Modernizing Cassini: Approaching Agile After a Decade at Saturn

ANDREA CONNELL, Jet Propulsion Laboratory, California Institute of Technology

Software for a long-duration NASA flagship planetary mission faces a combination of challenges – maintaining legacy systems, complex modeling and algorithmic systems, limited funding, very low tolerance for risk, and heavy process requirements. Our team with in the Cassini Mission to Saturn has faced these obstacles while applying Agile principles, and we have learned some lessons along the way.

1. INTRODUCTION

NASA's Cassini Mission to Saturn launched in 1997, and has been orbiting the ringed planet continuously since its arrival in 2004. The mission will come to an end in September 2017, after thirteen years of collecting scientific data about the planet, rings, and moons. Throughout this period of exploration, software has been used to plan science activities, simulate the effects of those activities within the spacecraft, and then translate the activities into commands for transmission to the Cassini spacecraft. As we learned more about Saturn and planned new kinds of maneuvers, the software systems needed to be updated accordingly. These systems were created before modern architecture and development process frameworks were popular, and typical legacy software challenges were heightened in the limited-funding and risk-adverse environment of a flagship planetary mission. This paper will describe the evolution of teamwork, testing strategy, and procedures on our software development team over the years.

2. BACKGROUND

The Jet Propulsion Laboratory (JPL) leads the United States' efforts in robotic exploration of our solar system. JPL is a federally funded research and development center, managed for NASA by the California Institute of Technology. We have been designing, building, and operating spacecraft since 1958, when Explorer 1 became the first satellite launched by the United States. Today JPL has rovers, orbiters, and other instruments studying Earth, Mars, Jupiter, Saturn, and beyond.

The Cassini spacecraft launched from Cape Canaveral, Florida on October 15, 1997 to gather scientific data about Saturn, its rings, and its many moons. The Cruise phase of the mission took seven years to get to Saturn, and included gravity assists from fly-bys of Venus, Earth, and Jupiter. Saturn Orbit Insertion occurred on July 1, 2004 and Cassini's original four-year Prime Mission began. At the end of the Prime Mission, NASA approved a two-year extension called the Equinox Mission. Two years later the spacecraft was still healthy and making new discoveries, so another seven-year extension was added as the Solstice Mission. Mission planners knew, however, that this exploration of the Saturn system could not continue indefinitely. We are now nearing the end of the second extension and running out of fuel on the spacecraft, so we needed to find a fitting way to end such an historic mission. On April 26 2017, Cassini began making weekly dives between the innermost rings and the upper atmosphere of Saturn. This narrow gap – only about 1500 miles wide – is entirely uncharted territory. The dives, occurring at speeds up to 77,000 miles per hour, were designed to gain new insight into the rings, the magnetosphere, and the planet itself. The spacecraft will complete twenty-two ring dives leading up to September 15, 2017. On that date, the Cassini spacecraft will plunge into Saturn's atmosphere and incinerate, ending the mission in a spectacular blaze of glory. Figure 1 shows the trajectory of these Grand Finale ring dives in blue, and the final orbit that will draw Cassini into the planet's atmosphere in orange.

