A high-availability, distributed hardware control system using Java

Albert F. Niessner

Jet Propulsion Laboratory, 4800 Oak Grove, Pasadena, CA, USA 91109

ABSTRACT

Two independent coronagraph experiments that require 24/7 availability with different optical layouts and different motion control requirements are commanded and controlled with the same Java software system executing on many geographically scattered computer systems interconnected via TCP/IP. High availability of a distributed system requires that the computers have a robust communication messaging system making the mix of TCP/IP (a robust transport), and XML (a robust message) a natural choice. XML also adds the configuration flexibility. Java then adds object-oriented paradigms, exception handling, heavily tested libraries, and many third party tools for implementation robustness. The result is a software system that provides users 24/7 access to two diverse experiments with XML files defining the differences.

Keywords: high-availability, distributed computing, hardware control, Java

1. INTRODUCTION

The original problem started with the needs of a single coronagraph experiment with softly-defined motion control requirements and one software requirement: The control software had to integrate with the scientist's analysis environment. When the Lyot coronagraph[5] first started, the preferred analysis environment was SPICA*. However, SPICA was not the only environment used; Matlab,[1] IDL,[2] and Octave[3] were also used. At some point, SPICA was dropped for IDL[2] and Octave[3] for SciPy.[4] Currently, SciPy[4] is the preferred and nearly exclusive analysis, command, and control environment. Since the software system is required to be integrated with the analysis environments, it has to adapt and change as the users adopted newer and different analysis environments.

As the experiment matured, the analysis environment was not the only change affecting the command and control software. The motion control changes rippled through the control software making it apparent that a new software requirement was for quick and easy configuration. Hence, the second software requirement: The software system must be rapidly adaptable to new motion control hardware. As the experiment continued to mature, new control hardware was added while existing hardware control was re-purposed increasing the complexity resulting in the unintended consequence that some of the hardware cannot co-existing in the same computer. The need to have more than one computer involved in controlling the experiment plus the increasing desire to run the testbed from anywhere the world formed the third requirement: The software system must be distributed to allow as many computers as necessary cooperate for complete control of the experiment without geographic location restrictions. The third requirement immediately led to the last requirement: The software system must make the testbed accessible 24/7. These requirements were used to formalize the resource distribution shown in Figure 1 on page 3.

During all these changes, the PIAA coronagraph[6] was introduced. It has a different optical layout and different motion control than the Lyot coronagraph. However, these two testbeds use the same software to command and control them with different XML files describing them.

*SPICA is an internally created, maintained and used analysis environment.

From a software perspective the interesting problem became; how does one architect, design, and implement a high-availability, distributed control system that is highly adaptable to three degrees of freedom (hardware, user environment, and experimental layout) while providing consistent behavior. The following sections cover how each of the goals (configurable, distribution, and high-availability) were met.

2. CONFIGURATION

XML and XML schema were used to solve two of the three degrees of freedom: experimental layout and user environments. For both of these degrees of freedom, the bulk of the work is designing and implementing a Domain Specific Language (DSL). The diversity of environments added the problem that the implementation rules for the DSL were always changing because the environments are so different in features and richness. Rather than trying to develop a language neutral lexical parser and semantic checker, XML schema was used to define the DSLs for both experimental layout and user environment interactions. JAXB[8] is used to validate and convert the XML to Java objects while generateDS[9] is used for Python and xsd[10] for C/C++.. Between the schema, validation, and conversion they act as a lexical parser and semantic checker. The benefit of XML is that everyone's favorite text editor or analysis environment will allow them to manipulate text, making support across environments simpler.

The last degree of freedom, hardware, was solved with the XML schema for the experimental layout plus a plug-n-play architecture. The XML would configure the final details for the driver of a piece of hardware, like the serial device name, baud rate, bits, parity and stop bits. The hardware drivers are designed and implemented with a plug-n-play architecture. The combination of these two allow for minimal driver implementation. Generally speaking, the plug-n-play architecture and XML configuration enables testing and adding new hardware into the testbed in just a few hours.

