Measuring Software Complexity to Target Risky Modules in Autonomous Vehicle Systems

M. N. Clark, Bryan Salesky, Chris Urmson
Carnegie Mellon University

Dale Brenneman
McCabe Software Inc.

Corresponding Author:
M.N.Clark (clarkmn@cmu.edu)

Tartan Racing developed 300 KLOC that represented over 14,000 modules and enabled our robot car "Boss" to win the DARPA Urban Challenge. This paper describes how any complex software system can be analyzed in terms of its reliability, its degree of maintainability, and ease of integration using applied flow-graph theory. We discuss several code coverage measurements and why this is important in certifying critical software systems used in autonomous vehicles. Our paper applies cyclomatic complexity analysis to the winning DARPA Urban Challenge vehicle's software. We show graphical primitives followed by views of modules using those constructs. In this way minimum testing paths are quickly computed and viewed. We argue for customizing evaluation thresholds to further filter the modules to a small subset of those most at risk. This "choosing our battles" approach works well when teams are immersed in a fast-paced development program.

Submitted to
AUVSI North America Conference
San Diego, California
June 11, 2008

Track
Cross-Platform Autonomy

This work would not have been possible without the dedicated efforts of the Tartan Racing team and the generous support of our sponsors including General Motors, Caterpillar, Continental, and McCabe This work was further supported by DARPA under contract HR0011-06-C-0142.

Introduction

Tartan Racing [1] developed 300 KLOC that represented over 14,000 modules that enabled their robot car "Boss" to win the DARPA Urban Challenge [2]. Over twenty software developers spent 14 months implementing code in six areas: Behavior, Mission Planning, Motion Planning, Infrastructure, Perception, and the vehicle itself. They used a Debian variant of Linux (Ubuntu LTS) and built 35 binaries with SCONS [3] instead of make. The code was installed on 10 Intel core2duo blade servers that resided in the hatch area of a 2007 custom Chevy Tahoe.

The inter process communication was a distributed message passing system, supporting TCP/IP, UDP, and UNIX Domain Sockets as transport mechanisms. Processes communicated with one another locally on a machine via UNIX Domain Sockets, and remotely via TCP/IP. All messaging went through an agent that was located on each machine. Processes connected to the agent local to their machine, and all the agents connected to one another between machines in a full-mesh configuration. The system was formally tested for 65 days whereby two autonomous vehicles logged over 3,000 autonomous kilometers (1,864 miles) [4].

Such a system as described above is not uncommon for autonomous vehicles. While in development, these systems require testing for correctness, reliability, and maintainability. As these systems mature, performing software maintenance and modifications often overwhelm schedules and budgets. Fortunately, a series of metrics and related threshold values offers indications for when we should be concerned. At the very least testing should demonstrate:

- The program fulfills specification – Verification
- The program performs the required task – Validation
- Modifications to existing good code do not produce errors – Regression testing

However, this does not state which tests should be done or when the tests should be stopped. Furthermore, testing for quality attributes like maintainability and reliability are difficult enough when dealing with physical systems; when applied to autonomous vehicle software, the difficulty increases. However, quality features can be measured or deduced by inspecting the source code, assessing its features and looking for defects. This inspection technique is called static analysis. These can be manual or automated methods. A second method called dynamic analysis, tests for defects and code coverage by actually running instrumented software. The effort in both is to produce quantitative numbers about the software so resource decisions can be made.

The testing community generally agrees that the following items contribute to the quality of code [5]:

- Overall program structure
- Program design techniques
- Code readability
- Code Complexity
- Input/Output dependencies within programs
- Data usage within programs

This paper focuses on automated software complexity testing as it relates to an autonomous vehicle application: specifically Tartan Racing's vehicle Boss.

Code Size, Content, and Complexity

Measuring software attributes falls into two camps:

- Code size and content
- Code complexity

Code size is one of the easiest things to measure, giving rise to 'line count' software metrics:

- Lines of code
- Lines of comments
- Lines of mixed code and comments
- Lines left blank

Line counts do not tell us:

- How difficult it is to understand the code
- The mental effort needed to code a program
- The algorithmic complexity of code units.

One premise of using software complexity to measure software complexity is that bug-laden code is usually complex. Furthermore, complex components are also hard to understand, hard to test, and hard to modify. If we reduce the program complexity, the programs will improve. In fact, practical experience [6] has shown this to be true.

Halstead's theory of software metrics [7] has its roots in evaluating complexity of low level programs like assembly language. He proposed a short set of measurable operator and operand quantities and then used them in equations to predict program features.

