Best Practices for Autonomous Vehicle Configuration Management

Thomas Denewiler[a] and Mark Tjersland[a]

[a]Space and Naval Warfare Systems Center Pacific, 53560 Hull St, San Diego, CA, USA

ABSTRACT

The development of autonomous systems presents many challenges, including safety, repeatability of results, and the high cost of system testing. These challenges are compounded because of the nature of autonomous systems. The systems are composed of numerous interconnected components encompassing a broad spectrum of disciplines, including mechanical engineering, electrical engineering, computer vision, computer science, networking, and communications. By necessity, each component is often under simultaneous development with one or more other components. Because of the close coupling of components, changes in one component can have wide-ranging systemic effects. In this paper we describe a set of processes and tools that allow a distributed team to coordinate parallel development of software for autonomous systems without introducing system instability or performance regression. These include processes and tools for sharing software configurations, deploying software to autonomous system computers that lack internet access, building software in parallel across multiple machines, verifying software configurations, and automatically ensuring a specified level of quality exists using static analysis, unit tests, and regression tests. We build on existing best practices to ensure a stable baseline software configuration necessary for quantitatively assessing software changes without the need for impractically time-consuming data collection and testing. We have incorporated best practices from software development and robotics communities as well as from military requirements for fielding systems. These practices and tools give an increased confidence that the results from each test case capture the desired data due to proper software configuration, and that the configuration can be merged successfully into the baseline system if warranted by results. The processes described allow for parallel development paths to proceed simultaneously while minimizing the occurrence of unforeseen and difficult-to-resolve conflicts.

Keywords: Configuration management, autonomous vehicles, best practices

1. INTRODUCTION

Autonomous vehicles are complex systems that require a large amount of testing to achieve reliable operation. The goal of most groups developing autonomous vehicles is the same: to continuously and efficiently improve overall system performance. To achieve this goal it is necessary to measure system performance and to use an evidence-based approach when determining which changes to make permanent in the system. Testing autonomous vehicles is expensive due to labor-intensive requirements such as safety drivers and operators, and vehicle maintenance. Infrastructure costs, such as the cost of a test range, can also be high. The high cost of testing is one of the main motivations for improving efficiency in the data collection and data analysis portion of system development. Efficiency gains can be most readily found by minimizing setup work, ensuring that data is collected using the desired configurations, that those configurations will run as expected, and that tests are constructed such that differences between configurations can be observed with as few trials as possible.

The ONR Code 30 Autonomy program is composed of a large group of geographically distributed developers from government and private firms that have been working for the past four years on developing autonomous driving capabilities in rugged, off-road, congested environments using low-cost sensors. Some developers work on a single subsystem, but most developers find it necessary to work on multiple subsystems simultaneously to create the desired improvement in overall system performance. There have been numerous lessons learned

Further author information: (Send correspondence to Thomas Denewiler)
Thomas Denewiler: E-mail: thomas.denewiler@navy.mil

Unmanned Systems Technology XIX, edited by Robert E. Karlsen, Douglas W. Gage, Charles M. Shoemaker, Hoa G. Nguyen, Proceedings of SPIE Vol. 10195, 101950N · © 2017 SPIE
CCC code: 0277-786X/17/$18 · doi: 10.1117/12.2267995

over the years in terms of how to support a large team, a complex code base, and extensive testing in multiple locations. We hope that at least some of the practices adopted for the ONR Code 30 Autonomy program will be useful to other groups planning similar development efforts with autonomous vehicles.

This work will describe methods to handle multiple configurations, verify configurations, test code, track changes and test results, and perform quality assurance of code. Incorporating best practices for software design, development, and testing increases the value of limited and expensive testing events while also making it easier for developers to successfully improve system performance.