2.1 Architecture and Design

The XML schema for configuration was created so that the XML structure would mirror the logical and physical structure of the testbeds it was to represent. Figure 2 on page 5 show the schema design. The XML allowed by the schema describes a testbed as a collection of computers that contain logical structures (the component element in Figure 2) as well as hardware communication devices (the communication mechanism element in Figure 2). The logical structures are used to describe physical elements on the testbed such as cameras, translation devices, rotation devices, and sensors. The logical structures are also used to describe logical elements of a testbed such a PID or FSM. Even complex devices such as fiber source with five degrees of freedom are encapsulated as a single logical element. The logical structure and the communication hardware it depends on are tied together into the computer element in Figure2. For instance, a PZT used for translation motion may be controlled through a DAC on computer A. The XML for the example would have the computer A would containing a logical structure for the PZT as a translation stage and the DAC as the communication path to the PZT.

When there are multiple computers in a configuration, they are allowed to depend on each other so that commands across computers can be serialized, an attribute of the computer element in Figure 2. For instance, the user may want to move a stage prior to acquiring an image but the supporting hardware does not co-exist in the same computer. The computers involved in completing the task must then cooperate to make sure the requests are serialized correctly. Dependencies are allowed to be either unidirectional or bidirectional. Acquiring the image may wait for the stage motion but the stage motion may not have to wait for the image to complete. The configuration is used to describe the dependency relationship among all the computers required to control a testbed.

An XML file describing a testbed is parsed and converted to objects for the control software. To continue the example started in the first paragraph with the software running on computer A, the XML is parsed and converted to Java objects

