

Testing and distribution of the RPS2 proposal submission system

Robert E. Douglas, Jr.

Space Telescope Science Institute, 3700 San Martin Drive, Baltimore, MD 21218

ABSTRACT

In 1995, the Space Telescope Science Institute (STScI) introduced RPS2 (Remote Proposal Submission 2). RPS2 is used by Hubble Space Telescope (HST) proposers to prepare their detailed observation descriptions. It is a client/server system implemented using Tcl/Tk. The client can transparently access servers on the user's machine, at STScI, or on any other machine on the Internet. The servers combine syntax checking, feasibility analysis and orbit packing, and constraint and schedulability analysis of user-specified proposals as they will be performed aboard HST.

Prior to the release of RPS2, observers used a system which provided only syntax checking. RPS2 now provides the observers with some of the more complicated pieces of software that had been used by STScI staff to prepare observations since 1990. The RPS2 system consists of four independent subsystems, controlled by the client/server mechanism.

A problem with a system of this size and complexity is that the software components, which continue to grow and change with HST itself, must continually be tested and distributed to those who need it. In the past, it had been acceptable to release the RPS2 software only once per observing cycle, but it became apparent before the 1997 HST Servicing Mission that multiple releases of RPS2 were going to be required to support the new instruments. This paper discusses how RPS2 and its component systems are maintained, updated, tested, and distributed.

Keywords: proposal submission, proposal preparation, testing, software distribution, Hubble Space Telescope (HST)

1. HISTORY

In order to understand the role of RPS2 in HST observing, it is first important to understand the process by which observations are performed. When an astronomer wants to observe with HST, he must first submit a Phase I Proposal at the beginning of an observing cycle during the Call for Proposals. All proposals are reviewed and ranked by an independent panel, the Time Allocation Committee (TAC). Once an astronomer's proposal is accepted by the TAC, he then becomes an investigator (referred to as a PI, for Principal Investigator). The PI then develops a much more detailed observing program (Phase II Proposal). He has to list the exposures to be performed, specifying for each exposure the target to observe, instrument to use, as well as the filter, exposure time, instrument parameters, and scheduling requirements. STScI provides the software necessary for preparing the proposal and submitting it to the Institute for further processing. Prior to 1995, this system was known as the Remote Proposal Submission System (RPSS). In 1995, just in time for HST observing for cycle 5, we introduced Remote Proposal Submission 2 (RPS2) to replace RPSS.

1.1. Remote Proposal Submission System (RPSS)

Prior to RPS2, PIs would develop their proposals in a text file. Then they would use RPSS to check the syntax of their file and would then electronically mail the file to STScI. RPSS was a command-line program written in C. PIs working under SunOS or VMS would download binary versions of the software to their site, where it was then able to be run. RPSS would provide a list of syntax errors in a few separate files, and PIs would make the necessary changes and rerun the software until the proposal was free of errors. They would then electronically mail their proposals to STScI. The RPSS process flow is shown in Figure 1 below.

