Technology Infusion of CodeSonar into the Space Network Ground Segment (RII07) Final Report

Markland Benson / Code 583 – Goddard Space Flight Center

Abstract—**The NASA Software Assurance Research Program (in part) performs studies as to the feasibility of technologies for improving the safety, quality, reliability, cost, and performance of NASA software. This study considers the application of commercial automated source code analysis tools to mission critical ground software that is in the operations and sustainment portion of the product lifecycle.**

I. INTRODUCTION

This report documents the results of a study funded by the National Aeronautics and Space Administration (NASA) Software Assurance Research Program as part of an effort to determine the feasibility and benefits of incorporating modern software assurance tools in NASA programs. This study focused on automated source code analysis tool use within the Tracking and Data Relay Satellite System (TDRSS) ground segment. The study was designed to target a single, mature tool and examine the costs and benefits of the tool against the current operating practices within the TDRSS ground segment, while drawing more general conclusions whenever possible. The tool selected was CodeSonar, which is produced by GrammaTech. This tool was selected to study because it is advertised to detect a number of problems experienced by the TDRSS ground segment maintainers and because the tool has beta support for the Ada language in addition mature support for C and C++. These three languages combined account for over 90% of the source code maintained by the TDRSS ground segment. CodeSonar will be referred to in general terms in this report except where a specific comparison of this vendor's technology is needed, in order to maintain the theme of using this specific tool to evaluate the technology in general.

II. PROBLEM CONTEXT

The NASA TDRSS consists of a fleet of nine geosynchronous satellites and three ground stations that provide continuous "bent pipe" communications services at and below low earth orbit from customer end items (frequently satellites) to mission control centers. Human spaceflight, space science, and earth science NASA missions depend on TDRSS to provide launch support, and on-orbit communications and tracking services. A sample of the customer base includes the Space Shuttle, International Space Station, Hubble Space Telescope, Fermi Gamma-ray Space Telescope, Time History of Events and Macroscale Interactions (THEMIS), Aqua, and Aura as well as the National Science Foundation (NSF) South Pole TDRSS Relay (SPTR).

TDRSS customers require high availability of communications resources and proficiency to maintain health and safety of assets and in delivery of science and engineering data. The minimum availability requirement for TDRSS is 97.00% and the minimum proficiency requirement for delivery of scheduled services by TDRSS is 99.90% while the standard of excellence for these metrics is 98.00% and 99.97% respectively. One problem with such high availability and proficiency requirements is that the TDRSS ground segment is controlled by over eight million lines of software. Casting aside the space segment, operator error, and hardware concerns, just creating software to meet these expectations is a challenge since it cannot be exhaustively tested *[1]*.

Measurements from the ground segment discrepancy report system collected from 2003 to 2008 show that on average 28% of lost service time attributable to the ground segment per year is caused by software. This number is equivalent to 7% of the overall loss recorded for TDRSS since external entities, such as the mission operation centers requesting services, account for the largest portion of data losses and the external losses are not counted against ground segment proficiency. *Figure 1 – Loss by Category* depicts the proportions of loss attributed to external entities versus loss within the control of TDRSS with a further breakdown of the proportions of loss attributable to hardware, software, and operations within TDRSS.

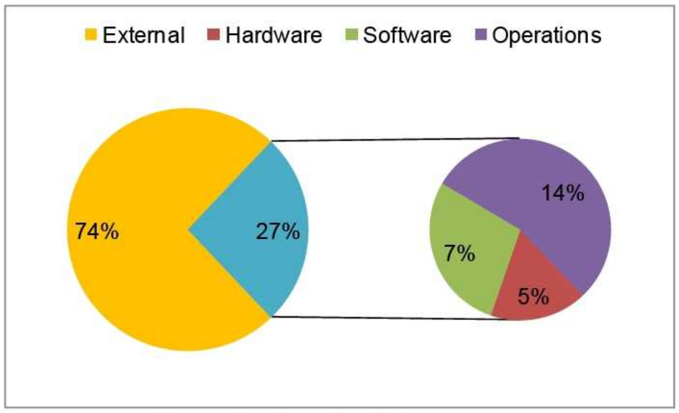

Figure 1 – Loss by Category

Figure 2 – Software Loss Trend (Percentage) shows hours of loss attributable to software as a percentage of hours of loss within the ground segment. The trend line is derived via the least squares method. The measure of percent loss can be somewhat deceiving in that it does not necessarily indicate that software is producing fewer errors. In TDRSS, non-software sources of error are growing as aging hardware reaches end of life and becomes less reliable, thus the overall percentage of errors associated with software is reduced.