Author's email: andrea.m.connell@jpl.nasa.gov

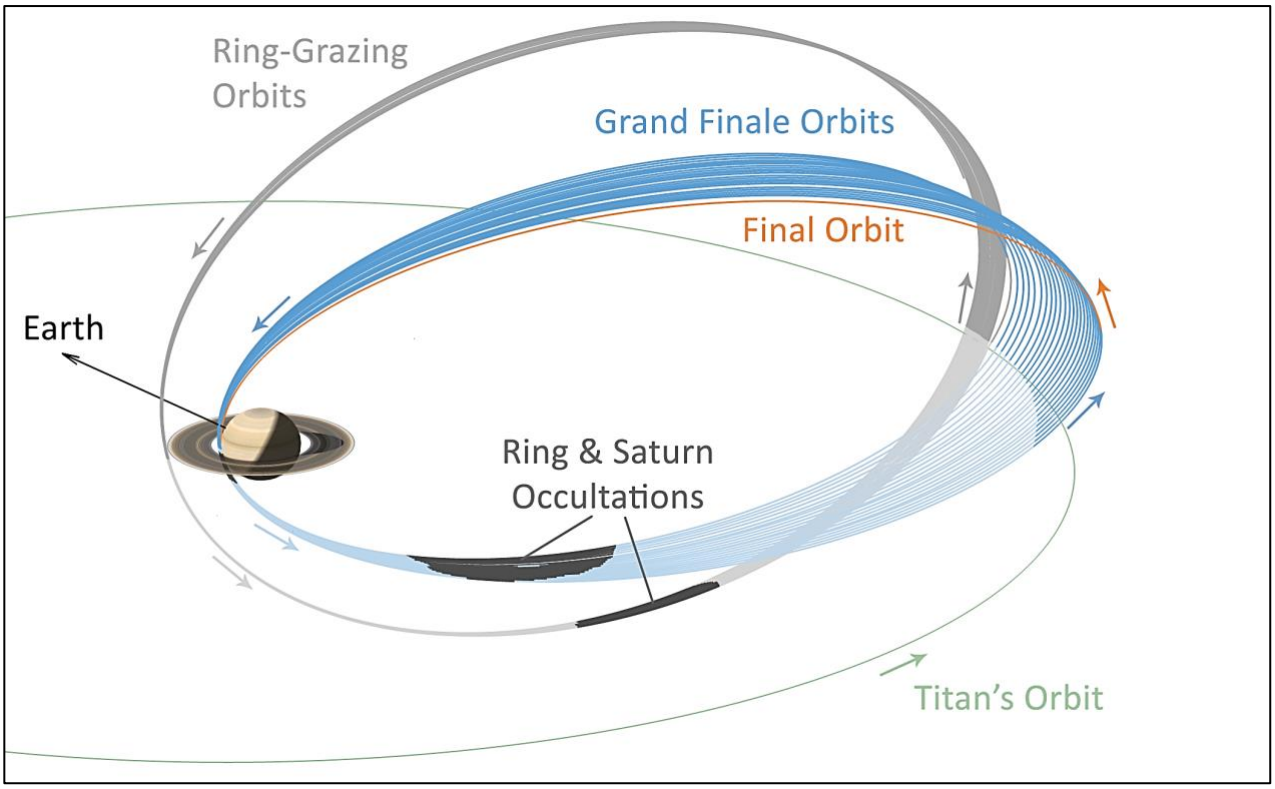

Figure 1. **Cassini's Grand Finale Trajectory**

Nearly twenty years will have passed between Cassini's launch and final plunge into the planet, which means the hardware and much of the software was created in the early or mid-1990s. For example, the main cameras on Cassini are only around one megapixel. The on-board solid-state recorders were the first of their kind to be used in a space mission (replacing the tape recorders used on previous spacecraft), but can only store 500MB of images and other data. Our team's software has similar indications of its old age. It was written with no automated testing in mind, no modular architecture, and a limited change management process. Additionally, budget restrictions encouraged the "inheritance" of technology from previous missions, some of which were already outdated at the time of launch.

The Mission Sequencing Subsystem (MSS) team creates software used to design and validate spacecraft commands before they are sent to Cassini. A "sequence" is a series of commands describing science observations and spacecraft activities, which are usually batched for five to ten weeks at a time. Scientists and sequence planners use the MSS software to design science observations, model spacecraft components, validate that the plans do not violate any mission constraints (such as turning the spacecraft too quickly or operating an instrument in an incorrect mode), and translate the plans into spacecraft commands. After the commands have been created, another software system transmits them to the Cassini spacecraft to be executed.

