PREPRINT: G. Meyer & S. Beiker (eds.) Road Vehicle Automation 2
Lecture Notes in Mobility, Springer, 2015, pp. 163-170

A Philosophy for Developing Trust in Self-Driving Cars

Michael Wagner[1] **and Philip Koopman**[2]

[1] Carnegie Mellon University, Robotics Institute, mwagner@cmu.edu

[2] Carnegie Mellon University, Department of Electrical and Computer Engineering, koopman@cmu.edu

Abstract

For decades, our lives have depended on the safe operation of automated mechanisms around and inside us. The autonomy and complexity of these mechanisms is increasing dramatically. Autonomous systems such as self-driving cars rely heavily on inductive inference and complex software, both of which confound traditional software-safety techniques that are focused on amassing sufficient confirmatory evidence to support safety claims. In this paper we survey existing methods and tools that, taken together, can enable a new and more productive philosophy for software safety that is based on Karl Popper's idea of falsificationism.

1 Trusting self-driving cars

For decades, our lives have depended on the safe operation of automated mechanisms around and inside us. However, the autonomy of these mechanisms is increasing dramatically, going from comparatively simple drive-by-wire control to fully autonomous self-driving cars. The safety risks posed by autonomous systems cannot be mitigated through mechanical interlocks or similar tried-and-true techniques. Furthermore these autonomous systems will operate in unstructured environments (including highways not designed for self-driving cars and unpredictable weather conditions) that will present a myriad of unexpected situations. Without question, automation holds the promise of reducing the *rates* of accidents; for example, self-driving cars have the potential to virtually eliminate accidents due to inattentive drivers. However, when he pays attention, a human driver has tremendous capacity for reacting responsibly to circumstances for which he has not been explicitly trained. With the human out of the loop, the autonomous car is far less capable of handling unforeseen circumstances. By definition, an "unstructured environment" such as a real-world road network includes plenty of unforeseen conditions. This lack of predictive capability demands new verification techniques to allow us to justify trusting self-driving cars in our everyday lives.

Questions of whether or not to trust a new technology are often answered by testing. Ambitious vehicle-centric test campaigns, for example, can examine the road worthiness and crash safety of a new passenger car. The world is only beginning to understand that serious vehicle risks can be posed by software implementation defects or, perhaps even more perniciously, weaknesses in software architectures and design. Testing remains too focused on evaluating software safety with only too few observations. Even thousands of test-miles are unlikely to detect low-rate systematic defects in software that operates an entire fleet of cars. Across a fleet of vehicles, this software will be subjected to billions of hours of use, so even low-probability failures will certainly occur repeatedly.

Furthermore we hypothesize that the kinds of risks posed by complex software are dissimilar to those posed by human drivers. Software can avoid becoming drowsy after a long drive, for example. But in exchange we must consider the "nonsensical" and unpredictable behavior caused by code defects. We must also consider a wide array of traditional and nontraditional security concerns; the nature of what constitutes a security exploit in a self-driving car is quite different vulnerabilities exposed by traditional IT software [1].

In response, industries have adopted safety standards governing how software is developed. Nearly all of today's standards define *processes* that must be employed when creating and validating the software that go far beyond just testing and address the entire software lifecycle. But the processes prescribed by today's standards cannot scale to software on which self-driving cars depend. Self-driving car software exhibits:

- Far more lines of source code; for example, compare the number of operations involved in planning and control of a driverless vehicle to those required for throttle control in passenger cars.
- High cyclomatic complexity needed to implement driving behaviors [2].
- Very high-dimensional interfaces to transmit rich perceptual data.
- Novel algorithms, especially involving machine learning and adaptation.

In some cases, the very methods needed to enable advanced autonomy – such as machine learning – are disallowed by standards because these methods are perceived as too risky.