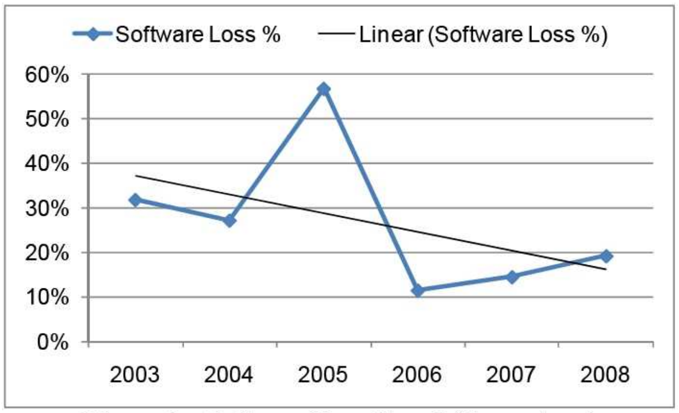

Figure 2 – Software Loss Trend (Percentage)

Figure 3 – Detailed Loss Trends (Hours) distinguishes hours of loss attributable to software and non-software ground segment sources. Linear regression trends for the ground segment as a whole, for non-software sources, and for software sources are also provided. Anecdotal evidence suggests that improvements in the software trend may be a result of formal manual software product inspections introduced in 2006 after a large loss attributable to software as well as improved software configuration management tools and methods introduced in 2007. Note the 2003-2006 upward trend line parallel to non-software sources.

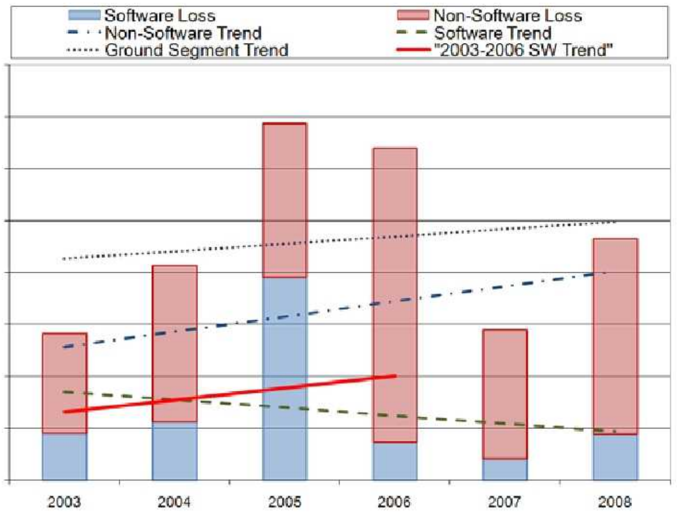

Figure 3 – Detailed Loss Trends (Hours)

Despite the overall downward trend in software's contribution to data loss, some CSCIs are known to be more error prone than others. Further, a large staff of software engineers is in place to troubleshoot and enhance the ground systems software. Process improvement efforts drive sustainers to looks for efficiencies to reduce software maintenance effort and to drive software issues to near zero. It is reasonable to expect that the percentage of data loss compared to the percentage of time that service is provided for a given component would be roughly equal when defects are minimized or at least evenly distributed. This expectation is not fulfilled in reality. For example, CSCI B, which has been in service for three years, accounted for 21% of the software induced lost time in 2008 while it accounted for 15% of the service time. CSCI A and CSCI B are relatively small compared to the remainder of the ground segment software, accounting for a combined total of 3% of the source lines of code but being responsible for 23-26% of losses in proficiency in 2008. The uncertainty on this number is because of recent discrepancies under investigation, whose root cause is yet to be determined. Anecdotal evidence has been given that suggests that CSCI A, deployed in late 2007, had higher than anticipated failure rates and required rework that delayed operational readiness of the system. Because of the anecdotal and quantitative evidence of higher latent defect rates of CSCI A and CSCI B as compared to the remainder of the TDRSS ground segment software, these CSCIs were chosen as targets of the automated source code analysis technology infusion.