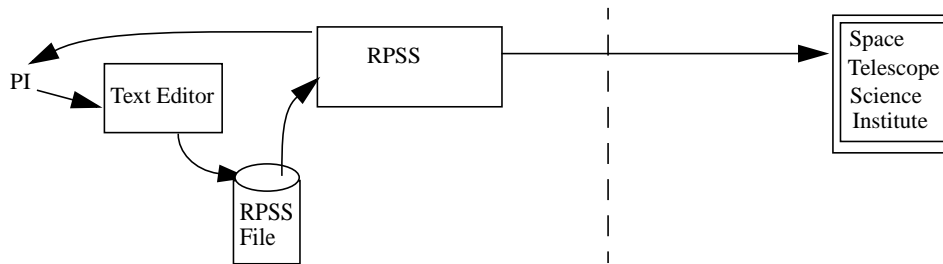


Figure 1: RPSS Process Flow

RPSS worked well for many years, and was used to prepare detailed proposals for 5 cycles of HST observing (cycle 0 - cycle 4). It solved the problem of entering all of the proposal information for hundreds of proposals a year into the HST planning and scheduling systems. However, once a proposal comes into the Institute, there is a significant amount of preparation needed in order to actually perform the observation. Before HST was deployed, we developed software at STScI to plan and schedule the observations. Transformation (Trans) plans the exposures of a proposal, including the addition of overheads and orbit packing, and performs feasibility checks on the observations.¹ Spike determines when during the year a particular proposal can schedule, and is used to generate a long-range plan of observations from a set of proposals.² Most problems with a proposal were uncovered while preparing a proposal using these tools. These problems were semantic rather than syntactic. A proposal which is syntactically correct might still not be schedulable during the year, or it may not be executable onboard HST.

When the staff at STScI discovered problems of this nature, they then had to contact the PI and relay enough information to help him make trade-offs in his proposal in order to correct critical problems. Only the PI is qualified to make these kinds of trade-offs, because the decisions he makes will affect the quality and kind of science that is performed. This process also interfered with the smooth operation of the ground system for scheduling observations.

Here we saw an opportunity for improvement: Could we use our submission system to reduce the problems found after submission? We set out to provide a system that was easy to use and which improved the quality of the proposals when they were submitted. By improving the quality of the submissions, we would be able to spend more time helping to improve the scientific aspects of HST observing without having to focus on operational problems.

1.2. Remote Proposal Submission 2 (RPS2)

At the beginning of 1994, we set out to replace RPSS with a new system, which became known as RPS2. The main idea behind this new system was that we wanted to provide better insight and feedback into the proposal preparation process by incorporating Trans and Spike into the system. This would allow PIs to use this information to correct problems in their proposals prior to submission. Incorporating Trans and Spike increased the scope and complexity of the proposal submission software by several orders of magnitude. Creating a tool which incorporated them for use by PIs presented us with several challenges. We wanted to make the system easy to use, even easier than RPSS, while providing a greater functionality than RPSS. We also had to find a way to integrate software which was not designed for use in a single tool, and whose output required some amount of expertise to understand. Also, Trans and Spike are both very powerful tools, which perform very complicated task. We could not be sure that all PIs would have powerful enough processors to run both tools at their site. Finally, we had to support a variety of platforms other than those we used at STScI, so that all PIs would be able to use RPS2.

In order to make it easy to use we provided a GUI to control the system. We also replaced the syntax checking that was performed in RPSS by providing a graphical Proposal Editor (PED) for inputting proposal information.³ In order to make the output of Trans and Spike more useful to the PI, we built a Description Generator (DG), which provides a graphical display of the results of processing a proposal, including a textual listing, orbital display, schedulability display, and a listing of diagnostics generated by the systems.⁴

We developed a Control Program (CP) to provide access to all the software components of RPS2. The CP is a client/server system which can support servers running on any machine accessible through the Internet. The client provides the PI access to the GUI which controls the amount and type of processing to be performed on his proposal. The servers provide access to each of the software components. The CP allows the software components of RPS2 to be running on different machines. It is possible to add new components to the CP as desired. For example, we have thought of adding access to the Guide Star Selection System (GSSS) to RPS2. The RPS2 CP has very little HST or component specific information and could easily be adapted to incorporate other software.⁵

The CP was also designed to provide us with the most cost-effective way of handling the issue of providing access to Trans and Spike across different platforms. Both systems are implemented in LISP, which provides them with the functionality they need, but porting LISP to other platforms does have a cost. PED, the DG, and the CP were all implemented using Tcl/Tk, a scripting language which runs on a wide variety of platforms. Combined with a set of servers able to run Trans and Spike at STScI, we could easily provide access to RPS2 for most of the platforms used by HST observers.