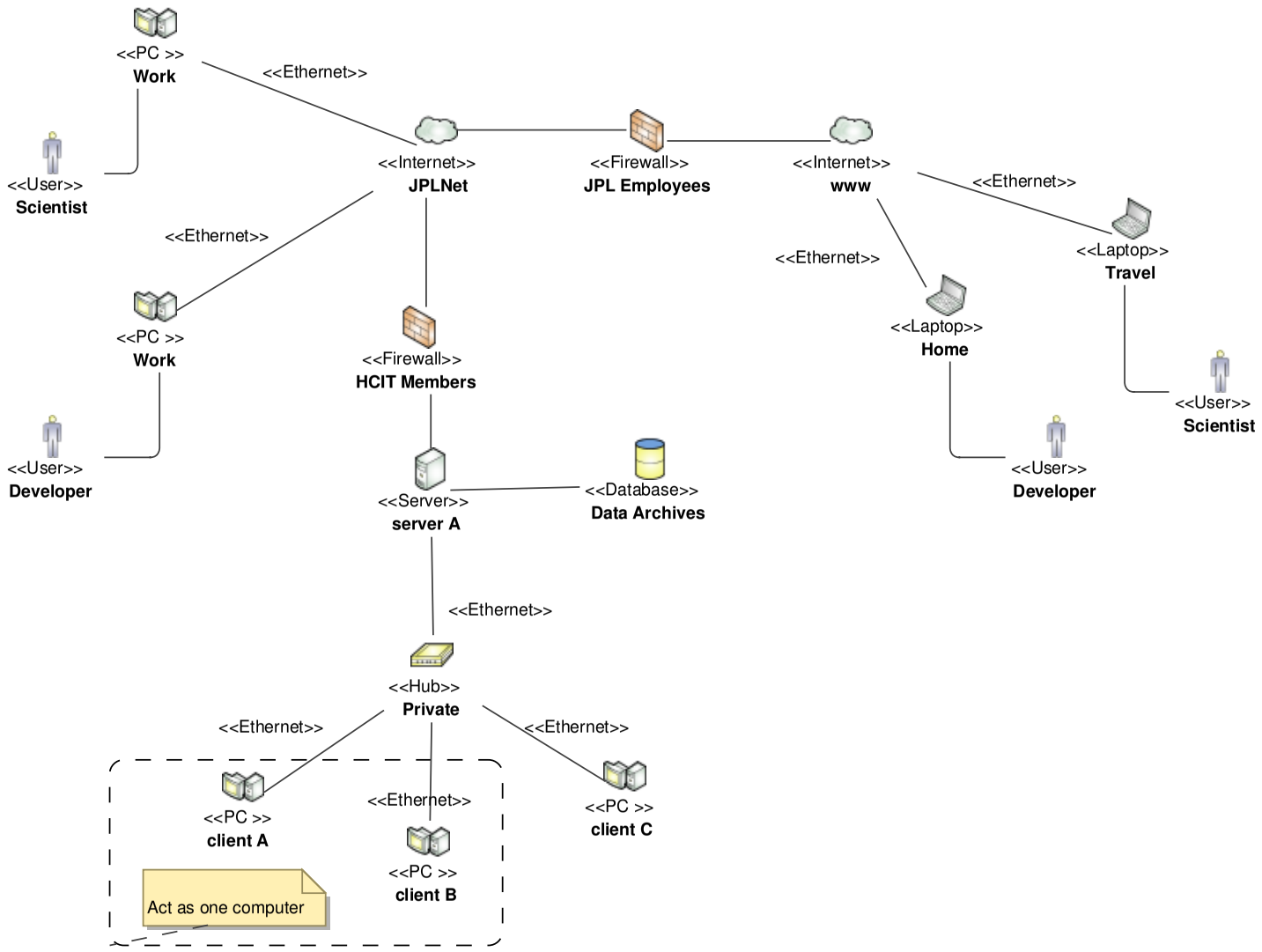


Figure 1: Geographic Distribution

containing information about a DAC residing in computer A and a PZT that is moving something on the testbed. The resulting objects are used to route information among generic elements as well as tie specific elements to specific channels as shown in Figure 3b on page 7. In other words, it is the XML that converts a collection of Facade and Strategy patterns[1] to be shaped into a control system for a testbed that can perform the required specific tasks.

Unlike the XML schema for configuration, the schema for the environment interface was created so that the resulting XML would be trivial to parse without the help of SAX or DOM parsers. The environments being used for commanding testbeds and analyzing the results are not always amenable to using XML. However, all of the environments can handle strings, the problem is to keep the schema simple so that it is trivial to parse. The schema allows for a building a command in pieces and then executed and for collecting a response. The schema specifically keeps the allowable commands and responses separate from the tags so that it does not change even though the testbeds do.

Ultimately, the use of XML allows for a highly configurable system at both the command layer and the hardware layer. The robust nature of XML, the millions of users and debuggers and many made tools, make it work so well for this application.

2.2 Tools

For configuration, which tends to be large and complex, the tools JAXB,[8] generateDS,[9] and xsd[10][10] provide robust parsing, validation, and conversion to objects. The availability of these tools meant the effort in implementing the design was in structuring the Java using patterns such as Facade and Strategy[1] to make plug-n-play software elements rather than in developing tools to parse and validate domain specific languages. For the person writing the configuration XML, it also means a simple way of validating the configuration before deployment. In both cases, it results in the early detection of problems making them simple and inexpensive to fix.

3. DISTRIBUTION

XML, XML schema, and architecture were used to solve the myriad of issues associated with distributed computing. While the requirement was to distribute the software control system across a set of computers, there were four main goals that further restricted the solution space. First, the communication messages should be human readable. Second, it was important to be able to intercept and capture the entire output stream and write it to a file for later analysis. Third, name resolution of computers should be left to DNS and the people configuring the testbeds. Fourth, connecting to the software system from any language should be trivial. These desires eliminated all of the off the shelf solutions that were examined including the two most obvious: CORBA[11] and Java RMI.[12]

3.1 Architecture and Design

Figure 1 on the previous page shows the physical separation of equipment and tasks. More importantly, it shows the many users and developers that monitor the testbed plus a single user or developer commanding it. The geographic distribution of the monitoring and controlling tasks make the well known Model-View-Controller (MVC) architecture the obvious choice. Mapping the MVC names to the other terms being used, the observers or monitors of the testbed are the View, the single commander is the Controller, and the testbed control software, firmware, and hardware are the Model. The benefit of the MVC architecture is the decoupling of the three parts of the MVC. An unintended benefit of the MVC is the isolation of changes in the users environment from the model and changes in motion control requirements from the views and controller.