2 Inductive inference

Beyond complexity, there is another important difference between traditional control software and the perception, planning, and control algorithms needed to realize self-driving cars. In traditional control software, deductive inference logically links basic safety principles to implementation. For example, we know from first principles that when a driver releases her foot from the accelerator pedal, she wants to reduce engine power to an idle. So in this case, we create a requirement that the throttle closes whenever the pedal is released, and meet the requirement through traditional reliability techniques such as redundant pedal-angle sensing. Early robotics research tried to use deductive reasoning to construct sophisticated rule sets that fully defined behavior. This resulted in robots that could to perform well-defined, controlled tasks in specific environments, but were brittle to real-world variations. This explains the early proliferation of fixed-base, factory robots and the lack of self-driving cars on the roads today.

Over the past decades, robotics has made spectacular gains using *inductive* inference, such as machine learning, which may not produce semantically understandable rules, but rather finds correlations and classification rules within training data. For example, we no longer define *a priori* rules for visually classifying pedestrians; instead, we train a classifier using labeled images of pedestrians. A machine-learning algorithm automatically finds which visual features are most effective at discerning the two classes. Based on the decisions of this classifier, the self-driving car then decides how to accelerate, brake, and steer. In many machine-learning applications, a human expert serves as a "safety net" for handling unanticipated problems; for example, the decisions of a medical-diagnosis algorithm might be confirmed by a doctor. Alternately, such systems are seen as automating the "easy" case and deferring the unusual or tricky cases to a human expert for resolution. But a human cannot be counted upon to act as a safety net for a self-driving car, precisely because of the car's excellent performance in typical conditions. The passenger is unlikely to observe erroneous behavior first hand, thus he may learn to (unjustifiably) trust the car and won't monitor its behavior for errors.

Inductive inference can yield excellent performance, in nominal conditions. We can observe examples all around us, from targeted ads to spam filters, which are easily able to satisfy requirements for 90% accuracy. In comparison, self-driving cars have much more stringent accuracy requirements (easily 99.9999%). Yet machine learning has trouble achieving this level of accuracy and, when it can, it is often due to overfitting [3]. We can argue against overfitting by evaluating performance on realistically-distributed test sets. However, this is just a restatement of our original verification problem, our inability to collect an epistemologically sufficient quantity of empirical data, which led us to induction in the first place.

Perfect (or approximately perfect) classifier accuracy could also be achieved by finding a perfectly discriminative feature space in which to classify data. However the discovery of such a space would obviate the need for machine learning. Any (nontrivial) trained classifier exhibits a tradeoff between accepting misses and accepting false positives. The less of one that you require, the more of the other that you must accept. This tradeoff is described in graphs like Figure 1, which shows the performance of several visual pedestrian-detection algorithms. Note that this tradeoff is fundamental, and not caused by to code defects. However, code defects are present in all software, and can only degrade potential accuracy.

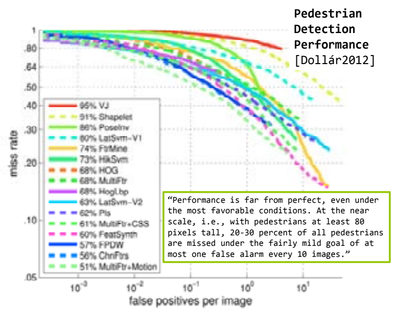

Fig. 1. Performance of sixteen pedestrian-detection algorithms from [4].

Can we improve the performance of pedestrian detection by combining results from a set of classifiers? This idea, called boosting [5] has been successfully used in many applications. However, in the context of safety we concern ourselves not with nominal classifier performance, but with worst-case behavior. From this perspective we must account for common-mode errors that can be observed to occur across otherwise diverse redundant software versions [6].

Inductive inference can produce unexpected behavior in low-probability, off-nominal conditions. Such conditions are, by definition, not well represented in training sets, and thus tend to be overwhelmed in the data. Developers have trouble anticipating what constitutes off-nominal conditions, which biases training data away from very rare "black swan" events. Test sets, on which an algorithm's performance is measured, constitute a subset of the total collected data, which further reduces the chances that developers examine performance in off-nominal conditions.

