a few to date and most were due to developers not starting with clean versions from CVS. The limited number of conflicts could be in part due to the limited number of developers and the manner in which tasks are assigned regarding overlap.

The CruiseControl report has been instrumental in ensuring a quick turnaround for broken builds. The report shows the exact failure as well as the file changes since the previous build, which allows for a quick analysis of the problem. Turnaround time to fix a build has ranged from minutes to a few hours.

Performance Testing

Development testers can test performance requirements. Additional performance testing must be done in the testbed environments by the user acceptance team. The latest performance testing by developers to date was to process 2 weeks of MER cruise data (87.3 Mbytes) in 40 minutes. Two memory leaks were discovered using a commercial Java utility. More performance testing will be performed after our installation in the testbed environment in January 2007.

Performance tests are planned each release to monitor and detect potential changes due to additional features.

Automated Build Environment

To evaluate the success of the automated build environment, MPCS Chill is compared to a previous task that was similar in nature, but did not have an automatic build cycle. Less time was spent resolving integration problems with MPCS, than with the previous project. We believe we have been

very successful in this regard, and that allowing developers to resolve conflicts at check in or upon build failure reduces the number of integration problems.

Limited Test Data

The MSL MPCS ground system is being developed in parallel with the MSL Flight software. As a result, MSL test data is nonexistent. MPCS Chill is therefore developing with MER data by analogy until MSL data is available. Integration and user acceptance tests will be first to encounter real MSL data. Once MSL data is generated and available, it will be fed back to the MPCS development team to be made part of the data test set.

Command Line Options

In hindsight, more attention should have been directed toward defining the command line options for each application earlier in the development, and making them as consistent as possible between applications. This impacted the effort to automate tests because the options were in flux until late in Phase 2 development, and the tests were constantly being upgraded to reflect minor and major changes to the command line. In addition, early definitions could have provided more consistency between applications, which would also benefit usability.

An additional complexity regarding command line arguments was that the number of requested arguments grew once flight software testset integration occurred, and they wanted more flexibility than was originally anticipated. Since the goal was to have every application run either via GUI or command line, the number of command line arguments grew unwieldy. The new plan is to define these parameters in a configuration file instead of the command line.

8. FUTURE WORK

The end-to-end tests were run by manually in our previous integration period. We are now working to automate these tests so that they can be applied regularly to each build to ensure that changes do not adversely affect the system.

The next customer is the MSL testbed, which will take delivery of MPCS Phase 2 in January 2007. Phase 3 is scheduled for release in March 2007.

9. CONCLUSIONS

The Chill development and test efforts are a work in progress. CruiseControl has proven to be instrumental to the build and JUnit test cycle, as it continually integrates and validates the software updates. Our next step is to fully implement the CruiseControl thread test loop and exercise its capability to run the "nightly build," which will execute the automated thread test suite at night on the latest version of the software successfully built during the day.

Similarly the development test loop, the goal of this secondary test loop is to ensure testing consistency on a nightly basis so that, like development, problems can be quickly identified and therefore remedied before being allowed to compile.

We have verified our processes for requirements review and thread test development and we have enough mature thread tests to build an initial suite. As MSL data becomes available and as more requirements are implemented in each phase and bugs are fixed, new tests will be incorporated to address these additions.

Spurred by the need for cost efficiency, the MPCS development test task is challenged by a staffing level of 0.5. In order to meet our Phase 2 integration deadline, the I&T test engineer was brought in early to assist, thus increasing the staffing level to 1.5. Additionally, the developers tested during the pre-integration and integration periods, and were instrumental in defining the end-to-end tests. Our efforts were successful to date as only two additional branch deliveries were required to support the flight software test set environment.

10. ACKNOWLEDGEMENTS

The author would like to thank the MPCS Chill team members Jesse Wright, Martha DeMore, Daniel Allard, Brent Nash, Richard Borgen, Jordan Lei, and Quentin Sun of the Jet Propulsion Laboratory for their contributions toward this effort.

The work described in this paper was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

[1] Allard, Daniel, "Development of a Ground Data System Messaging Infrastructure for the Mars Science Laboratory and Beyond ," IEEE 2007.

BIOGRAPHY

Kathryn Sturdevant *is the lead test engineer for MPCS and a system engineer at the Jet Propulsion Laboratory with many years of experience in developing and operating ground data systems for planetary spacecraft missions.*