III. FINDINGS

A. *Findings Overview*

Findings documented in this report are the result of using the automated source code analysis tool to analyze CSCI A and CSCI B in their entirety. Firmware components associated with these CSCIs will be a source of future scrutiny but were not analyzed in this study. *Figure 4 – Original Finding Categories* shows the distribution of findings produced by the tool as determined by maintainers of the software. The y-axis indicates the number of findings produced by the tool and the x-axis categorizes the findings. The four categories are defined as follows. True Positive indicates that the maintainer was able to identify the finding as a defect in the code that needs corrected. This identification occurs using only the automated source code analysis tool interface and output. Requires Research indicates that the maintainer was not able to determine whether the finding is a true defect or not without further analysis. False Positive indicates that the maintainer was able to identify the finding as a non-defect without further review. Finally, Vendor Software indicates that a potential defect exists in off-the-shelf software rather than software maintained in-house.

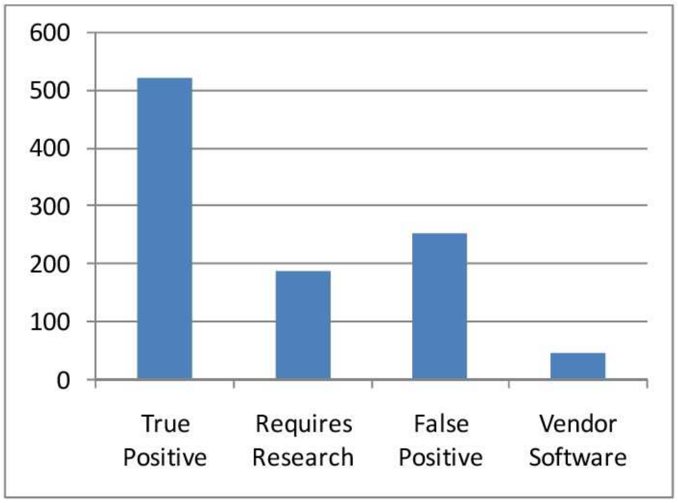

Figure 4 – Original Finding Categories

Approximately 19% of the findings were classified as Requires Research, which could be a cause for concern regarding the amount of effort required to further assess the findings. Fortunately, of the 189 findings of this type, the vast majority are likely non-defects. The two largest categories of findings marked as Requires Research are Unreachable Code (93 instances) and Redundant Condition (32 instances). While these could be valid findings, experience with this tool indicates that these types of findings (as well as some others) are often a result of a single finding that the tool determines would cause a processor reset. For example, a Null Pointer Dereference causes an exception on a number of processors. Logic in the tool states that given an exception, certain portions of code will not be executed, resulting in unreachable code, redundant conditions, uninitialized variables, invalid file descriptors, useless assignments or memory leaks (not an exhaustive list).

Given this history, having developers to eliminate definite true positives known to cause processor resets prior to evaluating findings categorized as Requires Research will likely result in a marked reduction in the number of defects reported by the tool even though many of these defects do not appear to be directly related to the defect corrected. This study cannot await such modifications to the software to determine if this holds true in every case. So, rather than count all findings marked as Requires Research as False Positive and possibly understate the number of defects, the remainder of discussion in this paper will consider all items categorized as Requires Research to be true positives except those marked as Unreachable Code and Redundant Condition. This approach allows the study to proceed without performing extensive review of the 189 findings categorized as Requires Research with the reasonable assumption that the 125 findings of types Unreachable Code and Redundant Condition are False Positive while the remaining 64 findings categorized as Requirements Research are True Positive. The simplified categorization of findings is shown in *Figure 5 – Simplified Findings Categories*, including incorporation of Vendor Software as False Positive. Divisions in bars distinguish the origin of the findings, whereas the x-axis shows the final categorization of findings.