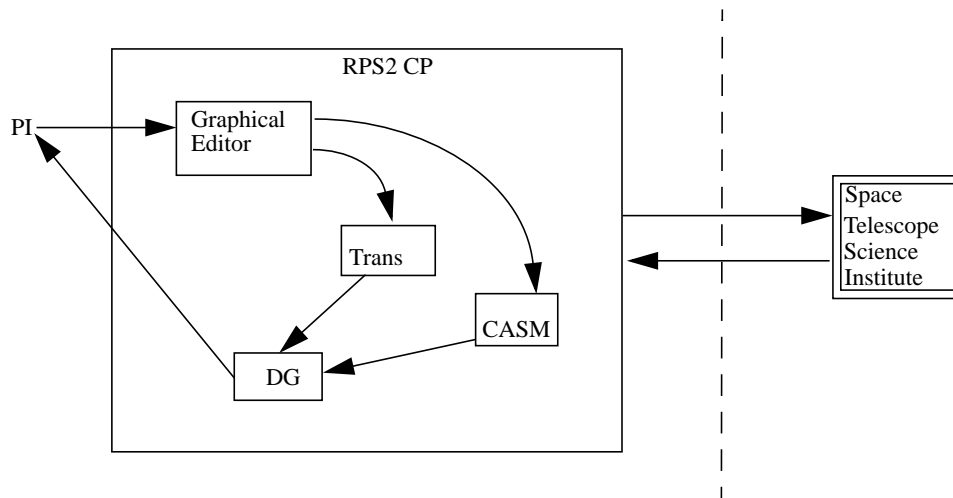


Figure 2: RPS2 Process Flow

RPS2 is a very complex software system, capable of providing extensive feedback about a proposal, as shown in the RPS2 process flow in the Figure 2 above. A system of this complexity presents us with the equally complex task of testing and distributing the system. We have a system which can handle billions of possible inputs and outputs, involving several sub-systems: How do we test them all? We want to continually update the software throughout an observing cycle to take advantage of new observatory capabilities and support new instruments, as necessary to keep HST on the leading edge of astronomical discovery. We have to get the software right or we jeopardize the smooth operation of the HST. And we have to do this under a very tight schedule. Given all of that, how do we routinely provide quality software to hundreds of PIs? One way we could this is to hire hundreds of testers to validate each release by checking each of the billions of combinations that are possible. This is not a very cost effective solution, so we developed a different one.

2. DISTRIBUTION PROCEDURES AND TESTING TECHNIQUES

Given the size and complexity of RPS2 and the goal of releasing the whole system periodically throughout an observing cycle, as new observatory capabilities become available, it was important to develop a testing procedure which would give us the most confidence in software quality while requiring few enough resources to make it workable and a delivery procedure which would deliver it into the hands of hundreds of users very quickly at a minimum of cost. Delivery of RPS2 is kept simple by keeping the complexity away from the user. Testing is performed by using the software in-house, beta testing, and making use of regression and usability tests as appropriate. We also developed some measures of the amounts of testing necessary to evaluate the quality of the software following a change.

2.1. Delivery of RPS2

We have solved the problem of delivering RPS2 to our users in a cost effective way by relying on Tcl/Tk, our client/server architecture, and a simplified installation procedure. Our Tcl/Tk license allows us to build executable images of Tcl/Tk and freely distribute them along with our software, so that it can run on many platforms. This means that we do not have to worry whether or not the user's site supports Tcl/Tk already.

For those systems which are not written in Tcl/Tk (Trans and Spike), we can provide executable images under Solaris, but the client/server mechanism allows users who are not running Solaris to still use RPS2. We provide servers at STScI for all of the tools in RPS2. We could even use this mechanism to add other tools to RPS2 without making the delivery more difficult, systems such as the Guide Star Selection System (GSSS) or Moving Object Support System (MOSS).

When it is ready for operational release, a complete version of RPS2 is created and built, and then archived using the "tar" utility, data compressed, and made available for FTP on the Web. The installation procedure is very simple. It requires the user only to know where he would like to install RPS2 and what platform he is using. The user need only click on the appropriate version from a Web browser, use the "zcat" and "tar" utilities to uncompress and restore the software from the archive, and then type "make install".

