What Embedded Software Engineering Can Learn from Enterprise IT Testing Techniques

Technical Whitepaper

Abstract

Embedded software engineering has become a much bigger and more complex domain than we could have imagined. As devices are expected to communicate with other devices and embedded subsystems, a much larger surface area has emerged for defects that threaten the safety, security, and reliability of the software. For example, the connected car not only introduces software safety and security concerns within the car as a system, interactions with environmental components, such as communicating with 'smart traffic lights' and vehicle-to-vehicle communication, potentially expose additional risk. Additionally, as car makers develop and merge functionality into 'the autopilot' mode, driver-assist technologies have become safety-critical technologies.

Embedded software organizations have always taken a 'shift-left' approach to software quality, rigorously applying defect prevention techniques early in the lifecycle. The demand for IoT requires a new testing paradigm that more closely resembles the challenges that Enterprise IT have faced for decades. As enterprise IT struggles to 'shift-left', embedded systems are struggling to 'shift-right' by testing more componentized and distributed architectures.

This poses a significant challenge associated with team structure, as well as the traditional challenges enterprise IT has been struggling with for years, such as:

- Targeting the right granularity of the unit test
- Solidifying automation at the message layer
- Gaining access to fresh, realistic, and credible test data
- Accessing the distributed environment in order to test continuously

 © 2017 Parasoft Corporation | Perfecting Software

In this paper, we will discuss how to implement lessons from enterprise software development to overcome the challenges the changing world of embedded software engineering presents. As we discuss these concepts within the context of automotive software engineering, you will learn how to:

- Combine data from coverage analysis and message layer testing technologies, which provides insight into application readiness and traceability across testing practices.
- Emulate dependent systems with service virtualization, which removes testing constraints and increases flexibility across complex test environments.
- Test at scale when scalable hardware is unavailable or impractical.
- Leverage penetration testing techniques to expose security vulnerabilities and trace their impact directly to the code base.

IoT; Security; Automotive; Software Testing; API; Service Virtualization;

Overview

The global automotive software market is expected to reach $10.1 billion by 2020, according to a 2014 report by Global Industry Analysts, Inc. What's more, software content now represents nearly 21 percent of a vehicle's value, a dramatic shift from 40 years ago, when vehicles were comprised of mechanical/structural components and rudimentary electronics. Today's automobiles are highly-advanced cyber-physical devices driven by software.

Why the increasing emphasis on software? Because it's the most effective way (and many times, the only way) to provide the connectivity and innovation that today's consumers demand. Consumer demand is forcing automotive suppliers and manufactures to develop software applications that are, in many cases, larger and more complex than their IT business-application counterpoints. Along with this shift, the following automotive software trends are forcing development teams to rethink development process, approach, and tooling requirements:

1. Open Architecture (Open Standards, Messaging, Protocols):

Software-driven and software-assisted connected cars are becoming increasingly complex. In fact, the Chevrolet Volt EV has more than 10 million lines of code and over 100 electronic controllers – the "king of software cars," according to Wired magazine. Because of this complexity, the auto industry is increasingly relying on an open standards approach to software development, using enterprise architectures, hardware, messaging protocols, operating systems (i.e., Linux), and software languages. The similarity in architecture, OS, and language makes today's automobile software very similar to advanced enterprise applications.

2. 24x7 Connectivity via Public Internet:

Increasingly challenging for modern automobile software is its ever-expanding attack surface. Interconnecting the 30-year old CAN bus with modern protocols, such as Wi-Fi, 3G, Bluetooth, and GPS, brings significant new software challenges for automobile manufacturers. Vehicle buyers are increasingly demanding "con-

nected" vehicles, which offer features such as hands-free calling, in-vehicle web browsing, navigation with real-time traffic data, and real-time weather updates, but which result is an outdated CAN platform being exposed to the Internet and public cellular networks. Unfortunately, as security researchers have already demonstrated, these high-tech interfaces are the perfect entry point through which to gain access to the CAN bus, where a vehicle's controls and safety systems are open to be manipulated. To mitigate these risks, developers need to augment their development and testing processes and tools, learning from their enterprise colleagues.