McCabe's model for software metrics [8] determines the complexity of a program unit or module by measuring the amount of decision logic. The McCabe number is a predictive measure of reliability. In itself, the complexity value v(G) does not identify problems or errors but there is strong correlation between v(G) and code errors in case studies [6].

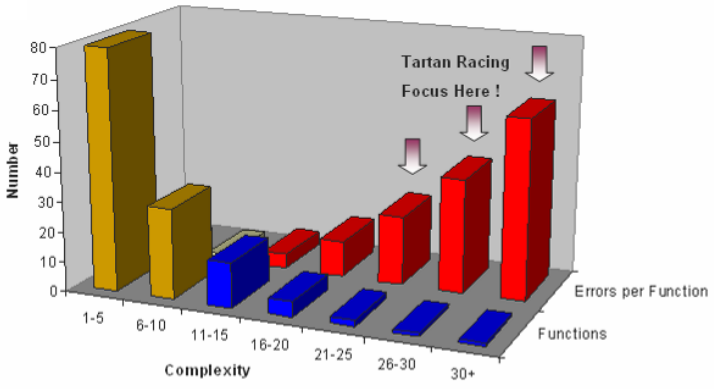


Fig. 1- Bug-laden code is usually complex

Both Halstead and McCabe models are applied to real projects and are supported by commercial tools [9] [10]. This paper describes using the cyclomatic complexity metric v(G), i.e. McCabe number.

Flow Graph Theory

The cyclomatic complexity metrics are described below. For any given computer program, its control flow graph, G, can be drawn. Each node of G corresponds to a block of sequential code and each arc corresponds to a branch of decision in the program. The cyclomatic complexity [9] of such a graph can be computed by a simple formula from graph theory, as:

$$V(G) = E - N + 2P$$

where

$v(G)$ = cyclomatic complexity

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components.

Cyclomatic complexity is alternatively defined to be one larger than the number of decision points (if/case-statements, while-statements, etc) in a module (function, procedure, chart node, etc.), or more generally a system.

Complexity and Software Quality Attributes

Quality Attributes are often called the non functional system requirements. They guide the system architecture, design and implementation. Understanding the quality attributes provides focus for the business drivers. Flow graph techniques make quantifying non functional properties like reliability, maintainability, and ease of integration possible. Not only can the techniques be automated, the results are non subjective and thereby helpful when making software resource decisions.

Reliability- Cyclomatic complexity v(G) is a size indicator in that it measures the number of logical paths in a module. It is also the minimum number of tests needed to forecast high reliability. Cyclomatic complexity is often referred to as the McCabe number. Modules with numbers below a threshold of 10 are considered reliable.

Maintainability- The Maintenance number ev(G) or "Essential" complexity measures the degree to which a module contains unstructured constructs. It is the reduction of cyclomatic such that only unstructured code remains in the flow graph. Studies show that code is harder to maintain when the ev(G) is above 4.

Integration- The Integration number iv(G) measures the decision structure which controls the invocation of a module's immediate subordinate modules. It is a quantification of the testing

effort of a module as it calls its subordinates. Modules with numbers above 7 are considered to require a significant amount of integration testing effort.

Figure 2 shows familiar software constructs as graphs. Each has nodes and edges.