NASA classifies software in terms of criticality. MSS is Class B software, the designation for non-human space-related software upon which primary mission objectives rely. A problem in one of these software components could affect the ability to generate commands to send to the spacecraft and our ability to return science data from instruments. Class B software requires much more rigorous and formal processes than what many Agile teams may be used to. We support ten main software components, plus an assortment of scripts, utilities, and tests. The components are written in a mix of Perl, C, C++, and domain specific languages. It is interesting to note that at least one component was originally written in Assembly then later ported directly to C, and retains some of the restrictions and maintenance difficulties of the original language.

I joined the mission as a Software Engineer in 2015. Although the mission had been ongoing for many years, big software changes for the upcoming Grand Finale maneuvers were underway. In anticipation of these changes, the team of six began improving their software development processes. They had adopted a well-defined change management process, had recently moved to Scrum, and had begun automating their test suite. However, they

continued to rely heavily on a couple team members with many years of experience on the mission, and the unique challenges of this legacy software system were still having an effect. The sections below will describe those problems, our responses, and the lessons learned in detail.

3. MOVING TO SCRUM

For many years, the MSS team used a waterfall development process. At the beginning of the mission, there was a large team of engineers and managers to support big, complex deliveries. As the mission proceeded through the Prime phase and initial extension, fewer deliveries were needed and the team size shrank. However, the second extension brought a daring new type of trajectory that was outside of the original mission parameters. Software change requests came more rapidly as the team planned to fly the spacecraft closer to the rings and planet than ever before. The team gained a few new employees, but not enough to meet the mission demands while adhering to the old process models.

In 2015 (more than ten years after the Cassini spacecraft had arrived at Saturn) the team decided that a new approach was needed, and opted to switch to Scrum. The primary motivation for the change was to improve communication and transparency among team members. It had become clear that a lot of time was being spent in testing because not enough collaboration was happening during the development phase. Switching to two-week sprints encouraged the team to do smaller, faster iterations even though formal deliveries were still about six months apart. The interaction between developers and testers increased, which made the team more efficient overall. Within a sprint, knowledge was shared more effectively between all members of the team: developers, testers, the product owner, and the change management engineer. We had an opportunity at the daily standup meetings to get awareness into each other's work, identify potential issues that required further discussion, and see where things were blocked. Retrospectives helped to identify which areas still needed improvement and which changes were having positive impacts so that we could continuously improve.

In one case, our daily and bi-weekly meetings helped us see that we were at risk of not being ready for the next delivery on time and mitigate the issue. At that point, our testing process still required a lot of manual effort which was typically done by one or two people. Thanks to our newly improved communication culture, we were sharing the testing progress with the rest of the team every day. When it became clear that the task might not be finished on time, the whole team swarmed on it. Not only was testing completed in time for the delivery, but the problem of manual tests became very clear to the entire team. This situation resulted in increased effort to automate more, and by the time the next delivery was ready the test suite required much less manual work.

However, over time we became less diligent about the sprint planning and backlog grooming events. We did not put enough effort into breaking down story functionality or implementation details, so we missed some key components which we then had to add in at the last minute or in a later software patch. More frequent user involvement from the scientists and sequence planners could have also caught these items sooner. These oversights meant that we had a fair amount of work roll over from one sprint to the next, which ultimately caused the team to take sprint planning less seriously. After about 18 months of using Scrum, our organization offered a ScrumMaster training course. Knowing that we could improve ourselves, about half of the MSS team participated in the class. Learning about the roles, rituals, and artifacts together provided a shared sense of the goals of Scrum, which was much easier to build upon.

Organizational culture also presented some challenges for our Scrum migration. Most people within at JPL work on multiple projects at the same time, which adds a layer of context switching. One value of Scrum is to improve focus by limiting work in progress and thus reducing context switching, but this focus was not entirely possible for us. Every team member split their MSS responsibilities with tasks in other groups at one point or another, which prevented the strong sense of focus and consistent throughput that a successful Scrum team relies on. One way of reducing the impact of this situation was to have specific work hours for each project. Team members who worked half time on Cassini and half time on another mission, for instance, could have decided to work only on Cassini in the mornings and only their other project in the afternoons. Barring any emergencies, this scheduling helped to reduce context switching and improve time estimations.