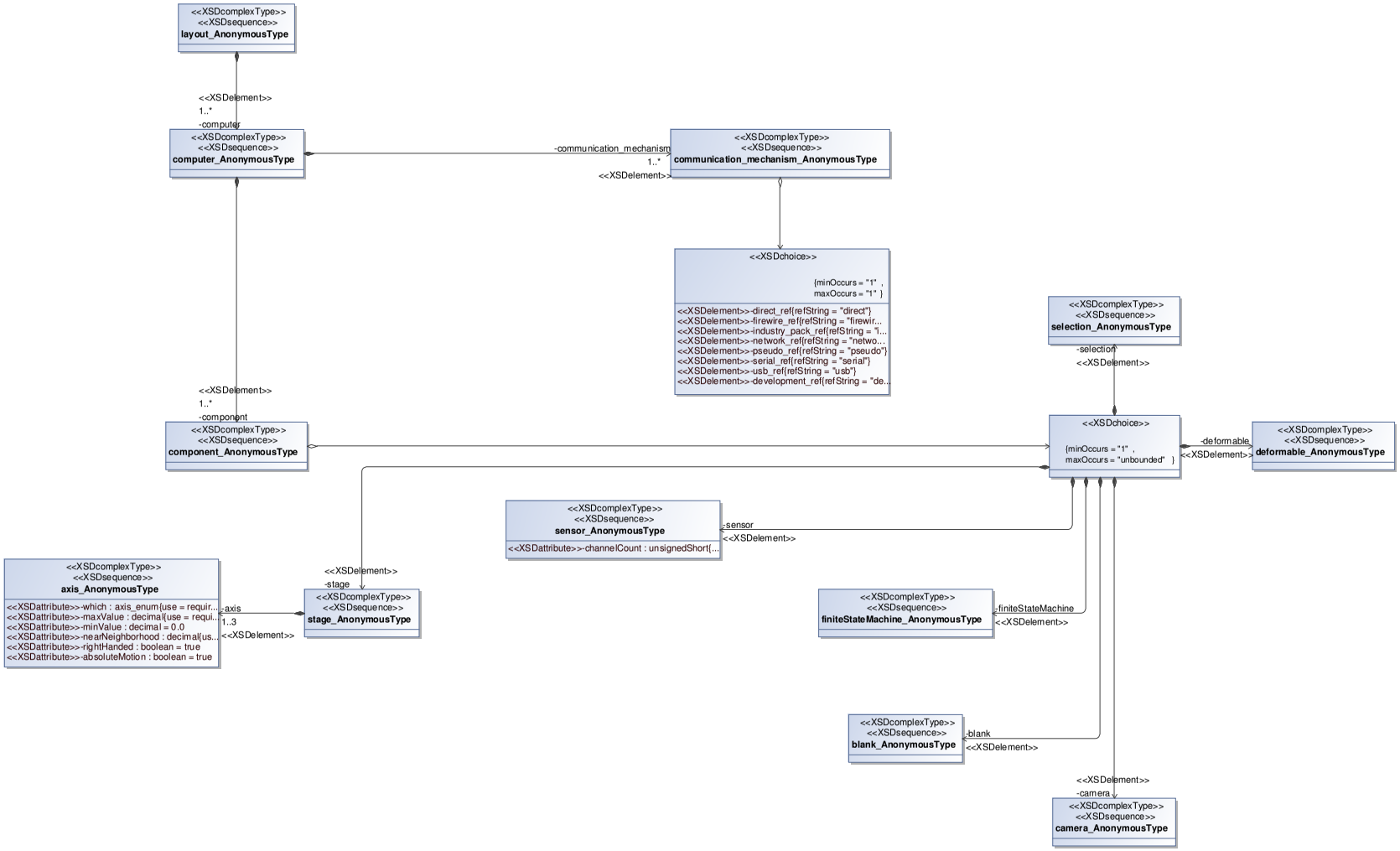


Figure 2: XML Schema

The traditional MVC connects the Model to the View and the Controller to the Model, but the distribution of Figure 1 requires a slightly different communication pattern. The Controller and View are simple entities that each execute within a single process on a single computer. The View receives its data through the state and data telemetry channels while the Controller sends commands through the command channel following the traditional MVC. The Model, however, may be composed of more than one computer breaking from the traditional MVC. The configuration XML contains a list of the computers involved that make up the Model and how those computers depend upon each other. The Controller and View use the same configuration XML making them responsible for connecting to each of the computers in order to be connected to the Model. The interconnect and the connection policy for the Controller and View allow the collection of computers to be treated as a single logical Model returning the architecture to the traditional MVC.

The messages exchanged among the MVC parts are XML and sent over TCP/IP. TCP/IP ensures that the messages are reliably delivered and the XML schema ensures the messages are reliably interpreted. Using the same tools used for the configuration, the messages are converted from objects to XML before transmission and then back to objects at the receiver. The View has the simplest communication protocol; it opens sockets to the Model's telemetry channels (both data and state channels) and then displays or records it. Since it is passive, any number of views may be connected to the system simultaneously. The Controller has a more complex communication protocol; it must connect to all the computers over TCP/IP before sending commands because there should only be one Controller connected at a time to ensure command serialization. If the Controller cannot connect to all the Model's computers, then it must back off and try again later. The computers that make up the Model use the telemetry data but over a private channel to prevent long delays from multiplexing to slower View channels.