3 Falsificationism and software safety

At this point the Luddite (or, the practitioner of traditional safety-engineering techniques) may throw up his arms and denounce self-driving as hopelessly unsafe. However this point of view also seems unjustified, because in most conditions automation has the potential to perform spectacularly. It seems reasonable to expect self-driving cars to reduce the number of driving fatalities; but fatalities will continue to occur, and their nature is likely to be distinct from accidents that occur today. Most notably, injured passengers will be largely blameless for their accidents if the car is driving itself. We are obligated to understand and to mitigate those risks that remain to the extent possible. How can we do this?

The verification philosophy we advocate is based on the ideas of Karl Popper, who framed science not as a constructive process of building theories (which is inductive) but rather as an adversarial process of falsifying proposed theories with experimental results [7]. This use of "denying the consequent" (i.e., *modus tollens*) is powerful. One needs only a single negative example to falsify a theory, while a theory's proponent must demonstrate that it holds in *all* cases, which is epistemologically untenable. This led Popper to the observation that a theory must be *falsifiable* to be meaningful. We are of the opinion that safety cases – like scientific theories – should also be falsifiable to have significant value. This point of view places extraordinary power in the hands of the tester. No longer should her goal be to collect mountains of confirmatory test results. Working toward this goal can lead to confirmation bias. Rather, we argue her goal should be to find the negative test result that motivates ongoing, iterative software improvement. By analogy, her position is that of the pioneering physicist understanding under what experimental conditions a well-established theory fails, which in the end produces more complete scientific understanding.

Structured safety cases [8] and other argument structures [9] go a long way toward reaching this goal, because they make logical arguments explicit along with the evidence supporting them. We must also shift away from viewing safety cases as costly artifacts that are expected to remain unassailable over time, and instead develop automated tools for iterating safety cases efficiently as we learn more about the technologies and applications they cover.

Run-time verification is a powerful means of examining whether a given implementation upholds the key assumptions of its safety case. Run-time verification employs formal logic – ideally compatible with logics used in model checking – to precisely capture claims and safety-case assumptions, and then compiles them into a specification. Our research employs modified versions of metric temporal logic [10] with state-machine descriptions used to encode mode-based state. Then, a run-time monitor observes an executing system and detects deviations from this formal specification. This can serve two purposes. First, it can detect subtle "cracks" in a safety case. We have used this approach to uncover violations of safety properties in an automotive development platform [11], even with only very limited source-code access. Run-time verification can also be used to mitigate risks from hazards posed by a complex software controller. Our group used an informal form of run-time verification for risk-mitigation on the Autonomous Platform Demonstrator [12]. In this case, a simple *safety monitor* developed at a high level of rigor was responsible for enforcing the key safety requirements for the vehicle, such as maintaining a speed limit. This made a strong safety case practical to build even on a complex unmanned vehicle.

Although these approaches can require a lot of creativity to apply properly – and it is nontrivial to apply them to more complex safety requirements – there is tremendous benefit in even just pondering how to formally specify safety properties of a complex system. Too often, such properties are left ambiguous, which can lead to significant and misunderstood gaps in a safety case.

When employing a *modus tollens* approach, one also seeks powerful testing tools that can *generate* observations that invalidate a safety case and thus motivate improvements. Field testing, such as accumulating on-road miles, is indispensable for validating technology, at least initially. At some point continued field testing yields diminishing returns, especially considering its high cost. Simulation is often an attractive alternative to field testing, but the test inputs generated in simulation will be biased toward those of expected conditions (i.e., one will never simulate a black swan).