3. Software Configurable and Remote Updates:

As connected cars have become a reality, the landscape of in-vehicle infotainment (IVI) has evolved. While formerly hardware-defined, IVI systems are now multi-function systems that support a combination of vehicle telematics and diagnostics, passenger entertainment, and driver assistance. The 2015 report Strategy Analytics predicts "an enhanced role for infotainment software and operating systems as they become more important to infotainment system functionality and ease of use." IVI systems are now mirroring the type of software and ease-of-use requirements for enterprise applications.

Software Validation Best Practices

For many years, the IT industry has struggled with complex multi-tiered systems that integrate multiple technologies and services together to quickly provide market-leading innovations. Today's embedded software industry is facing very similar challenges. Monolithic applications written in C, entirely isolated from the external world, are becoming a rare exception. Systems now require connectivity. Messages are propagated from the lowest microcontroller layer up to server applications in the cloud, and of course in the reverse direction as well. An example is Tesla's Model S connected car, which wirelessly communicates via cellular 3G/4G or your private Wi-Fi, when safely parked in your garage, to provide diagnostic, remote control of selected features such as checking battery charging status. There are many similar examples in the medical industry and in industrial automation.

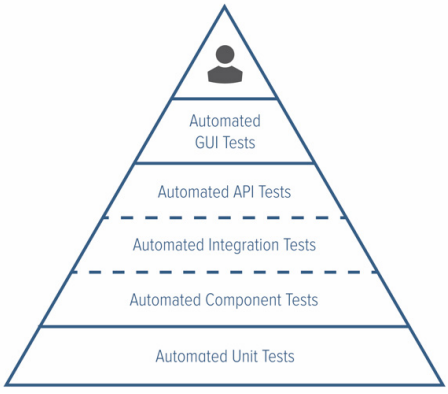


Fig. 1: Software Testing Pyramid

With this shift from individual isolated components to connected systems and services, the approaches to validation and verification also need to change to meet the increased complexity of the systems being designed. Often this complexity is coupled with the introduction of agile development practices to enable more accurate delivery to market – and this further compounds the challenges faced by traditional testing philosophies. At this point, we can leverage the practices and techniques already used in development by enterprise IT organizations to solve comparable problems in the world of highly-connected service oriented architectures.

Automated API Testing and Service Virtualization

Where traditionally embedded development organizations have focused on the bottom of the testing pyramid (see Fig. 1) – automating unit testing to test isolated components – enterprise organizations on the other hand have focused on the top of the pyramid, traditionally heavily leveraging 'lower cost' and 'quicker' manual testing. However, manual tests are neither 'lower cost' or 'quicker' and do not scale as the systems grow in complexity.

Automated unit testing is often too expensive for enterprise IT organizations because it takes too much time and requires sophisticated resources/expertise to create meaningful tests. Conversely, manual testing is perceived to be 'quicker' because everything is wired together and is 'lower cost' because only basic domain expertise is needed to perform the testing required. Unfortunately, this ignores the cost and complexity of the test infrastructure and the time taken between a defect being injected into the codebase and it being found via testing. Furthermore, manual testing efforts are much more costly when it comes to traceability and compliance requirements, since they require extensive paperwork and auditing of both the results and the tests themselves.

But there is a middle ground that has been especially efficient for enterprise IT when developing within open architectures, like those required to support IoT systems, leveraging automated API Testing and Service Virtualization (see Fig. 2).

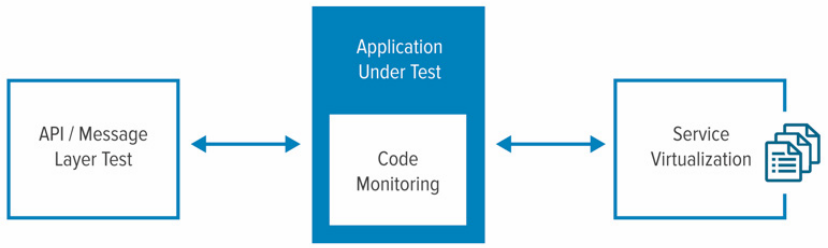


Fig. 2: API Testing and Service Virtualization

By automating tests at the Message layer, testing can be more easily automated and created with minimal 'development skills.' Effective API testing tools enable developers to create simple service or operation level 'unit tests' that QA/testers can then take and extend into more complete functional tests. By extending both data leveraged by the individual tests and chaining tests/operations together into scenarios, developers can take output of one operation and pass it to another without the need to write complex scripts or code.