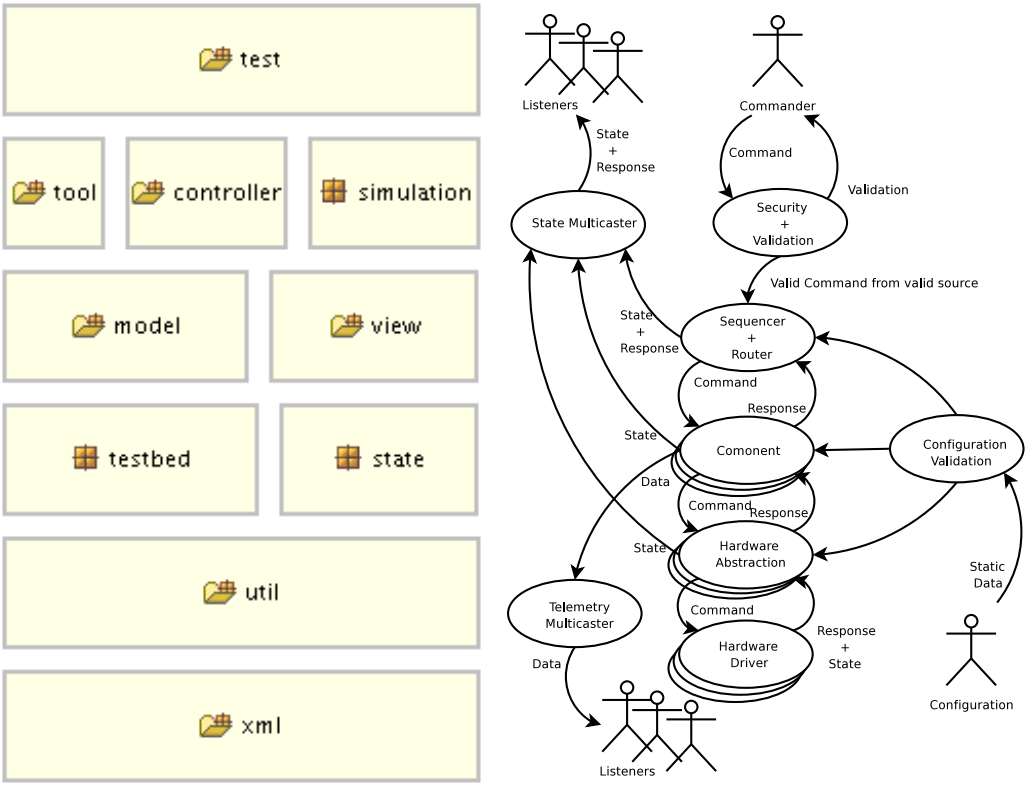
Beyond simply breaking the system into an MVC architecture, many other architecture choices were made to maintain the separation and independence of the MVC parts in order to realize the benefits of the MVC. The most significant choice was the plug-n-play architecture that applied to the MVC parts as well as all the constituents that make up the MVC parts. The plug-n-play architecture keeps the elements decoupled making distribution simpler. The other significant choice was to handle telemetry data the same way at all levels throughout the MVC. Uniform handling of telemetry data disconnected its generation from its use making it trivial to distribute for use.

The architecture choices of using XML and TCP/IP solved the most of the distributed computing issues. TCP/IP solved the reliable delivery problem while XML solved the robust message problem. The plug-n-play architecture choice keep the elements separated and independent allowing them to be easily distributed.

3.2 Tools

There were three primary tools used to make sure the implementation of the architecture met their desired goals. First, as with the configuration, JAXB and XML schema are used to generate the XML messages from Java objects at the transmitter and then validate and decode the XML to Java objects at the receiver. Second, design patterns are used to convert architectures to designs to be implemented. Third, design visualization and validation tools, like Structure101,[13] to verify that the implementation matches the design and architecture..

The plug-n-play architecture used several design patterns. The first step in allowing different implementation to be swapped for one another is to use a Java Interface to define the strategy of the Strategy Pattern.[1] Since elements, like the Model of the MVC, are very complex, implementations of the strategy also use the Facade Pattern[1] allowing the complexity to be hidden behind the facade. The Abstract Factory Pattern[1] is used to decouple concrete strategy/facade creation from those that depend the them. Choosing the specific factory through Java properties and reflection adds the extensibility for complete plug-n-play.