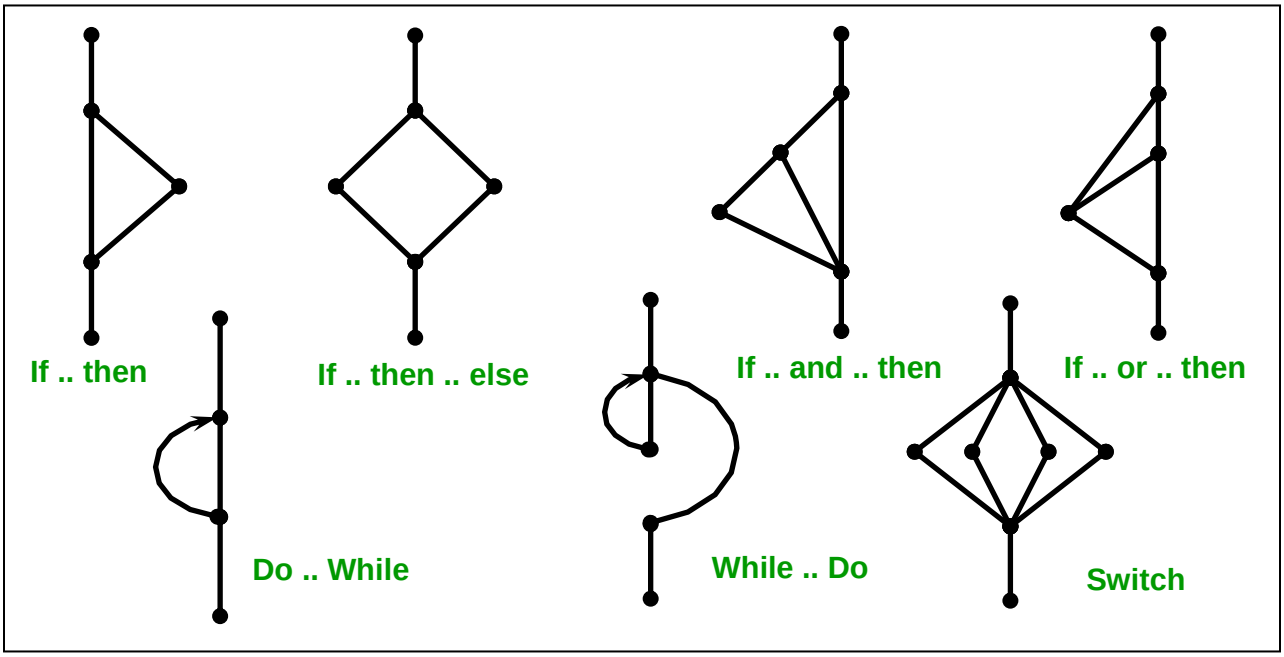


Fig. 2 - Flow Graph Primitives

Maintaining code is difficult when there is a high degree of branching into and out of structures as seen in Figure 3.

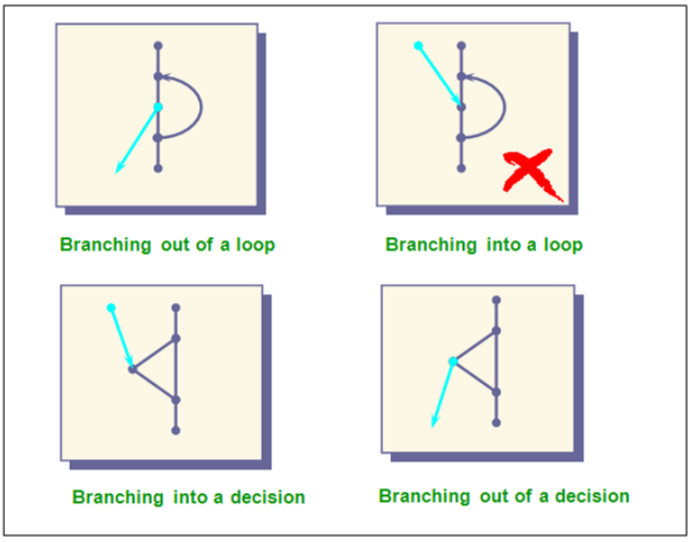


Fig. 3 - Unstructured Logic (hard to debug)

Minimum Path Coverage

Time limits us from testing all statistical paths in even the simplest of software structures. The module shown in Figure 4 has one switch, an 'if', and a loop statement. If one execution took one nanosecond, then testing all statistical paths would take 31 years. Instead, we test the number of unique closed loops in the module plus one. For example, in Figure 5 the software structure is a composite of the primitives described in Figure 2. There exist 9 closed conditions that result in 10 paths to be tested. This will cover all the code and decision logic.