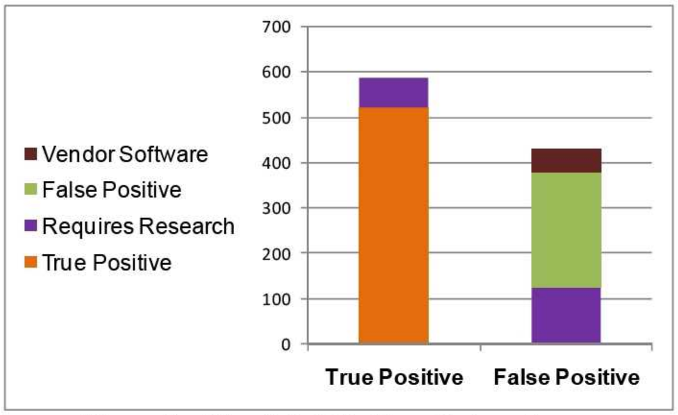

Figure 5 – Simplified Findings Categories

B. *Findings Content*

According to the simplified findings categories, a total of 585 findings were produced by the tool that were determined to be True Positive. This section discusses the True Positive findings in more detail. Any reference to findings in this section as well as the next section entitled *FINDINGS IN CONTEXT* is constrained to those of the True Positive persuasion. In further discussion, the CSCI B will be discussed as a whole as it was analyzed in this manner but CSCI A will be broken down into CSC B, CSC C, and CSC D.

One avenue to question is how findings compare across different components analyzed. In order to compare findings across components, the number of findings was normalized by dividing the number of findings in each category by thousands of source lines of code (KSLOC) in the applicable component.

Findings were not found to be highly consistent among different components analyzed though some similarities exist. Consistency among components was determined by concentrations of findings of a given type among components. For this study, a finding is said to be concentrated if it has more than 1 defect per KSLOC and somewhat concentrated if more than 0.5 defects per KSLOC. CSCI B had findings most concentrated in category Uninitialized Variable. CSC B had no high concentration of findings. CSC C had findings most concentrated in Unreachable Code. CSC D had concentrations in Useless Assignment and Ignored Return Value with a lesser concentration in Negative File Descriptor. It is clear that no significant overlap regarding concentration of findings exists among the components. Each was developed by different developers, with some small overlaps in personnel, even though all components of CSCI A were developed within a single organization. Note that explanations of findings types are given in APPENDIX B.

Figure 6 – Concentration of Finding Types illustrates the distribution of findings. Defect type names are not shown for the sake of readability. The horizontal axis represents the number of findings per KSLOC. Ovals highlight the areas of concentrations of findings. *Table 3 – Defect Counts By Type Per CSC* found in *APPENDIX A* provides a detailed breakdown of the counts (rather than densities) of defects across types and components.

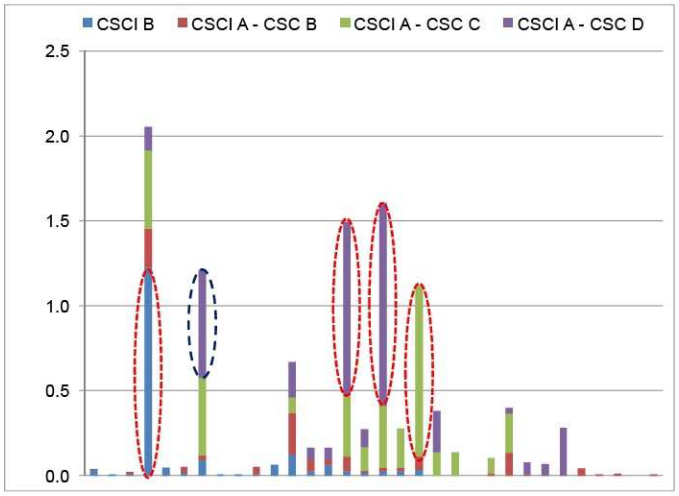

Figure 6 – Concentration of Finding Types

A second avenue to examine is the overall concentration of findings in each component. The overall concentration uses the same method as the previous comparison but considers all findings together regardless of category. *Table 1 – CSCI Sizes and Finding Densities* shows the findings density of the components analyzed using thousands of non-comment source lines (K-NCSL). This measure is given here since this type of automated source code analysis tool only operates on executable code, discarding comments and blank lines. Later portions of this report provide data in terms of KSLOC to allow comparison of data for use in other research efforts. It is noteworthy that the smaller components have a significantly larger finding density than the larger components. This finding agrees with *[2]* and *[3]*, which indicate that small modules have a significant share of overall defects in a system, possibly because smaller modules are overlooked as a source of errors because of the conventional wisdom that larger modules have higher defect densities.