(a) Top Level

(b) Processing Flow

Figure 3: Architecture

The telemetry architecture incorporates the plug-n-play architecture and design patterns as well as uses several other design patterns. Telemetry data is passed through the system from the source to the many sinks, and the elements handling the telemetry data as it is passed to the sinks view all the telemetry data as being the same container making them plug-n-play. Once at the sink, they are no longer plug-n-play containers but rather unique data shapes. Directly extracting the data from these containers would couple the parts of the MVC and other plug-n-play throughout the software. Rather, using the Visitor Pattern[1] and a Strategy Pattern[1] for the visitor allows the data to be transformed from a generic container to a data shape that is easy to work with. These two patterns keep the telemetry data handling uniform through all three parts of the MVC while keeping them independent of each other.

To realize the benefits of these architectures and design patterns, the code must be correctly structured, which is to say that the class dependencies must be correct. Implementations do not always match the designs, and, when they do not, the design does not perform as expected.[1] Tools like Structure101[13] expose the structure of the software and their dependencies while verifying that the design and implementation match. Achieving the desired distribution did not happen on the first implementation because of unnoticed dependencies of the Controller and View depending on the Model. Figure 3a shows that the Model and View are independent but the the Controller still erroneously depends on the Model. While this paper is not interactive, the tool is, and it is possible to explore all the layers within a software system to validate its architecture and design. While exploring the dependencies of this software system, those that prevented distribution were corrected while the rest are currently being managed. An unintended consequence of using the design visualization and validation tools was to realize all of the design benefits but especially robustness and thus availability.

4. HIGH-AVAILABILITY

Java exception handling helps to solve the problem of high-availability. The set of software that existed prior to this work was not always available for use. There were many days that went by where the software simply did not work or crashed for very unexpected reasons. During the initial phases of design of this work, the failures of the initial work was examined to understand the failure causes. It appeared that the addition of simple exception handling would be sufficient to make the new system robust enough to provide nearly full availability. It was not to be the case. The real solution to robustness came from the long list of tools used to find, isolate, and remove software defects in the implementation.

4.1 Architecture and Design

Figure 3b on the previous page shows the processing flow of the system where the sources of commands and sinks of data and state are shown in use-case fashion with stick figures while the major processing entities are ovals. The command flows down from the user at the top through many layers generating responses at each of those layers to the hardware at the bottom. With respect to the processing flow between the sequencer and the hardware driver, the upper layer is responsible for understanding and acting upon the responses of the lower layer where exceptions are an acceptable response. Each parent layer is to handle the exception received from the called layer; it is acceptable to just pass the exception up the call stack. However, it is preferable to handle the exception as close to its origin as possible.

Handling exceptions as near to the origin as possible did help immensely with robustness but only for hardware exceptions. When the system was first built and deployed, a lot of time was spent finding the idiosyncrasies in the hardware. For instance, one piece of hardware returns an error code about 30% but succeeds most of the time. When the error condition occurs, it generates an exception in the hardware driver. The hardware abstraction layer has a better understanding of what is happening and simply tries again because of this known idiosyncrasy. The user is now able to have access to the testbed more frequently because time is not being wasted restarting a crashed system from ill behaved hardware. For problems generated at the hardware layer, the exception handling proved very helpful at adding robustness.

Where exception handling did not help was in the numerous software defects trapped in the system. As the hardware idiosyncrasies were cataloged and handled, the real robustness problems came to the fore. There were hundreds of defects lurking throughout the system causing it to generate exceptions frequently. The system would not crash because of the exception handling, but it would slowly degrade to a useless state because the exceptions did not match a known case. When a defect was hit, the exception would propagate to a handler that would misinterpret it or propagate to the highest level handler that would shut that component down as a last ditch effort. The rest of the testbed would still be available, but slow degradation still leads to a dead system eventually. In addition, the exception handling would hide the true defect. Between the shear number of software defects and the exception handling misinterpreting them as hardware failures, the availability was limited to just a few days.

Robustness and high-availability arrived after integrating static analysis tools, unit testing, and dynamic analysis tools into the development process. The two advantages of the static analysis tools are that it is is independent of the exception handling and it allows the defects to be corrected in parallel. The system is now available for months at a time with the availability being hardware limited.