This makes creating and running the tests easier, but still leaves the problem of how to convert these tests into 'continuous tests.' These tests are automated, meaning that a human does not have to manually execute them, but they still have a dependency on the test environment. This is one of the primary advantages of unit tests: unit tests are isolated from their dependencies using either stubs or mocks, and can therefore easily be run as part of a Continuous Integration workflow.

When working with the API, we have the application, or component, under test deployed and configured, but the dependences might be unavailable – or we might not be able to configure them to provide the behavior or data we need. As the systems become more connected, the testing infrastructure grows in complexity, ultimately becoming constrained by the availability and control over the connected subsystems. This is where

Service Virtualization helps. Service Virtualization simulates the dependent services and provides reusable assets that can easily emulate functional behavior, data, and performance characteristics.

Code Traceability Throughout The Pyramid

Another advantage of unit testing is easy integration of code coverage reporting; however, this is another area where enterprise IT has found a middle ground. Leveraging advanced runtime coverage analysis, it is possible to provide traceability to the code tested as part of these API tests, correlated back to the original requirements. This type of advanced coverage and merging of coverage from across different testing practices provides ability to analyze data from across the testing pyramid and open the door for more advanced process improvements such as 'change based testing.' Leveraging code analysis to determine changes in the underlying codebase, in conjunction with code traceability, provides the ability to identify which tests should be performed to validate the user-stories being worked on in the current sprint.