Component	K-NCSL	Findings / K-NCSL
CSCI B	121	1.8634
CSCI A - CSC B	139	1.1571
CSCI A - CSC C	22	3.7156
CSCI A - CSC D	29	4.1332

Table 1 – CSCI Sizes and Finding Densities

C. Time to Produce and Review Findings

Execution of the tool and disposition of the findings for the four components identified required 124.5 man hours. The code analyzed consisted of 439 KSLOC in total of which 59 KSLOC were blank lines, 134 KSLOC were comments and with 311 K-NCSL. The source code was distributed over 1,245 separate source files. The time to identify each true positive finding was approximately 15 minutes. Future runs of the tool are expected to contain only a small fraction of the number of findings from this usage of the tool given that the current findings have already been identified and saved for future reference. Filtering capabilities distinguish findings that have already been reviewed from new findings.

Considerations of the implications of time to use the tool versus its benefits can be found in the section entitled *LOOKING FORWARD*.

IV. FINDINGS IN CONTEXT

A. Baseline Context Data

Data collection on the amount of time necessary to implement a software change request has occurred over the past year in the TDRSS ground segment and will continue as part of software process improvement initiatives. These preliminary data indicate that the time to implement a change request in response to a discrepancy report is 34 hours on average. This accounts for the software engineer's time to review and update requirements and design information, implement and unit test a change, and perform inspections on the requirements, design, and source code. Additional time must be considered for the Software Review Board (SRB) to approve the change, independent quality assurance review of changes, generation and execution of software and operations test and deployment plans, and delivery of the change to the operational system. These additional activities often operate on groups of change requests rather than individual change requests, so analysis was needed to determine the portion of hours from each software delivery to attribute to individual change requests.

The contents of eight deliveries in the calendar year 2008 were analyzed against project schedules for those deliveries to determine the amount of effort associated with each discrepancy report that is in addition to programmer implementation and unit test. The deliveries consisted of corrections to discrepancy reports as well as change requests in support of new capabilities to the TDRSS ground segment. Operational enhancements to the ground segment tend to get broken into relatively small units of work as part of the change request process in order to make the changes more manageable. Because of the way changes are subdivided, the proportion of configuration management, test, and delivery effort for an enhancement is very much the same as for a discrepancy report correction. Therefore, even though the proportion of enhancements and discrepancy corrections is not uniform in deliveries (approximately one-third of the content of deliveries in 2008 was discrepancy correction), the average amount of effort to correct a discrepancy can be simply stated as the total effort for all deliveries in the time period divided by the number of change requests in the same period. This number is approximately 13.6 hours per discrepancy report or change request. Combining this number with the implementation time referenced earlier, the average hours of effort to correct a single discrepancy report is 47.6 hours.

The average data loss per software discrepancy report was 0.35 hours for the period 2003 to 2007. When the year 2008 is included the average is reduced to 0.30 but this is artificial in that many discrepancy reports without data loss attributed were imported from a legacy discrepancy reporting system into the operations discrepancy reporting system in 2008 in order to consolidate reporting systems. Unfortunately, a clear method of automatically distinguishing legacy versus new discrepancies was not devised at the time and the 2008 numbers will be skewed until the reports can be manually sifted to separate old from new.

B. *Tool Findings in the Baseline Context*

Table 2 – Findings by Criticality shows the distribution of defects across CSCs as reported by the tool. Routine defects are those which should be fixed but have no strong driver as to how soon the fix should occur. These defects pertain more to maintainability of the code than correct operation. Urgent defects are those that are perceived to have noticeable operational impacts, where data corruption or processor resets—or similar operationally impacting events—will occur under specific sets of conditions. Urgent Loss captures the quantity of Urgent defects found with respect to the average loss per discrepancy report.