Software-testing techniques collectively known as *robustness testing* have been shown to be cost-effective for finding unanticipated vulnerabilities in a number of contexts. *Fuzz testing* is a naively simple and highly automated robustness-testing technique that regularly finds zero-day exploits in the realm of security research. Findings by Miller [13] [14] repeatedly show notable robustness vulnerabilities in mature operating systems and utilities.

But fuzz testing's purely random input generation can be inefficient at searching for vulnerabilities. The Ballista project [15] improved search efficiency by using dictionaries of historically "interesting" values to test. Ballista generated test inputs based on the data types of function parameters of a module under test rather than the functionality of the module itself. Many APIs use far fewer types than functions (for example, POSIX requires only 20 data types as arguments for 233 functions and system calls). Thus, Ballista completely ignores the purpose of a function, and builds test cases exclusively from data type information (see Figure 2, left). Armed with only interface information, Ballista found "system-killer vulnerabilities" in mature, commercially available operating systems (see Figure 2, right).

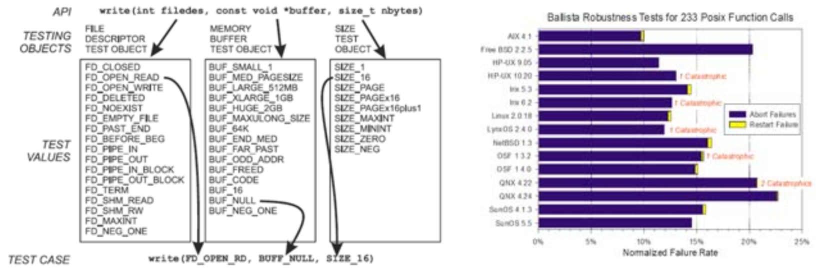

Fig. 2. Left: Ballista test case generation for the write() function. The arrows show a single test case being generated from three particular test values; in general, all combinations of test values are tried in the course of testing. Right: Robustness testing on operating systems revealed a significant task abort failure rate and several catastrophic system-killer failures.

More recent work applies techniques from Ballista to the robustness testing of autonomous vehicles. While operating systems and IT software often use a query/response model, autonomous vehicles use stateful control loops. While SCADA software also uses control loops, autonomous-vehicle control tends to involve perception data that is orders of magnitude more complex.

The Automated Stress Testing for Autonomy Architectures (ASTAA) project (see Figure 3) explores now Ballista's test-generation techniques can be applied to the kind of complex software. An important consideration for ASTAA is defining what constitutes a test failure. How should the control system under test behave when subjected to the kinds of unexpected inputs that robustness testing generates? One generally does not have access to a *test oracle* that is capable of answering that question. ASTAA avoids the need for unrealistically complex test oracles and instead monitors invariants on safe behavior that should always hold, regardless of any specific test input. Safety invariants can be drawn from a safety case; for example, speed limits, reaction time, and output bounds. In this new domain ASTAA has achieved the same kind of effectiveness we saw on Ballista. We have found vulnerabilities well over twenty systems on the project. These systems span the range of communications, control, perception, planning, and basic infrastructure functions. ASTAA testing has found safety problems caused by software bugs, flawed architecture design, communication failures, environmental conditions that exceed design parameters, inconsistent internal state, and gaps in system testing.

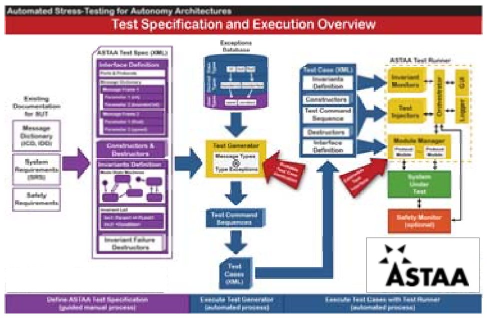
Automated Stress-Testing for Autonomy Architectures
Test Specification and Execution Overview
Existing Documentation for SUT
Message Dictionary (ICD, IDD)
System Requirements (SRS)
Safety Requirements
ASTAA Test Spec (XML)
Interface Definition
Constructors & Destructors
Invariants Definition
Invariant Failure Destructors
Exceptions Database
Test Generator
Message Types
Type Exceptions
Test Command Sequences
Test Cases (XML)
Test Case (XML)
Invariants Definition
Constructors
Test Command Sequence
Destructors
Interface Definition
ASTAA Test Runner
Invariant Monitors
Test Injectors
Logger
GUI
Module Manager
System Under Test
Safety Monitor (optional)
ASTAA
Define ASTAA Test Specification (guided manual process)
Execute Test Generator (automated process)
Execute Test Cases with Test Runner (automated process)