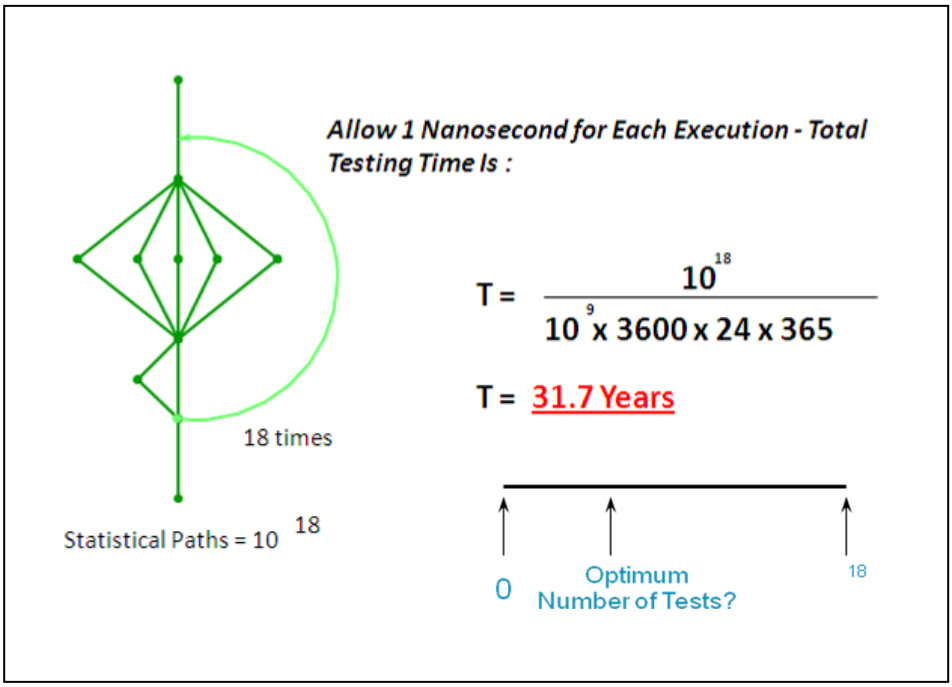


Fig. 4 - Test all statistical paths in a simple structure

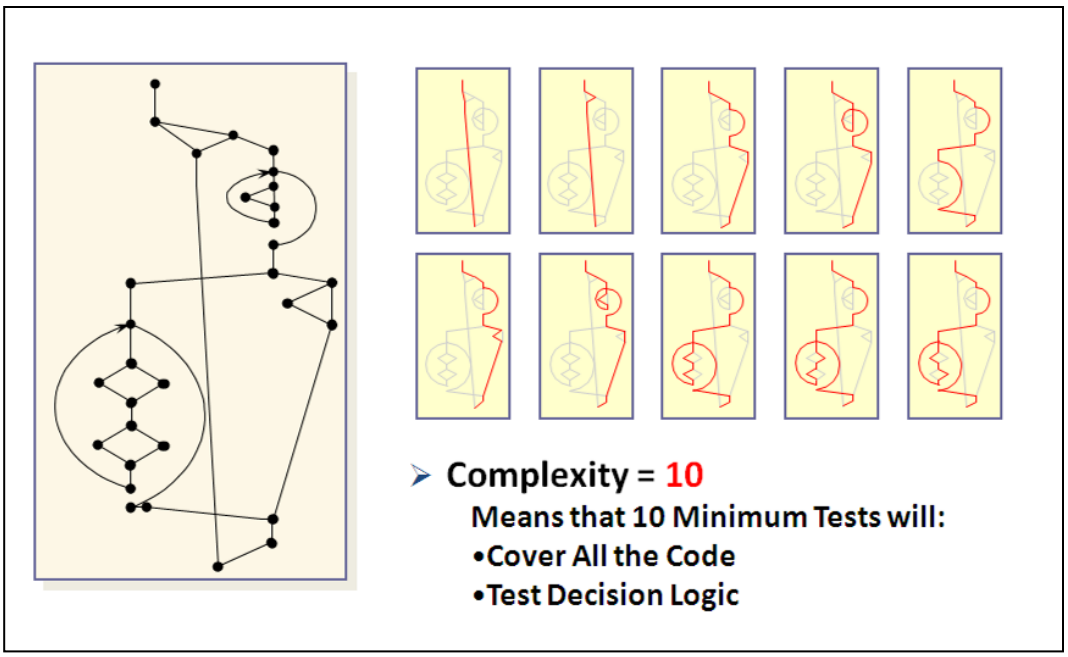
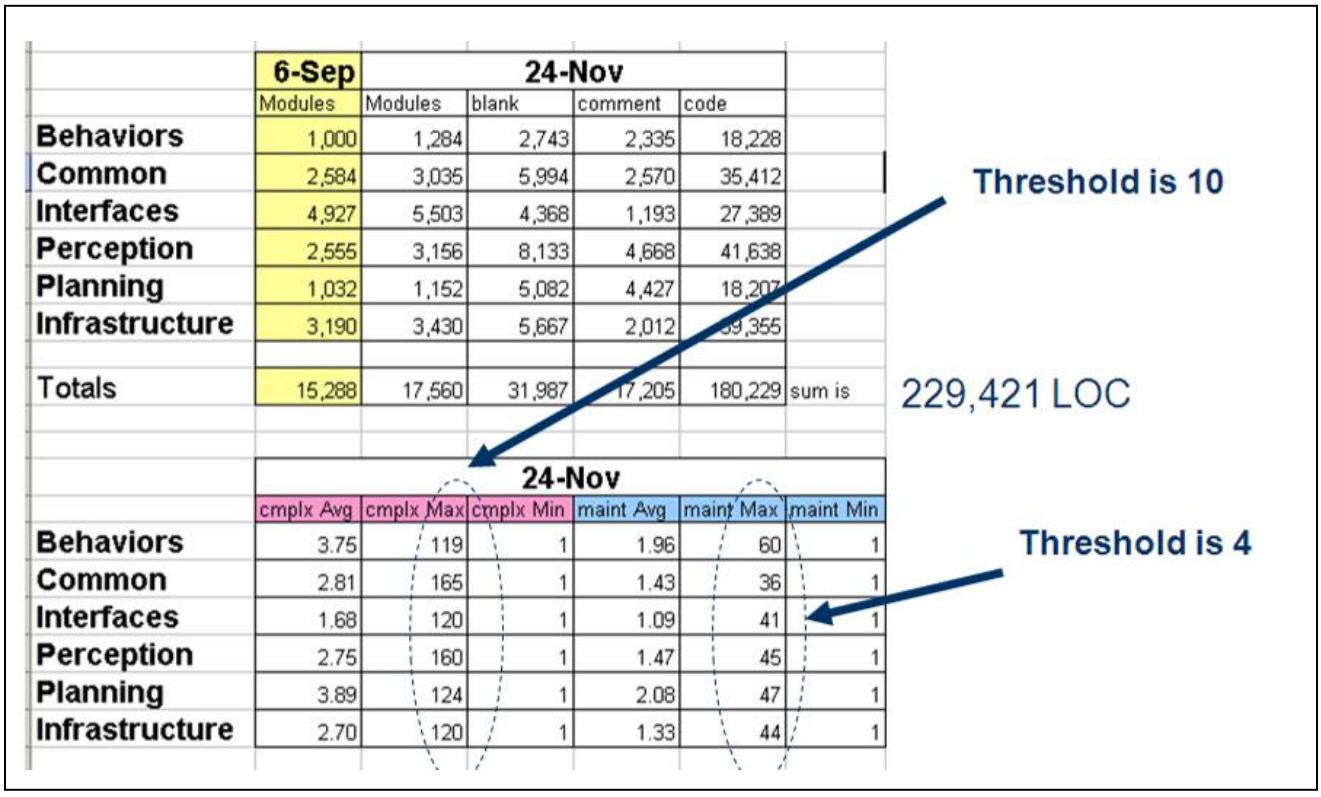