3.1 Lessons Learned

When migrating to Scrum, our team learned enough to initiate the process change, but not enough to motivate the mind set change required for a fully effective implementation. Team members did not have a shared sense of the goals of Scrum, which led to an informal execution that lost some of the most effective practices over time. In the future, teams migrating to Scrum should go through a training together to make the most of the process.

4. TEST AUTOMATION

Twenty years ago, the MSS software architecture design did not anticipate the eventual need for modular and automated tests. True unit tests would be difficult to implement without a major refactor, which was not feasible given the budget and potential risk it could add to the mission. Instead, we created automated system tests that mimicked the existing manual test procedure: run a specific scenario with defined inputs and compare the output to expected values.

One complex test case involved validating that thermal modeling of instrument sensors was accurate. Some of the instruments onboard the Cassini spacecraft are very sensitive, and if they get too hot they may produce inaccurate science data. After a scientist designed an observation (for example, measuring the infrared spectrum of Saturn's south pole), the software needed to calculate the projected temperature of each of these sensors and present a warning if it was over the thermal limit. The algorithms were complex, taking into account the visible light and thermal radiation from the sun directly, reflecting off of Saturn, and reflecting off of or attenuating through the rings as well as thermal emissions from Saturn and the rings themselves. It also required models of the Cassini spacecraft and the instrument in question to understand when the sensitive detectors were being shaded from other parts of the spacecraft. Runge-Kutta and Monte Carlo integrations were then applied to numerically approximate the effect of the thermal flux that reached the detector. Over time, some of these factors changed. For example, as a result of Cassini's exploration we learned more about Saturn's ring system, which meant that we needed to adjust the ring attenuation factor to know how much sunlight would pass through various parts of the rings. The rules themselves have changed throughout the mission as well, which often required a correlating model change to handle the new operating environment. For example, it was originally considered a violation for the spacecraft to go within the rings of the planet, but that now makes up the essential key to our mission's exciting Grand Finale. The models needed to be updated to optimize science data acquisition in this new environment. Additionally, temperature profiles changed depending on the instrument's operation mode, which was adjusted over time. With so many factors in play, knowing what the correct output should be was very difficult.

Generating the expected outputs required one team member with deep domain knowledge to run the system, inspect the output, and either agree that it looked correct or investigate anomalous results. Once they gave their approval, we considered these values the "baseline" set. This approach required significant time from one specific team member, but some aspects of the work, like executing tests and creating validation scripts, could be shared with other team members if we committed to collaboration and transparency in the process.

Additionally, the Monte Carlo algorithm generated randomness in the output, and the degree of variation in output depended on the precise geometry of a given test case (where the Cassini spacecraft is relative to the Sun and Saturn, for example). For these cases, we created thresholds to understand the maximum acceptable deviation from baseline values. Since the thresholds could change depending on geometry and algorithm implementation, making them easily configurable was key to reducing test maintenance overhead.

After the initial set-up cost and as long as the algorithms did not change, this method was adequate. Tests could be run much faster and with much less manual effort. More tests could be created, and we were better able to characterize the behavior of the software under various conditions since the validation was more precise. We could also get a better understanding of the impact of new changes.

However, when a test failed there was no way to narrow down exactly where the difference originated. The tests were too high level to know which code change caused the deviation from expected values and the complex algorithms made it difficult to determine what the expected output should be. The worst example of this situation was in a delivery where there was a change in the thermal modeling calculations. When we ran the system tests again, most of them failed because the values were outside of their acceptable deviations. This was expected, but we had no way to be sure that the new outputs were correct. We had to repeat the same process of manual verification and call these outputs the new "baseline" values.