Fig. 3. The ASTAA architecture [16].

4 Conclusions

Self-driving cars of the near future will rely heavily on inductive inference and complex software to operate safely. Traditional software-safety techniques are not up to the task of analyzing and mitigating risks they will pose. We argue that a new software-safety philosophy, drawn from Popper's falsificationism, is indispensable for deciding whether to trust the software in control of an autonomous vehicle. Such a philosophy is a sharp departure from traditional software verification, in particular because it concedes that the problem of induction is insurmountable. However, its use of *modus tollens* empowers the tester. No longer assigned the unrealistic task of compiling never-ending supporting evidence, the tester instead becomes the pioneering scientist, on a quest to collect experimental observations that, in our context, motivate safety improvements.

5 References

[1] K. Koscher et al., “Experimental Security Analysis of a Modern Automobile”, Proc. IEEE Symposium on Security and Privacy, 2010.
[2] D. Ferguson et al. "A reasoning framework for autonomous urban driving", IEEE Intelligent Vehicles Symposium, 2008.
[3] D. J. Leinweber, "Stupid Data Miner Tricks", Journal of Investing, 2007.
[4] P. Dollár et al., “Pedestrian Detection: An Evaluation of the State of the Art”, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 34, No. 4, 2012.
[5] Schapire, Robert E. "The strength of weak learnability." Machine learning 5.2 (1990): 197-227.
[6] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming." IEEE Trans. on Sw. Eng., (1986): 96-109.
[7] Popper, Karl, “The Logic of Scientific Discovery”, Basic Books, New York, NY, 1959.
[8] T. Kelly & R. Weaver, “The Goal Structuring Notation – A Safety Argument Notation”, Proc. Dependable Sys. & Networks 2004, Workshop on Assurance Cases
[9] S. Toulmin, “The Uses of Argument”, (1958) 2nd ed., ISBN 0-521-53483-6
[10] J. Ouaknine & J. Worrell, "Some recent results in metric temporal logic." Formal Modeling & Analysis of Timed Systems, Springer Berlin Heidelberg, 2008. 1-13.
[11] Kane A., Fuhrman T., Koopman P. “Monitor Based Oracles for Cyber-Physical System Testing”, In Dependable Systems & Networks, 2014.
[12] M. Wagner et al., "Building safer UGVs with run-time safety invariants," 2009 Nat’l Defense Industrial Assoc. Systems Engineering Conference
[13] B.P. Miller et al., "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33, 12 (December 1990)
[14] B.P. Miller et al., "An Empirical Study of the Robustness of MacOS Applications Using Random Testing", 1st Int’l Workshop on Random Testing, 2006.
[15] Koopman, P. & DeVale, J., "Comparing the Robustness of POSIX Operating Systems," Fault Tolerant Computing Symposium, June 1999
[16] ASTAA project web page, http://www.nrec.ri.cmu.edu/projects/stress_testing

6 Full author’s information

Michael Wagner
The Robotics Institute, Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
United States of America

E-mail: mwagner@cmu.edu

Phil Koopman
Department of Electrical and Computer Engineering, Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213
United States of America
E-mail: koopman@cmu.edu

7 Keywords

Software safety, autonomous vehicles, self-driving cars, inductive reasoning, software robustness testing, runtime verification