2. CONFIGURATIONS

In order to evaluate the effect of a change on system performance it is necessary to have a baseline configuration to measure against. The baseline configuration must have low variance in outputs and behavior when given similar inputs over repeated tests. In other words, the system should drive the same roads following the same trajectories from day to day while identifying and avoiding objects in a repeatable, deterministic manner. When a configuration accomplishes this we consider that configuration to be stable. By maintaining a stable baseline configuration it is possible to test changes using a smaller number of trials than would be necessary if the baseline had a different success rate per test from day to day. When using statistical test methods, such as analyses of variance, to determine if a significant difference exists between two configurations, then metrics — such as reliability and confidence intervals — can be achieved by decreasing the variance or increasing the number of trials.[1] Therefore, with a limited testing budget it is imperative to maintain a stable, low-variance baseline configuration to minimize the number of expensive, time-consuming trials. In Section 5 we discuss some automated ways that make it more likely a configuration under test will be stable.

Typically, software is maintained in three different configurations: stable, testing, and development. Supporting a large number of development configurations that any developer is free to experiment in and make any changes they desire allows for exploration of new ideas. After initial test results show system improvement, quality assurances are completed successfully, and code reviews are performed, then development code is tested on live vehicles. If vehicle testing is successful then the development code moves into the testing configuration.

On a periodic schedule, typically two or three times a year, the testing configuration is frozen, meaning no code changes for a few weeks, and extended vehicle testing is performed. If extended testing shows that the testing configuration has improved performance over the previous stable configuration and that the system performance is itself stable then the current testing configuration becomes the new stable configuration.

3. TESTING

In order to facilitate the comparison of vehicle testing results over time and across systems it is important to track the test method configurations and test results in a version control system alongside code repositories. Tests must be updated over time in order to provide differentiation between the baseline configuration and development configurations. An example of this is performing a slalom test. Initially the space between required turns can be ten vehicle lengths. As time progresses and the system performance is improved the space between required turns can gradually be reduced to five vehicle lengths. During evaluation one of the test metrics could be time taken to complete a slalom course of a given length. Without tracking the space between required turns it is possible to misinterpret earlier results as showing that system performance has actually degraded despite the earlier system being unable to even complete the new test method configuration.

3.1 Test and Evaluation Master Plan

In order to compare test results consistently across time and across configurations a test and evaluation master plan (TEMP) is required.[2] The TEMP details test methods and operational conduct such that results generated on varying systems with a diverse set of operators in multiple environments can be compared to draw accurate conclusions. The NIST Robot Evaluation Exercise test methods[3] were used as a blueprint for creating new test methods appropriate for passenger vehicle size autonomous vehicles. It is strongly recommended that the TEMP be stored in version control so that past results can be considered within the context of the test parameters at the time the test was conducted.

When appropriate test methods should be defined by parameters that are a function of vehicle size and target speeds. This allows for a single test method to be used for multiple sized vehicles where the physical boundaries of the test are adjusted to accommodate a specific vehicle. When testing precise navigation components the lane widths should be decreased for smaller vehicles. When a system is intended to drive faster than others then the test method should ensure that properties such as perception horizon and stopping distance are exercised for the faster speeds.

4. TOOLS

Configurations can be made more stable by fixing static code analysis issues, testing in simulation, and using playback of recorded sensor data from multiple locations. Static code analysis includes enforcing the initialization of all variables and limiting the number of branching paths through a function (which makes the construction of unit tests easier). Testing using simulation and playback of recorded data are particularly useful for testing edge cases that are difficult or dangerous to recreate with live vehicle testing.

4.1 Version Control and Issue Tracking

Issues are tracked for the ONR Code 30 Autonomy program via Atlassian JIRA. Source code is stored in Git repositories with Atlassian Stash providing a Web-based frontend. A distributed version control system, such as Git or Mercurial, is highly recommended for multiple users in a distributed team environment. Github or Gitlab would work equally as well for the workflow described in this paper.