	CSCI B	CSCI A CSC B	CSCI A CSC C	CSCI A CSC D
Routine	200	130	78	118
Urgent	25	31	3	0
KSLOC	222	204	29	39
Defects/KSLOC	1.01	0.79	2.80	3.01
Urgent/KSLOC	0.11	0.15	0.10	0.00
Urgent Loss(hrs)	8.75	10.85	1.05	0.00

Table 2 – Findings by Criticality

Based on these data, it is expected that correcting the 59 urgent findings from the tool in CSCI A and B would result in an expected reduction of 20.65 hours of loss over a year, which is equivalent to nearly half of the software loss for 2008 and nearly a third of the average yearly loss recorded for the ground segment software.

C. *Comparison to External Entities*

Other groups have analyzed software defects, which may lend insight to the defect levels observed from this analysis. While the other groups did not restrict their observations to source code defects found by automated source code analysis tools, their results provide a point of departure for understanding the TDRSS ground segment results.

The Jet Propulsion Laboratory (JPL) performed a case study that found 1.2 fielded defects per logical KSLOC in ground software systems they developed *[4]*. Further, Capers Jones found that CMMI Level 5 organizations deliver 1.05 defects per KSLOC *[5]*. The findings reported by the tool account for a defect density of 1.18 defects per KSLOC in the CSCs analyzed (note for purposes of comparison KSLOC is used here rather than K-NCSL used in section *III*). The defect rate when limiting the focus to NCSL is necessarily higher.

Coverity, which develops a competing automated source code analysis tool, makes available results of its tool on open source projects *[6]*. At the time of this writing, Coverity reported an average of 0.605 findings per KSLOC over 96 projects. The findings reported were limited to the set of defects most easy to understand as opposed to all possible findings that can be reported by the tool. In this context, it appears that the open source community performed better than the mission critical software analyzed for this project; however, the limited set of defects reported by Coverity in its open source initiative may be a large factor in the differences reported. This curiosity cannot be answered in this report and is left to future work.

V. LOOKING FORWARD

To understand how to move forward with respect to CodeSonar in the ground segment software maintenance, costs and benefits must be weighed. The data collected in this initiative is sufficient to make an informed decision based on facts. These decisions use extrapolations of what was observed in this study to predict potential outcomes. Many of the data collected work well for these extrapolations but others may be less certain. For instance, this study focused solely on C and C++ source code while a large portion of the ground segment code is Ada. There is a version of CodeSonar for Ada but it remains to be seen if the defect rate and types in C and C++ versus Ada are largely the same or different. Previous work in *[7]* suggests that Ada has a lower overall defect rate. However, a well established factor to adjust the expected defect rate is not documented. So, we shall have to assume that the defect rate and types are the same. Also, the nominal amount of phase leakage of defects in the ground segment is not a known quantity so assumptions must be made here (phase leakage counts the number of defects that are not removed in the phase where created). Despite these uncertainties, reasonable arguments can be regarding the costs and benefits of CodeSonar specifically and automated source code analysis in general.

For the 204 most recently completed change requests, both addressing discrepancy reports and enhancements to the ground segment software, the average number of lines of code modified or created was 209 and the average number of programmer hours to implement and unit test changes was 48. Given these numbers in the context of an average of 1.18 defects per KSLOC found with the automated source code analysis tool in this initiative, each change request is expected to have 0.25 total defects or approximately 0.03 urgent defects associated with its change. Programmers would be injecting 53 total defects or 6 urgent defects not previously in the system that could be detected and prevented by use of the tool. For the sake of argument, consider that 10% of the newly introduced defects are discovered in test and require rework. Had the defect been found by the programmer, 6.29 hours of time for software test could have been prevented for each such defect per the statistics kept for time per change request. So, approximately 36 hours of wasted effort occurs by not using automated source code analysis here. Now, consider that an additional 10% of the newly introduced defects are propagated to operations and must be closed out via the discrepancy report process. It was determined that 47.6 man hours were needed to close a single discrepancy report. Using $50 per hour as an (low) estimate of the cost of the hours to perform work, each discrepancy caught in test costs $315 to rework and each discrepancy caught in operations costs $2,380 to correct. We correct six of each kind for a total cost of $16,170. The expected data loss for the defects assumed to enter operations is approximately two hours. Placing a dollar value on the data loss is a difficult thing to accomplish. Data loss in a critical timeframe, like a mission launch or spacewalk, could impact the health and safety of the mission. Loss of single pass of science data may carry lesser consequences.