Fig. 5 - Minimum Test Paths

Selecting the Metric Threshold Values

Tartan Racing developed, implemented, and qualified their autonomous vehicle software in 14 months [12]. Over 20 software engineers contributed code. Complexity analysis started 10 months into the program. A static complexity analyzer [9] looked at six sections for code reliability and maintainability. Together the sections contained over 15,000 modules. In the top chart of Figure 6 shows the lines of code (LOC) while in the bottom chart the cyclomatic complexity and essential complexity (maintenance) are shown. The tool's computation time for each section was approximately ten minutes running on a 1.2 GHz processor. The threshold value for complexity is 10 and for maintainability it is 4. Tartan Racing's average numbers were well within the threshold values except for a few modules that were an order of magnitude over. These were identified to the software leadership in both list and scatter graphs forms. Figure 7 shows scatter graphs for three of the six software sections.



	6-Sep	24-Nov				
	Modules	Modules	blank	comment	code	
Behaviors	1,000	1,284	2,743	2,335	18,228	
Common	2,584	3,035	5,994	2,570	35,412	
Interfaces	4,927	5,503	4,368	1,193	27,389	
Perception	2,555	3,156	8,133	4,668	41,638	
Planning	1,032	1,152	5,082	4,427	18,207	
Infrastructure	3,190	3,430	5,667	2,012	99,355	
Totals	15,288	17,560	31,987	17,205	180,229	sum is

	24-Nov					
	cmplx Avg	cmplx Max	cmplx Min	maint Avg	maint Max	maint Min
Behaviors	3.75	119	1	1.96	60	1
Common	2.81	165	1	1.43	36	1
Interfaces	1.68	120	1	1.09	41	1
Perception	2.75	160	1	1.47	45	1
Planning	3.89	124	1	2.08	47	1
Infrastructure	2.70	120	1	1.33	44	1