4.2 Operating System Software

A standard operating system is used to help achieve consistent results across multiple testing platforms. For the ONR Code 30 Autonomy program the standard operating system is based on Ubuntu 14.04 with a custom set of system packages installed. This operating system is installed on each autonomous vehicle computer as part of hardware installation, and the system packages are updated once per month.

Apt and pypi mirrors are used to update the system software packages for computers without external Internet access. These mirrors are very convenient because they allow deployment of updates to multiple simultaneous vehicles over a private vehicle network. The computer containing the mirrors can be taken into the field during testing where Internet connections are either unreliable or non-existent. The mirrors allow for both the platform computers and developer computers to be updated (in case one of the computers is missing a certain package or new capabilities are being developed in the field).

4.3 Local Git Repositories

A tool we developed, called Robogit, is used to track all git repositories and branches used on the vehicles and push bare git repositories to a single computer on each robotic platform. This allows many workspaces to be created on each vehicle and those workspaces can use any branches needed for a specific configuration, whether a development, testing, or stable configuration. The use of a single repository location per vehicle means that all vehicle machines can be configured to point to the same upstream source code location. This consistency eliminates a potential source of confusion when working in the field.

4.4 Workspace Management

An arbitrary number of development configurations are supported with the ROS (Robot Operating System) concept of workspaces. The ONR Code 30 Autonomy program heavily leverages ROS[4] as the system middleware that allows inter-process and network-distributed communications. ROS includes a tool called `wstool` to facilitate workspace creation and updating.

Each workspace is a collection of local repositories at a specified version, where the version is typically a branch or a tag from the repository.

When testing proposed changes a workspace is configured with the current baseline configuration, and then only those repositories affected by the proposed changes are modified. This creates a straightforward method of creating identical workspaces across multiple developer computers and on the robotic platforms. Test results can easily be duplicated during test and verification steps. When using Git the commit hashes of each repository in the workspace can be captured and used to define the exact state of the software used for any given test.

A workspace manager was created that is built on top of Python Fabric. The workspace manager executes workspace configuration commands in parallel across a set of remote computers. This streamlines the creation and verification of workspace configuration, especially when creating identical workspaces on multiple computers. This tool allows a developer to specify a directory name to contain the workspace, a vehicle to create the workspace on, and a variable number of configuration files that specify which repositories and branches should make up the workspace.

After a workspace has been configured and the repositories are cloned, then the workspace manager builds all the code in parallel across machines. This can save a lot of time and reduces the chances of typos made by a developer logging into each machine separately to run the same commands repeatedly.

After building the code it is then critical to verify that each workspace is configured as expected. The workspace manager helps here as well by using a configuration specification file that defines repositories used, the ROS workspace layout, and the hash of each repository that identifies a specific version of code. The same information is then gathered from each computer for the current workspace and compared to the expected configuration. Any differences are displayed by the workspace manager tool. In addition, the workspace manager checks to see if any of the repositories have local modifications that do not cause the repository hash to change but that can cause the software to behave in a way that is not intended by that configuration. Local modifications can cause a repository to ignore merging of upstream changes. If this condition is ignored then older versions of the software or temporarily disabled behaviors can mistakenly be used instead of the desired configuration.

This is a largely automated process that provides users, developers, and testers a tremendous amount of confidence that the entire system is configured properly. By configuring the system properly reasonable expectations of vehicle operation can be set, and test results are expected to be meaningful.

5. QUALITY ASSURANCE

The Department of Defense (DoD) defines an autonomous software category as “software capable of exercising autonomous control authority over potentially safety significant hardware systems, subsystems, or components without the possibility of predetermined safe detection and intervention by a control entity to preclude the occurrence of a mishap or hazard”.[5] Software in this category that is capable of catastrophic results (such as causing damage to people or property) is required to have a review of requirements, architecture, design, and code. By utilizing the following tools and tracking results over time it is possible to be highly prepared to demonstrate to the DoD that a system is ready to be deployed.