4.2 Tools

There are many tools available for improving the robustness of a software system. The ones that had the largest impact on this software system were static analysis tools, testing with coverage, and dynamic analysis tools. While each of these tools can be used independently, it is was their combined use that significantly improved robustness. They were introduced

into the development process serially to solve the biggest problems in achieving the robustness goals, but serial integration into the work flow is not necessary. The dynamic analysis tools were first followed by the static analysis tools and then testing with coverage.

Two types of dynamic analysis tools were used, and they helped with performance based issues and threading. The first was a traditional profiler, like YourKit,[14] that highlights memory and CPU usage. The second was a thread analyzer that monitors a running system to watch the interaction among threads. When these tools were added, the availability went from hours to days. With performance high[†] and exception handlers working well, the software defects became the limiting factor to the availability.

Static analysis tools, like FindBugs,[15] helped with identifying the defects lurking in the software. When first analyzed, there were over a hundred defects identified throughout the software. Most of them have been eliminated and constant analysis of the software as it changes keeps the count low. The reducing the number of these defects improved robustness and availability because it allowed the exception handling mechanism to handle the exceptional event rather than just bad programming. When static analysis was added, the availability went from several days to weeks and even months. There were still defects in the software that static analysis just cannot find.

Lastly, the testing in place was reviewed and coverage was added to understand how to better find the remaining defects. Coverage tools, like EMMA,[16] mixed with testing suites, like JUnit,[17] give a view of what is being tested and what is not. The testing being performed goes well beyond standard unit testing. While some elements are simple unit tests, the first extended use of JUnit was to write a test suite that would instantiate all the implementers of a Java interface and verify that they follow the behavior contract that the interface defines. The technique found many subtle problems that were interfering with availability through the plug-n–play architecture. JUnit was also used to do full up or integration testing as well. The integration testing requires simulations of the hardware components including their known idiosyncrasies. The integration testing found the last of defects preventing high-availability. Interestingly, these three forms of test cover an amazing amount of the system but knowing what is not being tested is a problem. Coverage tools solve that problem easily. Unit, behavior contract, and integration testing plus knowing what has been and needs to be tested made it easier to eliminate the last of the significant defects in the software system resulting in availability that is hardware limited.

5. CONCLUSION

The distributed computing and configuration goals were achieved using XML and design tools. While the architecture and design are used to solve specific problems like plug-n-play, it is the use of design tools to verify that what is being built matches the design that the benefits to be realized.

The high-availability goal was achieved using design and implementation tools. Exception handling works well to capture hardware idiosyncrasies, but the implementation tools solved the software robustness problems. Static analysis, testing, and dynamic analysis tools all help to reduce defect count.

The result of this work is a high-availability, distributed Java command and control system that adapts to the needs of the user with little effort.

ACKNOWLEDGMENTS

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

[†]Here high performance means CPU and memory usage are at the expected minimum levels.

REFERENCES

[1] http://www.mathworks.com

[2] http://www.ittvis.com/ProductServices/IDL.aspx

[3] http://www.gnu.org/software/octave

[4] http://www.scipy.org

[5] Lowman, A.E., Trauger, J.T., Gordon, B., Green, J.J., Moody, D., Niessner, A.F., Fang Shi, "High contrast imaging testbed for the Terrestrial Planet Finder coronagraph," Proc. IEEE 4, 2156-2161 (2004).

[6] Brian Kern, Ruslan Belikov, Amir Give'on, Olivier Guyon, Andreas Kuhnert, Marie B. Levine-West, Ian C. McMichael, Dwight C. Moody, Albert F. Niessner, Laurent Pueyo, Stuart B. Shaklan, Wesley A. Traub, and John T. Trauger, "Phase-induced amplitude apodization (PIAA) coronagraph testing at the High Contrast Imaging Testbed," Proc SPIE 7440, (2009)

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides [Design Patterns: Elements of Resuable Object-Oriented Software], Addison-Wesley, Massachusetts(1998)

[8] http:///jaxb.dev.java.net

[9] http://www.rexx.com/~dkuhlman/generateDS.html

[10] http://www.codesynthesis.com/products/xsd

[11] http://www.omg.org/gettingstarted/corbafaq.htm

[12] http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[13] http://www.headwaysoftware.com/products/structure101/index.php

[14] http://www.yourkit.com

[15] http://findbugs.sourceforge.net

[16] http://emma.sourceforge.net

[17] http://www.junit.org