Fig. 6 - LOC, complexity, and maintainability measurements

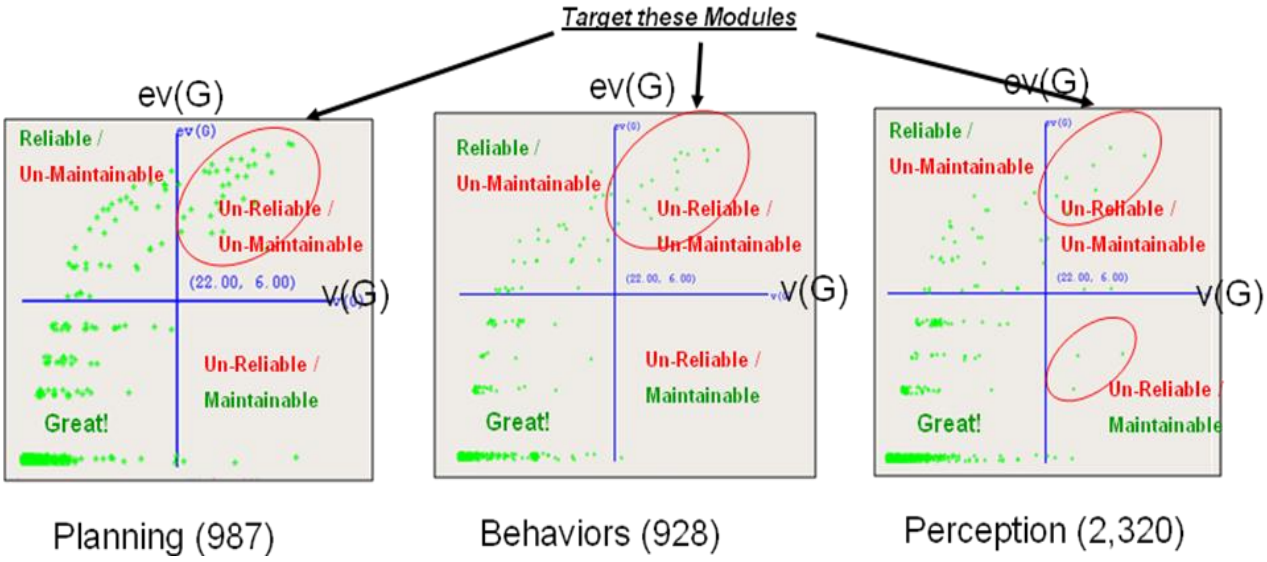


Fig. 7 - Example of Scatter Graphs

The modules of interest for Tartan Racing were in the scatter graph's upper right quadrant because these modules were both unreliable and not easy to maintain. The lower right quadrant was the least populated. The number that exceeded the threshold of 10 for complexity and 4 for maintenance were too many for the software team to review given their tight delivery schedule. Therefore, the analyst changed the complexity threshold from 10 to 22. This decreased the number of unreliable modules needed for peer reviews. The analyst raised the threshold based on the development team's software maturity level. He reasoned that in the ADA language, twenty-two can be an acceptable complexity threshold because of team discipline. That and a need to "pick our battles" were the reasons the analyst raised the complexity threshold.

Maintaining the code was secondary to having reliable code. While the developers knew intuitively which modules were hard to de-bug, a quantitative metric helped the software leader plan when modules would be ready. For similar reasons given for complexity, the analyst raised the code's maintenance threshold from 4 to 10.

Code Coverage Levels

Code coverage is a measure of what software ran during testing. When code coverage tools using source-code based techniques are used, it inherently means the code was instrumented, and coverage results are gleaned from log data. Code coverage analysis is especially useful for troublesome modules and program sections. Moreover, mission critical software should be validated according to a determined level of code verification. Figure 8 shows several increasing code coverage levels, with two alternatives for very thorough techniques for mission critical verifications.

The first code coverage level, "Module" is simply knowing the module was entered and keeping a count. The second level, "Line", counts whether any part of a line was performed. An example

is that the following line is counted as executed whether the decision's true outcome or false outcome is executed. Branching is not counted at this level.

if A = B do C else do E

For mission critical software like sections of autonomous vehicle code, one looks to at least the "Branch" coverage level. For the statement above, the true outcome of the decision would be counted as one branch, and the false outcome as a second branch, and the execution of each would be counted separately.

If more thoroughness is desired, then one of two alternatives might be taken, depending on the mission requirement. One alternative is Boolean or Modified Condition/Decision Coverage (MCDC) coverage. In the example of the following code,

if A or B or C do E

MCDC coverage requires that all three conditions of this decision be checked separately. For airborne software to be certified, it must comply with the FAA's D0-178B code structure coverage criterion. While DO-178B does not explicitly accept cyclomatic complexity data, depending upon the level of criticality (C, B or A) increasing code coverage is required. Level C requires that a test case (and requirement) trace to every code statement. Level B to every code branch (specified path), and Level A to every MCDC. DO-178C (due in 6-8 months) will allow some credit for modeling under some conditions.