By incorporating continuous build systems into the standard development cycle and checking for safety issues the ONR Code 30 Autonomy Program satisfies several characteristics of successful system safety programs as specified in the Unmanned Systems Safety Guide.[6] Those characteristics are paying attention to safety early and often, building safety expertise, and creating a positive safety culture.

5.1 Static Analysis

As the lead systems integrator the government has a particular interest in the ability to maintain and extend the features present in a codebase. Therefore, as part of automated testing, several tools are used to ensure that code follows conventions intended to improve readability and allow code to be understood by developers not involved in the original creation of the code. The ONR Code 30 Autonomy program enforces coding standards via a code compliance tool called Statick.

Statick is a framework for performing software analysis using a variety of tools that perform static code analysis. The outputs from many tools are gathered and formatted so that the results can be displayed in a continuous integration server such as Jenkins. The use of source code analysis tools supports a secure software development life cycle, reduces the number of software vulnerabilities in a system, and improves the overall reliability of software.[7] Statick allows for multiple levels of rigor to be established where each level corresponds to a different set of configurations of the individual tools. A plugin system is used to integrate each analysis tool, allowing third parties to write interfaces for new analysis tools. The currently supported tools include `catkin_lint`, `clang-format`, `clang-tidy`, `CMake`, `cmakelint`, `CppCheck`, `lint`, `lizard`, `Make`, `pylint`, `xmllint`, and `yamllint`. The current configuration for each tool is listed in Appendix A. The Joint Software Systems Safety Engineering Handbook[8] has a large number of recommendations that can help direct the specific set of flags to use for each tool.

Those tools provide valuable insights and suggestions on ways to improve the operation, maintenance, and extendability of source code.[9] The result is that the code is more likely to produce robust, repeatable, deterministic outputs that make system performance more reliable and easier to debug. This leads to a decrease in the number of times that the system changes behavior when operating in the same environment and lowers the variance of system performance.

Bundling these individual tools into a single framework provides several benefits over using each tool on its own. Most static analysis tools have complex flags to control what checks are performed by the tool. Statick handles this by storing those flags in a configuration file per level so that individual developers do not have to memorize the flags. Statick also provides a flexible way of suppressing false positives compared to standard inline ignores and allows for ignoring some files completely. The output of each tool in Statick is gathered into a single place and reported to the developer via the command line, but those outputs are also formatted appropriately for displaying the results in Jenkins when performing automated builds.

Table 1 shows the history of warnings generated by Statick along with a few other metrics related to the code base. The number of warnings shown ignores whitespace changes. The introduction of static analysis tools has been incremental, as has been the severity of the flags used to configure each of the tools. This has allowed developers to become accustomed to the coding standards, but has also been driven by the discovery of bugs that are now prevented from occurring in the first place. A concerted effort has been made recently to remove unused code and to address warnings generated by the latest flags. Note that, when using the current tools and flags, `v4.0` had some issues that cause some of the code analysis tools to hang and the tests were not completed. The flags used at the time `v4.0` was released did not have these issues.

Table 1. Static Analysis History

Version	Files	Source Lines of Code	Warnings	Warnings / SLOC
v3.0	1237	201629	953	0.0047
v4.0	2625	390691	N/A	N/A
v4.1	2722	401293	3036	0.0076
v5.0	2328	306860	1467	0.0048
v6.0	2145	276015	1095	0.0040
v6.1	1225	144628	700	0.0048
v6.2	1398	173843	343	0.0020

5.2 Simulation

Simulation is a standard practice when developing unmanned systems. Robot simulations can range from very basic simulation of sensor inputs to high fidelity 3D simulators such as Gazebo. Gazebo was developed into the DRCSIM, which was used as a qualifying stage for the DARPA Robotics Challenge as a condition for some participants receiving additional funds.

Where possible, simulation environments should be designed to mimic situations that can be expected to be encountered in the robot's use case. These environments can be set up to the same specifications and dimensions as the test methods used for system tests.