The installation procedure will determine all the features it needs to know about the users environment and build the software to provide RPS2 complete and ready to use. It is able to install itself on top of previous versions of RPS2. In Cycle 7 alone, which began in January, 1997, we have delivered 7 operational releases of RPS2 in support of new scientific capabilities. Our delivery and installation procedure has reduced the development overhead of releasing the software this often to a less than an hour per release. Additionally, we have found that it is simple enough that most PIs tend to install the latest version of RPS2 whenever they want to change their proposals, even though they may not be required to do so (e.g. if they are not using any new capabilities). As developers, we prefer that they keep their versions of RPS2 updated, as it greatly simplifies the process of diagnosing problems, should the user develop a problem. So we have begun discussing the idea of allowing changed files to be downloaded automatically when running RPS2. This would mean that no one would have to install RPS2 more than once, simplifying life even more.

2.2. Testing with in-house use

As was stated above, Trans and Spike are the same tools that have been used for many years to prepare observations before they can be flown aboard HST. Rather than requiring ourselves to maintain multiple different software systems which perform the same task, we continue to use all of the subsystems of RPS2 within STScI when preparing observations, including PED and the DG. Each of the subsystems is maintained throughout the year, and new versions of these systems are installed in-house whenever a significant change is completed. It is critical that the in-house versions be kept up-to-date with changes in HST. As a result, the RPS2 released versions of the subsystems will lag behind the in-house versions. One of the benefits of this is that we can have some amount of confidence in the software that we intend to release with RPS2, as we will have been using that version for mission operations for some time already. Defects found with a new implementation of some capability are more likely to be caught with the in-house versions and fixed before the capability is ever released in an RPS2 release.

Trans in particular is especially good at turning around quality enhancements in a short time-period. It uses a model of development which supports independent testing of an individual change before the change is made part of the released software.⁶ This is an important capability, since Trans actually models activities aboard HST and as a result is the most susceptible to changes that can occur aboard the observatory.

2.3. Beta testing

In order to verify that all of the components of RPS2 continue to work together properly, we perform an end-to-end, integrated system test of the system. The purpose of the Beta test phase is not to catch bugs, but to evaluate the quality of the software prior to release. If problems are found, they are analyzed for their impact on the RPS2 user community. The cost of fixing them is established, but in general they are only documented and planned to be fixed in a future release. If a particularly insidious defect is found, then we can make an informed decision about whether or not to fix it, and how long to delay the release (see section 2.6).

Two weeks prior to releasing a new operational version of RPS2, we build a Beta test release. We involve several of our in-house experts in Beta testing RPS2. Each of these experts looks at RPS2 from a particular aspect of HST observing. They are assigned specific HST instruments, HST parameters, or RPS2 functionality to test. For example, a STIS Instrument Scientist would test new STIS modes and optional parameters, while a Moving Target expert would look at the PED Moving Target Editor. Each Beta tester concentrates on his area of expertise. This gives us a wide range of testing coverage, while gathering feedback from those people at STScI who know best about how the software should work.

2.4. Automated regression tests

Automated regression tests have a distinct advantage over human testers: they perform the same tests over and over, reliably, not just once, but forever. We have developed a suite of automated regression tests for Trans and Spike to protect against undesirable side-effects of the implementation of a new capability. Additionally, these tests continue to grow over time, providing greater and greater coverage of the systems. By running the regression tests as close to the time that a change to the system is implemented, you are able catch potential problems very early in the software lifecycle. This promotes a higher level of confidence in the software as it nears time for operational delivery.

Trans regression tests are managed by the Looker, an expert system able to look at the internal workings of Trans and intelligently compare them against the expected results.⁷ The Looker plays a key role in allowing Trans to be enhanced quickly in response to changes in HST or mission operations (see section 2.2).

2.5. Usability tests

We test our graphical tools by providing them to experts who use them and provide feedback about how well the tool works at its task. There is no way to develop a set of requirements for a graphical tool without discussing them with a user, and no way to guarantee that the tool is usable even if it meets those requirements except by having someone use it. These usability tests provide us with a tremendous amount of feedback regarding the best way to present information to the RPS2 user.