Cyclomatic complexity coverage (or basis path coverage) as defined in the Structured Testing methodology [6] is a valid alternative for thorough code coverage. This methodology indicates that the minimum number of tests required for high reliability is equal to the Cyclomatic complexity; the number of linearly independent paths through a graph of a software unit.

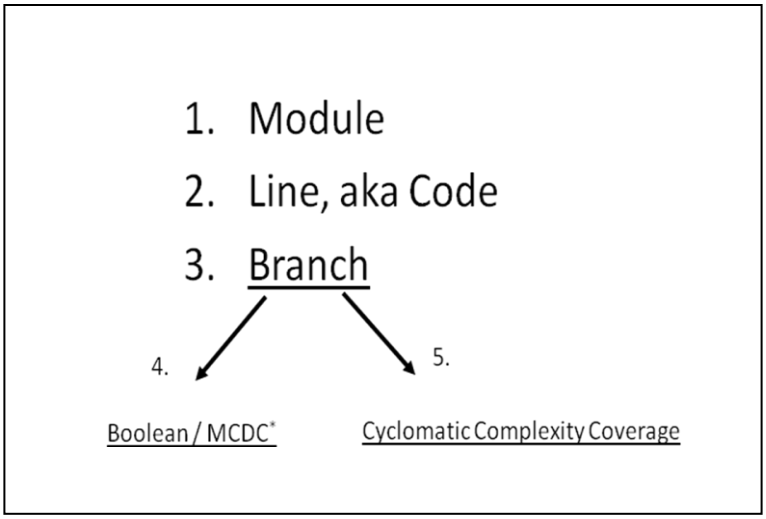
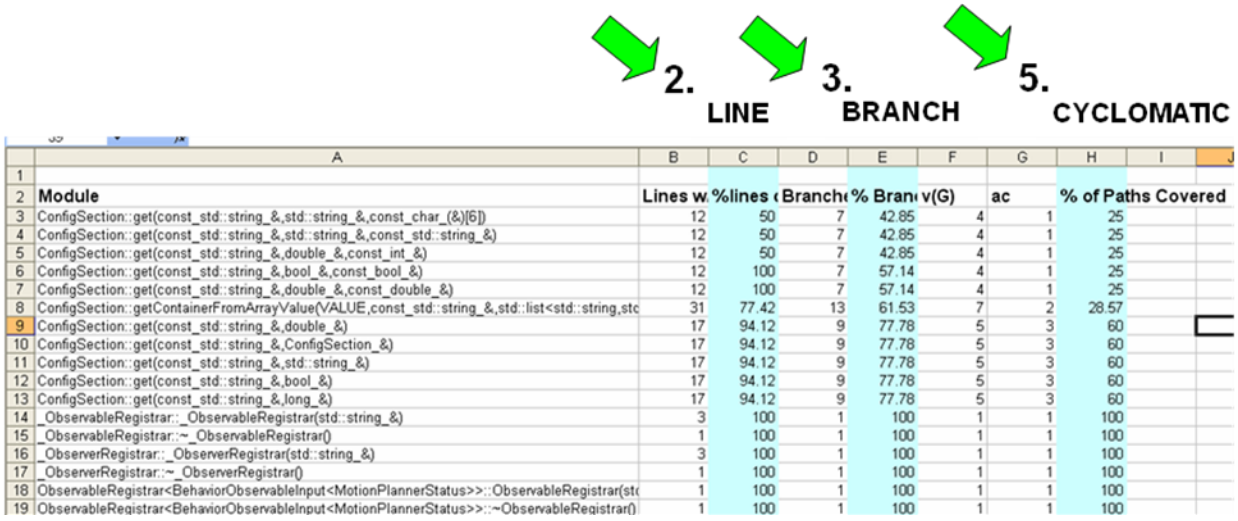


Fig. 8- Code Coverage Levels

Future Work

Tartan Racing's schedule and late exposure to complexity analysis did not allow for implementing code coverage. A proof of concept showed that the code could be easily instrumented and results achieved quickly, Figure 9. When the "BehaviorTask" binary was executed for one minute, 82 of 928 modules had some coverage. The tool [9] can provide all of the discussed levels of code coverage. Future work remains to evaluate the effects of instrumented code on the application's performance.