Another challenge in test automation was the user interface (UI). Some of our software systems use ActiveState Tcl/Tk for the GUI, which our team found difficult to test in an automated fashion. Even after automating the validation as described above, we continued to use the GUI to enter inputs manually. The details of how the UI interacted with the software engine were largely unclear and undocumented until one team member spent the time to do a deep-dive investigation. Luckily, they discovered that the UI and the logic engine were decoupled enough that we could execute the core functions directly without manually clicking through the screens. This change was definitely worth the time spent discovering it – all tests that aren't specifically testing the UI could now be run without any user intervention. We still performed manual testing on the UI itself.

Thanks to these efforts, a test suite that used to take a week to execute manually could now be completed almost entirely overnight. Since the overhead was so much lower, we could also test more often. In one situation, we were considering making an improvement to the model of the physical attributes of one of the instruments. The code changes themselves were fairly straightforward, but since the mission is risk-adverse and process-heavy, management wanted to know if the impact of the change would be worth doing an entire delivery. Our team was able to use our new test framework to run the existing version of the software against a version with the proposed fix for the mission's 22 Grand Finale orbits. In fact, we ran two configurations of each software version for a total of 88 system executions within just a few days. The validation with built-in threshold checking immediately showed us the extent of the impact. In this case, we found that the change did not make a significant impact on the software output, and ultimately the project decided that the value gained was too small to justify delivering the update.

4.1 Lessons Learned

Of course, the best lesson is to design code to be modular and testable, and to write unit tests before or during the initial development phase. Unit tests that operate on functions with a single responsibility and allow for injections would have let us test each factor individually under controlled circumstances. Knowing that each individual calculation was returning the correct results under the exercised scenarios would provide a lot more confidence in the overall result, and allow new team members to validate results more quickly. However, many legacy systems have already been written and refactoring them to today's standards isn't always realistic. If big or complex changes are needed to legacy systems, automating tests at some levels can still be beneficial to the team. Collaboration and commitment from the entire team was crucial in our experience of finding creative ways to automate verification and validation.

5. TOOLS AND PROCESS AUTOMATION

One obstacle to introducing agility on the MSS team was the process requirements for Class B software within the Cassini mission. Throughout the planning and implementation of a software change, many checkpoints had to be passed. The impact of each new feature or bug fix had to be assessed by multiple groups of stakeholders, including Navigation, Spacecraft Systems, Instrument Subsystems, and Mission Assurance. The Project Manager then approved or denied the change in a control board meeting. The development team had to obtain another level of approval before deploying a new package to the integration environment for final testing. When the package was ready for delivery, we had to create extensive documentation on the changes and completed tests and hold another review meeting before the deployment was scheduled.

In addition to the Class B software requirements, MSS also followed NASA's requirement to meet CMMI-DEV Maturity Level 3 standards. CMMI (Capability Maturity Model Integration) is a program that appraises project teams in terms of 22 process areas, such as Configuration Management, Decision Analysis & Resolution, Measurement & Analysis, and Validation. A CMMI appraisal identifies areas where a team can improve processes to create higher quality software in a more reliable way. All of these process requirements are in place to reduce mission risk and increase the accuracy of cost and schedule estimates, but they do add overhead to the development process. Our team had the most difficulty with the auditing requirements of CMMI – ensuring that every decision, risk, and procedural step was documented appropriately. We were able to reduce the impact of some these process barriers using tools and automation.

A key to our automation success was selecting the right issue tracking and version control systems. Over time, the MSS team has tried multiple tools, looking for ones that match our process needs most efficiently. We have transitioned from CVS to Accurev to GitHub for version control. GitHub provides more opportunity for collaboration between team members, and supports more integration and automation options. We made these transitions slowly over time, moving one code repository to GitHub when we had more work to do on that system, and then following with the other pieces of our software suite later. We also transitioned from an in-house issue tracking system to GitHub Issues to JIRA. At the time, GitHub Issues wasn't robust enough to handle the MSS process, but we had a lot of success with JIRA's powerful features and configurability.