Simulation will, of course, never cover all cases that could be encountered when testing a live unmanned system. However, if a piece of software does not work in simulation then it can be reasonably expected that the software would not work on a live system. The simulation can then be used to get that software to a state where there can be some evidence that it has potential to work in the actual system.

For the ONR Code 30 Autonomy program we configured a Stage simulator to run through as many of the courses described in the TEMP as possible. This included slalom courses, long loops with sharp turns, and urban environments. With the Stage simulator we were able to test the entire software pipeline except for perception. Recently, we have begun using Gazebo with the same test environments with the added benefit that perception algorithms can be tested at the same time. When a software configuration is able to successfully navigate in these simulated environments there is a much greater possibility that the actual vehicle will perform well with that software configuration.

5.3 Unit Tests

Unit tests allow small pieces of software to be tested at a very low level. These tests can be designed before adding a feature to make sure that the feature to be implemented works as intended. Tests can also be added for existing code to test that changes do not break currently working functionality.

One difficulty in developing unit tests for unmanned systems is that there are many components that require hardware interaction. A driver that communicates with a specific hardware interface can't be tested if that hardware isn't present, such as when testing on a remote build server.

Unit tests also have problems when working with distributed systems, such as a robot running ROS. The interactions between nodes, especially if they have some timing dependency, can be difficult to isolate and test properly. In these cases, the expected output of an upstream node can be recorded to a file and later passed as input to a downstream node. In the cases where these interactions can't be recorded to a file, higher level tests at the subsystem or system level will need to be developed.

5.4 Regression Tests

Once a performance baseline has been established, it is desirable to make sure any changes do not break functionality that previously worked. Regression testing helps with this and is particularly well suited to the study of edge cases that are difficult to recreate. Events that occur at remote locations, are dangerous scenarios, or rarely occur on their own during normal operations are great candidates for regression testing. It is necessary to acquire data that exhibits the need for specific actions to be taken by the system. That data should then be used with the software in an offline mode and labels must be added to describe the desired actions. The actual actions of the software are then measured against the desired actions to ensure any proposed changes are still capable of handling all scenarios as desired.

The use of regression tests allows a program to build confidence that they are always progressing and monotonically increasing system performance over time. Successfully completing regression tests decreases the likelihood that capabilities will silently fail because they are not being fully exercised by challenging edge cases.

5.5 Continuous Integration

Continuous integration is the practice of pulling software changes into the mainline branch of code as often as possible. This is typically facilitated with workflows such as Git pull requests for code review and software such as Jenkins for automating steps of software verification. The use of the continuous integration server software allows for the code review to focus on the design logic of the change by catching more subtle issues such as static analysis warnings automatically. The continuous integration server can also run unit tests to check that the changes do not add any regressions before it is integrated into the software mainline. Without following continuous integration practices, software branches can end up as very long-lived branches and become increasingly more difficult to integrate into a constantly changing mainline as time goes on.

5.6 Pull Request Builder

Prior to merging new code into the mainline branch a request is submitted to the continuous integration server in the form of a single rosinstall file that describes which repositories are to be modified from the baseline configuration. A new workspace is created that contains the new configuration. The code in that workspace is then built and installed, unit tests and regression tests are run, and then all of the static analysis tools are run. All of the tests must run successfully in order for code to be merged. If the proposed changes modify the runtime behavior of the system then test results from simulators or live vehicle testing must accompany the automated tests before the new code is merged.

Automating the build, unit testing, and static analysis process saves a considerable amount of developer time. When code reviewers are able to see an automated report that shows syntax and style guidelines are followed it frees them up to focus on the design architecture, logic, and potential edge cases of the changes.