	Module	Lines w	%lines	Branch	% Bran	v(G)	ac	% of Paths Covered
3	ConfigSection::get(const_std::string_&,std::string_&,const_char_(&)[6])	12	50	7	42.85	4	1	25
4	ConfigSection::get(const_std::string_&,std::string_&,const_std::string_&)	12	50	7	42.85	4	1	25
5	ConfigSection::get(const_std::string_&,double_&,const_int_&)	12	50	7	42.85	4	1	25
6	ConfigSection::get(const_std::string_&,bool_&,const_bool_&)	12	100	7	57.14	4	1	25
7	ConfigSection::get(const_std::string_&,double_&,const_double_&)	12	100	7	57.14	4	1	25
8	ConfigSection::getContainerFromArrayValue(VALUE,const_std::string_&,std::list<std::string,stc	31	77.42	13	61.53	7	2	28.57
9	ConfigSection::get(const_std::string_&,double_&)	17	94.12	9	77.78	5	3	60
10	ConfigSection::get(const_std::string_&,ConfigSection_&)	17	94.12	9	77.78	5	3	60
11	ConfigSection::get(const_std::string_&,std::string_&)	17	94.12	9	77.78	5	3	60
12	ConfigSection::get(const_std::string_&,bool_&)	17	94.12	9	77.78	5	3	60
13	ConfigSection::get(const_std::string_&,long_&)	17	94.12	9	77.78	5	3	60
14	_ObservableRegistrar::_ObservableRegistrar(std::string_&)	3	100	1	100	1	1	100
15	_ObservableRegistrar::~_ObservableRegistrar()	1	100	1	100	1	1	100
16	_ObserverRegistrar::_ObserverRegistrar(std::string_&)	3	100	1	100	1	1	100
17	_ObserverRegistrar::~_ObserverRegistrar()	1	100	1	100	1	1	100
18	ObservableRegistrar<BehaviorObservableInput<MotionPlannerStatus>>::ObservableRegistrar(stc	1	100	1	100	1	1	100
19	ObservableRegistrar<BehaviorObservableInput<MotionPlannerStatus>>::~ObservableRegistrar()	1	100	1	100	1	1	100

Fig 9 - Proof of Concept- Code Coverage for "Behavior Task"

Summary

We established that there are metrics and related threshold values for software quality attributes like reliability, maintainability, and ease of integration. There are tools that quickly compute these values without running the application. Scatter graphs help us visualize the modules belonging to the quadrant of unreliable and hard to maintain domain. To prioritize which modules to select for analysis, the tool's standard threshold values were increased. The rationale was one of team software maturity level and the need to meet delivery of the application in four months. This helped focus the developers on the most at-risk modules. Moreover, software developers can tactically select which modules need peer-reviews based on the module's code complexity. Program managers can set realistic times to fix code based on the use of unstructured code.

We also established that Cyclomatic complexity can determine how many code coverage-related tests should be performed and when testing can stop. We presented several levels of code

coverage that can be measured, using instrumented software. The simplest was the module coverage level, then line coverage level. But for mission critical software like autonomous vehicles, at least branch coverage should be used, and for certain components requiring very high reliability, there were two recommendations; coverage of every Boolean condition (MCDC) or meeting the cyclomatic path or basis path coverage criteria. Certainly both can be applied. The benefit of the most thorough levels of code coverage is assurance that the right tests were performed to predict a low occurrence of errors. Equally important is that the code coverage process is automated, thus providing quick, non subjective results. Finally, work remains to understand how instrumented code affects the system's performance.

References

[1] Tartan Racing: http://www.TartanRacing.org
[2] DARPA Urban Challenge: http://www.darpa.mil/grandchallenge
[3] SCONS http://www.scons.org
[4] *Validating the Performance of an Autonomous* Car- AUVSI 2008 Proceedings- M.N.Clark et al.
[5] *Software Engineering for Real-Time Systems*-ISBN-10: 0201596202- Jim Cooling 2003
[6] *Structured Testing: A testing methodology using cyclomatic complexity metric,* A.H. Watson and T.J. McCabe, NIST special publication 500-235, 1996
[7] *Elements of software science,* M. Halstead, North-Holland, ISBN 0-137-20384-5, 1977
[8] *A Software Complexity Measure,* T. McCabe, IEEE Transactions on Software Engineering, Vol. 2, pp 308-320, 1976
[9] McCabe IQ- www.mccabe.com/iq_qa.htm
[10] Understand- http://www.scitools.com/products/understand/cpp/product.php
[11] Wikipedia-Cyclomatic Complexity -http://en.wikipedia.org/wiki/Cyclomatic_Complexity
[12] *Autonomous Driving in Urban Environments: Boss and the Urban Challenge* Journal of Field Robotics, accepted for publication- Urmson et al.