Assuming that the initial purchase of an automated source code analysis tool is not an issue (already paid), the estimated annual maintenance cost to the vendor for CodeSonar is $50,000 for the more than eight million lines of code in the ground segment. Given these numbers and forgetting data loss for a moment, from a tool cost versus labor cost perspective the reasonable answer would be to only purchase CodeSonar for a subset of the ground segment software. The selection of software to use CodeSonar would be a balance of criticality of the software and fault proneness of the software. A mitigating factor in this line of reasoning is the issue of data loss. One serious incident that endangers the TDRSS fleet, astronauts, or causes hours of mission data loss requires much more effort than the typical discrepancy closeout process. Special teams are created in these circumstances to investigate root causes, taking into account all aspects of the ground and space segments segment. This can be an extensive process involving hundreds or, in the most severe cases, thousands of hours of effort. The question to answer is whether automated source code analysis tools prevent defect types that have historically caused these types of anomalies. Over the past three years, no severe anomaly (requiring thousands of hours of investigation) was identified that could have been prevented by automated source code analysis tools but approximately one to two significant anomalies (requiring hundreds of hours of work to close) each year might have been prevented by use of automated source code analysis tools. Again, for the sake of argument, assume that one such anomaly requires a total of 500 man hours of investigation and reporting at the same low estimate of $50 per man hour. One such incident requires $25,000 to investigate and resolve. Using conservative numbers and combining prevention of one significant incident a year with efficiencies gained in the maintenance process with the addition of automated source code analysis tools provides at savings dollar figure very near to that of the annual maintenance cost for all ground segment software. Similar arguments can be made for systems other than TDRSS where critical failures can cause serious or catastrophic results.

VI. Conclusion

The results of this study indicate that automated source code analysis technology is beneficial where high availability or proficiency is important. CodeSonar, specifically, and automated source code analysis tools, in general, are effective in finding source code defects not found by manual inspection and test processes currently employed by the TDRSS ground segment software sustaining engineering group. CodeSonar provides a unique advantage in this instance in that it has capability for C, C++, and Ada, which are the computer languages that constitute the vast majority of TDRSS ground segment software. Time to apply automated source code analysis tools is not prohibitive for use as part of the standard development process where an integrated findings database exists to permit search and filtering of results and where the tool can be run alongside or as part of the normal source code build process. While extensive costs comparisons were not performed for competing automated source code analysis products, long term costs for CodeSonar are in line with the benefits received. No major process changes are necessary to incorporate static source code analysis into an organization that already performs formal inspections. In conclusion, any organization that depends on internally maintained software for mission critical functions should strongly consider the addition of automated source code analysis to its product lifecycle. Products in this arena have achieved technical feasibility (at least for certain computer languages) at costs comparable to technical and resource savings received when applied to mission critical software.

References

[1] R.S. Pressman. "Test Case Design" in *Software Engineering, A Practitioner's Approach,* 4th ed., C. L. Liu, Ed. New York: McGraw-Hill, 1997, pp 453-454

[2] A. Güneş Koru et al, "Theory of relative defect proneness," *Empirical Software Engineering*, Vol. 13, Issue 5, ACM Press, 2008, pp. 473-498.

[3] C. Withrow, "Error Density and Size in Ada Software," *IEEE Software*, IEEE CS Press, 1990, pp. 26-30.

[4] J. Spagnuolo and J. Powell,"Defect Measurement and Analysis of JPL Ground Software: A Case Study," presented at the 7th annual Ground Systems Architecture Working Group, Manhattan Beach, California, 2004.

[5] C. Jones, *Software assessments, benchmarks, and best practices*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2000

[6] http://scan.coverity.com

[7] National Research Council Staff. "Detailed Comparisons of Ada and Other Third-Generation Programming Languages" in *ADA and Beyond: Software Policies for the Department of Defense*, National Academy Press, Washington, DC, 1997, pp. 92-100.

APPENDIX A

Table 3 provides a cross reference of defect types to CSCs.