6. CONCLUSION

Developing and testing autonomous vehicles is challenging in many regards. Many configuration management practices can ease the difficulties and provide a solid foundation for increasing system performance over time. In this paper we have highlighted practical lessons learned from autonomous vehicle development that can help with sharing software configurations accurately across developer machines and vehicles, deploying software, building and verifying software configurations, and performing quality assurance with automated tools. These practices are based on many best practices from the software engineering industry and have served us well in improving the performance of our own systems while providing a solid foundation for future improvements.

ACKNOWLEDGMENTS

This work has been supported by the Office of Naval Research. The views expressed are those of the authors and do not reflect the official policy or position of the funding agency or the U.S. Government.

APPENDIX A. STATIC ANALYSIS CONFIGURATION

The current set of flags used for all static analysis tools are listed here.

- `make`: `-Wall -Wextra -Wuninitialized -Woverloaded-virtual -Wnon-virtual-dtor -Wold-style-cast -Wno-unused-variable -Wno-unused-but-set-variable -Wno-unused-parameter`
- `catkin_lint`: `-W2`
- `cppcheck`: `-j 4 --suppress=unreadVariable --suppress=unusedPrivateFunction --suppress=unusedStructMember --enable=warning,style --config-exclude=/usr --template='[file:line]:  (severity id) message'`
- `xmllint`: Default settings.
- `yamllint`: `-d 'extends:  default, rules:  colons:  max-spaces-before:  0, max-spaces-after:  -1, commas:  disable, document-start:  disable, line-length:  disable'`
- `cmakelint`: `--spaces=2 --filter=-linelength, -whitespace/indent`
- `clang-tidy`: `-checks='*, -cert-err58-cpp, -cert-err60-cpp, -clang-analyzer-deadcode.DeadStores, -clang-analyzer-alpha.deadcode.UnreachableCode, -clang-analyzer-optin.performance.Padding, -cppcoreguidelines-*, -google-readability-namespace-comments, -google-runtime-int, -llvm-include-order, -llvm-namespace-comment, -modernize-*, -misc-unused-parameters, -readability-else-after-return'`
- `clang-format`: Based on included Google style with slight modifications to match ROS conventions.
- `pylint`: `--disable=I0011,W0141,W0142,W0511 --max-line-length=100 --good-names=f,x,y,z,t,dx,dy,dz,dt,i,j,k,ex,Run, --dummy-variables-rgx=' (+[a-zA-Z0-9]*?$$)|dummy*'`

REFERENCES

[1] Collins, J. C., [*Binomial Distribution: Hypothesis Testing, Confidence Intervals (CI), and Reliability with Implementation in S-PLUS*], U.S. Army Research Laboratory, First ed. (2010).

[2] Defense Acquisition University, [*Test and Evaluation Management Guide*], Defense Acquisition University, Sixth ed. (2012).

[3] Jacoff, A., Huang, H.-M., Virts, A., Downs, A., and Sheh, R., "Emergency Response Robot Evaluation Exercise," in [*Proceedings of the Workshop on Performance Metrics for Intelligent Systems*], 145 − 154, ACM (2012).

[4] Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., "ROS: an Open-Source Robot Operating System," in [*ICRA Workshop on Open Source Software*], (2009).

[5] Department of Defense, [*MIL-STD-882E: System Safety*], Department of Defense, Fifth ed. (2012).

[6] Department of Defense, [*Unmanned Systems Safety Guide for DoD Acquisition*], Department of Defense, First ed. (2007).

[7] Foreman, J., Fisher, D. A., Brune, K., Bray, M., Gerken, C. M., Gross, J., Haines, C. G., and Kean, E., [*C4 Software Technology Reference Guide – A Prototype*], Software Engineering Institute, Carnegie Mellon University, First ed. (1997).

[8] Department of Defense, [*Joint Software Systems Safety Engineering Handbook*], Department of Defense, Third ed. (2010).

[9] Wilson, G., Aruliah, D., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P., "Best Practices for Scientific Computing," *PLOS Biology* **12**, 1–7 (01 2014).