An integration between JIRA and GitHub automated workflow transitions between various work item states, as shown in Figure 2. When a developer committed a code change, they included the JIRA work item number in the commit comments or branch name. The initial commit moved the JIRA work item from "Not Started" to "In Progress" automatically. When the developer created a pull request to merge their code back to the development branch, the JIRA work item automatically moved to the "Resolved" state. After the team performed a code review and merged the pull request, the work item was automatically moved to "Reviewed" and was assigned to the

tester. With these transitions automated, we gathered much more accurate metadata on how long an item was in a particular state. At the end of each sprint, a team member ran a python script to pull this metadata into a custom report format to give project management insight into our progress. We also reviewed these reports during retrospectives to find areas where the team wanted to improve.

Required steps that could not be automatically detected were tracked manually via a JIRA checklist. During the code review, the team was required to verify that the changes met the requirements specified in the JIRA done criteria, the code changes adhered to coding styles and were documented, and that appropriate test cases had been created. Our customized JIRA workspace prevented the work item from moving forward until everything on the list had been checked off.

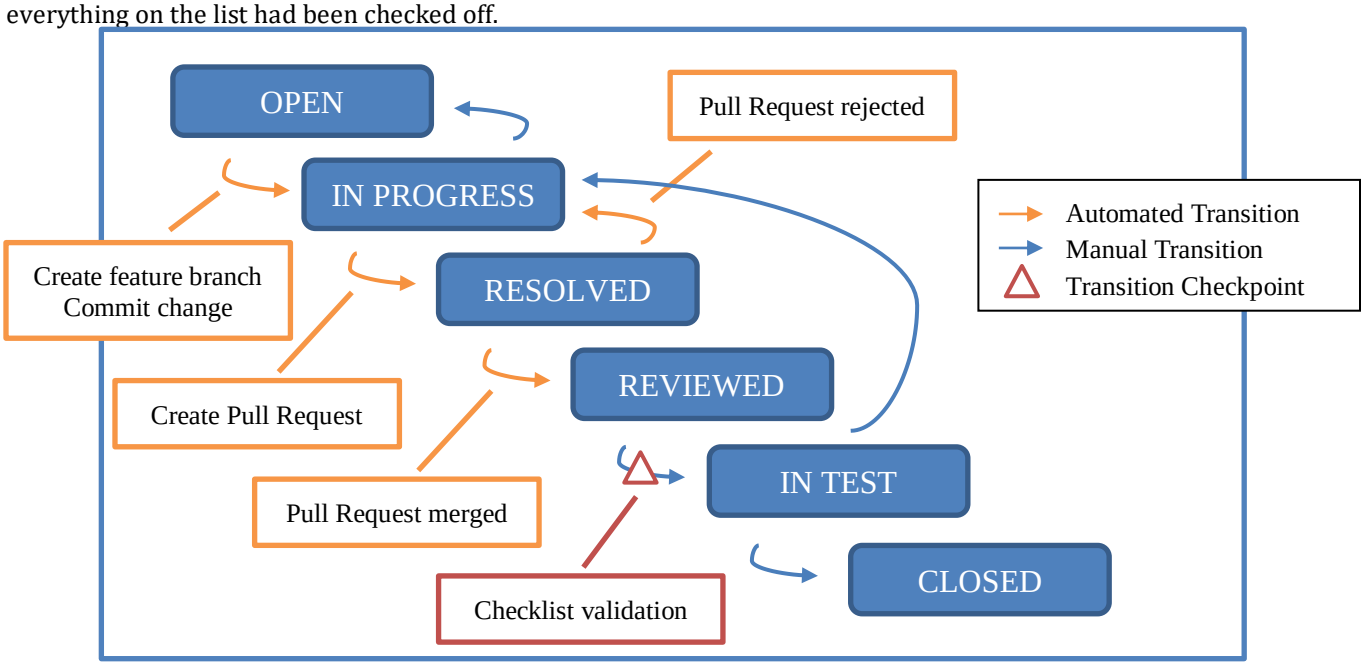

Figure 2. **JIRA Workflow**

Our team also created a home-grown test management framework that could kick off the automated test suite described in the previous section. The test management framework tracked metadata for each test, along with results from the latest run. This framework then automatically generated test reports as required by the project for each delivery. The test report, which contained detailed information about each of the tests that were executed, was over one hundred pages long for a typical delivery and would take about a week to recreate from scratch without this automation.