	CSCI B	CSCI A - CSC B	CSCI A - CSC C	CSCI A - CSC D
Leak	4	1	0	0
Return Pointer to Freed	1	0	0	0
Missing Return Statement	1	2	0	0
Uninitialized Variable	148	32	10	4
Double Unlock	6	0	0	0
delete Object Created by new[]	2	5	0	0
Negative file descriptor	11	4	10	18
Accept on socket in wrong state	1	0	0	0
Listen on socket in wrong state	1	0	0	0
malloc Buffer Length Unreasonable	1	6	0	0
Dangerous Function Cast	8	0	0	0
Null Pointer Dereference	15	34	2	6
Type Overrun	3	10	0	2
Buffer Overrun	8	4	0	2
Ignored Return Value	3	12	8	29
Unused Value	2	2	3	3
Useless Assignment	3	3	8	34
Redundant Condition	3	3	5	0
Unreachable Code	4	11	22	0
File System Race Condition	0	0	3	7
memcpy Length Unreasonable	0	0	3	0
Free Null Pointer	0	0	0	0
Format String	0	2	2	0
Empty If Statement	0	19	5	1
Double Close	0	1	0	2
strncpy Does Not Null-terminate	0	0	0	2
Cast Alters Value	0	0	0	8
Buffer Underrun	0	6	0	0
Double Free	0	1	0	0
Null Test After Dereference	0	2	0	0
strncpy Length Unreasonable	0	0	0	0
Use After Free	0	1	0	0
Total	225	161	81	118

Table 3 – Defect Counts By Type Per CSC

APPENDIX B

The following table provides descriptions of the types of defects identified by CodeSonar in the TDRSS ground segment.

Defect Type	Description
Accept on socket in wrong state	A socket operation is performed on a socket that has not in the correct state for that operation.
Buffer Overrun	A read or write to data after the end of an object in memory.
Buffer Underrun	A read or write to data before the beginning of an object in memory.
Cast Alters Value	A cast operation causes a value to be changed.
Dangerous Function Cast	A function pointer is cast to another function pointer that has an incompatible signature or return type.
delete Object Created by new[]	An attempt to release memory obtained with new[] using delete.
Double Close	An attempt to close a file or socket that has already been closed.
Double Free	Two calls to free on the same object.
Double Unlock	A mutex has been unlocked twice with no intervening lock.
Empty If Statement	
File System Race Condition	File System Race Condition, which is also known as "TOCTTOU (Time of Check To Time of Use)" vulnerabilities occur when a function that uses a named file follows a function that checks a named file. The triggering functions for this warning class are the use functions.
Format String	A function that should have a format string passed in a particular argument position has been passed a string that either is not a format string or is from an untrusted source.
Free Null Pointer	An attempt to free a null pointer.
Ignored Return Value	The value returned by some function has not been used.
Leak	Dynamically allocated storage has not been freed.
Listen on socket in wrong state	A socket operation is performed on a socket that has not in the correct state for that operation.
malloc Buffer Length Unreasonable	A function is passed a length parameter that is negative or extremely large.
memcpy Length Unreasonable	A function is passed a length parameter that is negative or extremely large.
Missing Return Statement	At least one path through a non-void return-type function does not contain a return statement.
Negative file descriptor	An attempt to close a file or socket that has already been closed.
Null Pointer Dereference	An attempt to dereference a pointer to the zero page (that is, any address in the range 0..4096).
Null Test After Dereference	A pointer is NULL-checked when it must already have been dereferenced.
Redundant Condition	Some condition is either always or never satisfied.
Return Pointer to Freed	A procedure returns a pointer to memory that has already been freed.
strncpy Does Not Null-terminate	The "source" string in a call to strncpy does not contain a null terminator.
strncpy Length Unreasonable	A function is passed a length parameter that is negative or extremely large.
Type Overrun	Code overruns a boundary within an aggregate type.
Uninitialized Variable	An attempt to use the value of a variable that has not been initialized.
Unreachable Code	Some of the code in a function is not reachable from the function entry point under any circumstances.
Unused Value	A variable is assigned a value, but that value is never subsequently used on any execution path.
Use After Free	A dereference of a pointer to a freed object.
Useless Assignment	Some assignment always assigns the value that the variable being modified already has.