2.6. Applying the right testing

The above testing techniques provide us the assurance that we are able to apply the right testing for RPS2. By the time RPS2 is ready to be released we have had new capabilities tested by in-house usage and Beta testing, and gained some assurance that existing capabilities are still intact through regression tests. Our usability tests ensure that the graphical tools are able to provide data in the best way possible and provide the necessary control over proposal preparation to make the whole system easy to use.

2.7. Release phases - controlling the scope

When problems are found during testing, we would like to fix them in order to deliver a higher quality product to the user. However, last minute changes can often cause other problems which resulted in either decreased quality of the software or excessive delays in the release schedule. We strive to provide software which we are confident is of high enough quality to perform the task that it must do.

Consider the following graph, which shows how development of an RPS2 release is broken down:

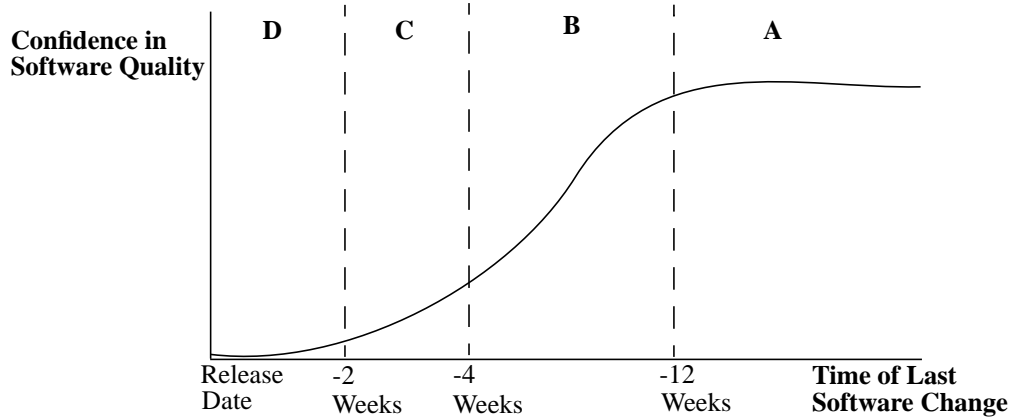


Figure 3: RPS2 Phase Release Schedule

The vertical scale measures our confidence in the quality of the software at a particular time in the development process. The horizontal scale is a measure of the time that a change is made before the software is to be released. Early in the software development process (area A on the graph), we can be assured that we have enough time to test any software change. As we progress closer to the release date, however, we eventually find that there is a point where our confidence in software quality after a change drops (areas B and C). This is because there may not be enough time to fully test the change. Finally, the software reaches a point just before release when no changes can confidently be included in the system (area D). The key to allowing the software to continue to change between A and D is to manage the types of changes that can be made during that part of the development process. For example, changing the wording of an error message has little impact on the quality of the software, but changing the algorithm used for a particularly complex scheduling problem might greatly affect the quality of the software. One can decrease the slope of the graph above by limiting the types of changes that can be made during this period.

We developed a procedure for defining the transition points during the software development lifecycle. We decided to manage the area between A and D by incorporating a phase transition, where we allow more extensive change during phase B than in phase C. The letters above each region of the graph identify four phases of software development that we use, and the times along the horizontal scale are the times that we use for transitioning between phases. Another software product that wanted to use this method would have to identify phases that have a qualitatively different meaning to their development process, and the proper amount of time before release to make the phase transitions.