After implementing these changes, the MSS team underwent an evaluation for CMMI Level 3 Rating and the appraisers found no weaknesses. In fact, our use of JIRA and GitHub to implement processes and peer reviews was called out as strengths by the CMMI appraisal team.

5.1 Lessons Learned

There is a seemingly endless number of ways that development teams can combine various tools to automate their processes today. Although we needed to switch between them multiple times over the years, we found a combination that worked to improve transparency and saved time on required process steps. We used a mix of commercially available and in-house products to get to the right solution. The result of our CMMI appraisal was an indication that we could automate many required processes without affecting the quality and reliability of our deliveries.

6. CONCLUSION

On September 15, 2017, Cassini will plunge into Saturn, sending as much data back to us as possible before the high pressures of the atmosphere will cause it to lose communications with Earth and then incinerate. The intrepid spacecraft has made incredible discoveries during its 13 years within the Saturn system.

We've discovered lakes of methane and ethane on the surface of Saturn's largest moon Titan. Now in addition to being the only known moon with its own thick atmosphere, Titan is also the only place we know of with an active liquid cycle similar to Earth. With its clouds, rain, rivers, and sand dunes, scientists believe that Titan may look now like Earth did billions of years ago, before life began.

We've discovered that the tiny moon Enceladus has a salty ocean underneath its icy crust, and that the gravity of Saturn and Titan push and pull on Enceladus enough to heat the interior and send plumes of water vapor shooting out of the south pole into space. Cassini flew directly through these plumes, sampling the expelled particles and discovering key ingredients for life in a place nobody thought to look before.

We've studied the composition and interaction of particles within the expansive ring system, and watched new ring features form and collapse. We've observed Saturn for half of its orbit around the Sun – seeing the seasons come and go along with giant storms and lightning. We've imaged auroras at the poles of the planet caused by interactions with its giant magnetosphere.

During this unprecedented period of exploration, hundreds of people have worked tirelessly behind the scenes to ensure that the mission was a continued success. The teams have learned countless lessons along the way – many related to the long duration of the mission. In the preceding sections I have covered three important lessons from the MSS team:

- Moving a legacy team to Scrum can be done successfully, but to be truly effective the entire team should go through a training session together.
- Even with strict organizational process requirements, a team can make improvements by using tools and automation.
- The biggest challenge remains to be automating tests for software that was not architected for it, but we did make useful improvements in this area as well.

Managing and collaborating with an international team of scientists, engineers, and support personnel to operate a piece of machinery 1.3 billion kilometers away while technology continues to change rapidly here on Earth will always present challenges, but is, of course, worth the effort. Many of the lessons learned by the MSS team can be applied to other software development teams, regardless of their place in the universe.

7. ACKNOWLEDGEMENTS

First and foremost, I am grateful for everybody I have worked with on the Cassini MSS team for starting this transformation before I arrived at JPL, for welcoming me and my suggestions when I joined the group, and for continuing to be a great team even as we encountered issues. Thank you to Diane Conner, David Tong, Barbara Streiffert, Usha Guduri, Jeff Boyer, Warren Kaye, Shahen Petrosian, Sandra Bottenfield, Sheila Chatterjee, George Vine, and Jonathan Castello. Extra appreciation goes out to Diane, David, Barbara, and Sandra for reviewing this paper and ensuring its accuracy. Thanks to the Cassini project management for giving us the opportunity to improve our processes while keeping the spacecraft safe and productive. Finally, thank you to my paper shepherd, David Kane, for his valuable insight and analysis to shape this paper into its final form. This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.