The current phase of development determines what types of software enhancements can be made at a particular time. This helps requirements writers to know when they need to provide requirements, and helps developers to feel that they have the time needed to make the change without adversely affecting the quality of the software. Knowing when we wish to release a version of RPS2 allows us to determine the current phase, and then determine whether or not a particular change should be implemented for that release. The following table indicates in which changes can be included in each software development phase for RPS:

Table 1: Changes allowed during each phase

Phase	Types of change that are allowed
A	Any change
B	Any critical changes, medium effort important changes, or any low effort change
C	Only critical changes and low effort important changes
D	No changes without delaying the release

The amount of effort needed is determined by the developers. They are the only ones who know how hard it will be to implement a requested change. We consider that anything requiring a day of work is low effort, work requiring less than 1 week is medium effort, and anything else is high effort. Priorities are set by an in-house group of RPS2 users, who decide if the change that is being requested is critical (meaning they would rather not release RPS2 without this change), important, or just that it would be nice improvement to the system.

The release phasing allows us to measure the progress of the software toward release. We know that if we want to make a change to the software which is not allowed during that phase, we should seriously consider delaying the release rather than trying to rush the changes into the delivery of RPS2. While we rely on this system for making decisions about what changes to include in a release of RPS2, we recognize that it is a tool for making better decisions, and not a straightjacket which interferes with the timely implementation of software changes that are needed.

Phase D corresponds to our Beta release phase. It is expected that no more changes will be made to the software. This gives us a stable test version of RPS2 for two weeks prior to release. If we find that a critical change has to be made, then the release has to be delayed and the schedule set back to an appropriate phase, i.e. if it requires low effort, then we return to phase C, but if it requires high effort, then we return to phase B. This allows us more time to test complex changes and to evaluate the quality of the latest software. Defects which are noted at any point during testing, but which are not fixed, are documented, along with their workarounds, and are scheduled to be fixed for the next release of the software. This is considered an acceptable trade-off between cost and quality. We would prefer to have software free of defects, but this is not cost-effective, so we hope that if we must release with defects, we prefer to make sure that they have low impact and are well understood.

3. CONCLUSION

We are always trying to find ways to improve the quality of our software and our ability to deliver that software to those who need it quickly, without compromising quality. We have added a code review process for some of the more difficult development tasks to help ensure quality. We continue to augment our regression test suites to improve our confidence that software enhancements do not have undesired side-effects. Providing high quality software allows astronomers using HST to focus on attaining the greatest scientific achievements possible. This is the ideal that led us to develop RPS2 in the first place, and which will help us continue to improve the testing and distribution of RPS2.

4. ACKNOWLEDGMENTS

I would like to thank all of the developers and testing specialists who have worked so hard to make RPS2 what it is today. You have done much and should be proud.

5. REFERENCES

1. A. Gerb, "Transformation reborn: A new generation expert system", *Proc. Goddard Conf. on Space Applications of Artificial Intelligence*, 1991.
2. M. Johnston, G. Miller, "Spike: Intelligent Scheduling of Hubble Space Telescope Observations" in *Intelligent Scheduling*, M. Zweben and M. Fox (eds.), San Francisco: Morgan-Kaufmann, pp. 391-422, 1994.
3. D. Asson, A. Bose, and T. Krueger, "A Tcl/Tk-based, intelligent graphical editor for preparing HST programs" in *Astronomical Data Analysis Software and Systems V*, G. H. Jacoby and J. Barnes (eds.), ASP Conference Series, vol. 101, pp. 447-450, 1996.
4. A. Bose, G. Miller, and A. Gerb, "An Interactive Tool to Aid in Proposal Preparation for the Hubble Space Telescope", *SPIE Proceedings*, vol. 2479, P. Wallace (ed.), pp. 434-444, 1995.
5. R.E. Douglas, Jr. and R.E. Jackson, "The RPS2 Generic Distributed Computing Framework" in *Astronomical Data Analysis Software and Systems V*, G. H. Jacoby and J. Barnes (eds.), ASP Conference Series, vol. 101, pp. 455-458, 1996.

6. A. Gerb, G. Curtis, R. Douglas, N. Nigro, V. Berman, and L. Swaminathan, "Delivering Quality Software in Twenty-Four Hours", *Proc. 10th International Software Quality Week*, vol. II, section 6M1, 1997.
7. A. Gerb, "The LOOKER: Using an expert system to test an expert system", *The World Congress on Expert Systems*, 1991.