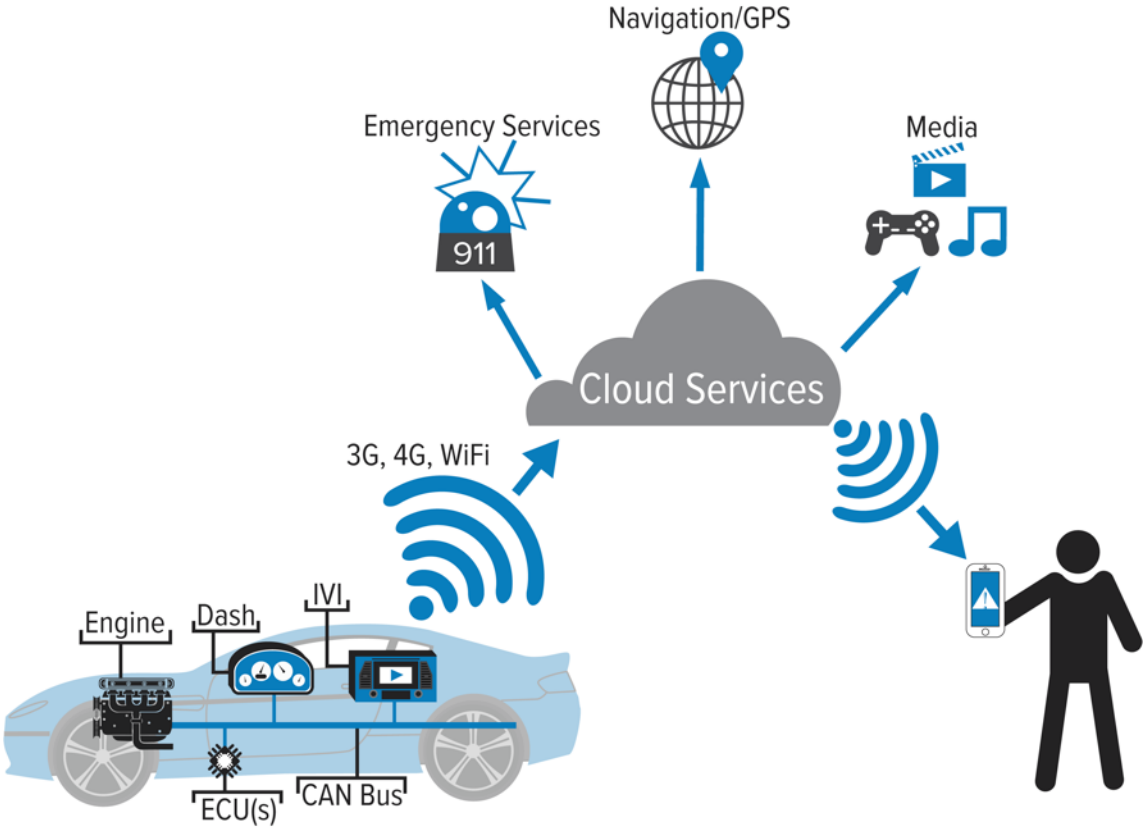


Fig. 3: Connected Car example system

Breaking it Down

Let's set a hypothetical example to better understand the benefits of API testing and service virtualization applied to testing connected embedded systems. Fig. 3 presents a system supporting several use-cases for a connected car; enabling a user to access the vehicle remotely - such as engine start/stop, remote a/c monitoring and operation, and diagnostics alerts (e.g. low oil warning); and enabling the vehicle to respond to and notify external systems – such as contacting emergency services in the case of an accident or autonomous driving and navigation responding to alerts from cloud-based services.

As we can see, there are several components included in the solution. For the simple case of diagnostic alerts, we have an oil level sensor attached to the engine (oil sump), and the sensor emits messages to the CAN bus

 © 2017 Parasoft Corporation | Perfecting Software

when a low level of oil is detected. Messages are then consumed by the dashboard to instantly notify the driver, and the multimedia system provides connectivity with cloud-based server components responsible for sending messages to the smartphone app. For simplicity, all aspects related to isolation of CAN from the connected layer are skipped.

With this example in mind, let's review how API testing and service virtualization can help in the verification of connected embedded systems.

Emulate dependent systems with service virtualization, which removes testing constraints and increases flexibility across complex test environments.

Assembling and maintaining a physical test environment to support complete testing of our example system is not going to be easy. How do we test the "emergency services" without crashing a car and having the cloud services call 911? How do we test to make sure that you cannot "turn off the engine" when the car is going 100 km/h? What happens to the phone app if the car sends an oil level of "-3"? And what about more complex scenarios? Using "remote start" to warm the car but requires being within 2 km of the vehicle but I cannot engage the transmission without unlocking the car from within 1m.

Creating a simple test bed for the basic isolated scenarios is expensive and complicated, and with these complicated interconnected systems, building a test environment to fully support end-to-end testing is impossible.

What we need to do is build a 'virtual test environment' that leverages Service Virtualization to provide multiple emulations of the different end-points. We can then reconfigure this test environment 'on demand' (see Fig. 4), enabling/disabling different virtual assets, and simulating different behavior of the end-points based on the requirements of the test being performed. This does not eliminate the need for a physical test bed, but it does enable the seamless execution of end-to-end scenarios during Continuous Integration (CI) workflows and enables testers to create and execute tests that validate the code being developed free of constraints imposed of external systems that they do not have control of.

Test at scale when scalable hardware is unavailable or impractical.

Once you have the core functionality tested, the next thing is making sure it works at scale – but how do you stress test something where it has dependencies on other backend systems and the test bed for those systems is limited in scale? What happens to the application if the payload is delivered over a slow 3G connection vs. 4G or Wi-Fi? What if the payload is broken/corrupted halfway through? What happens if the system is upgraded and suddenly services are responding in half the time they currently do?

This is another application of Service Virtualization. In addition to simulating the functional behavior, performance characteristics of the dependencies can also be simulated: long/short/random delays, load dependence profiles, and even segmented partial payloads. Going beyond what any physical test system can easily provide, a virtual 'test environment' provides complete flexibility and control to reconfigure system to validate both 'real world' and 'what-if' scenarios.

Leverage penetration testing techniques to expose security vulnerabilities and trace their impact directly to the code base.

The good news about leveraging standard internet protocols and open technology stacks is that there are many people that understand how the technologies work and therefore it is easier to design a scalable architecture and find the people that know how to build it. The bad news is that many people understand how the technologies work and therefore also know how to exploit them. This is a massive risk for IoT technology providers and especially the automotive industry, where security vulnerabilities can lead to potential losses of life. Again, we

can learn from the Enterprise IT community here, where they have been securing web applications for decades.

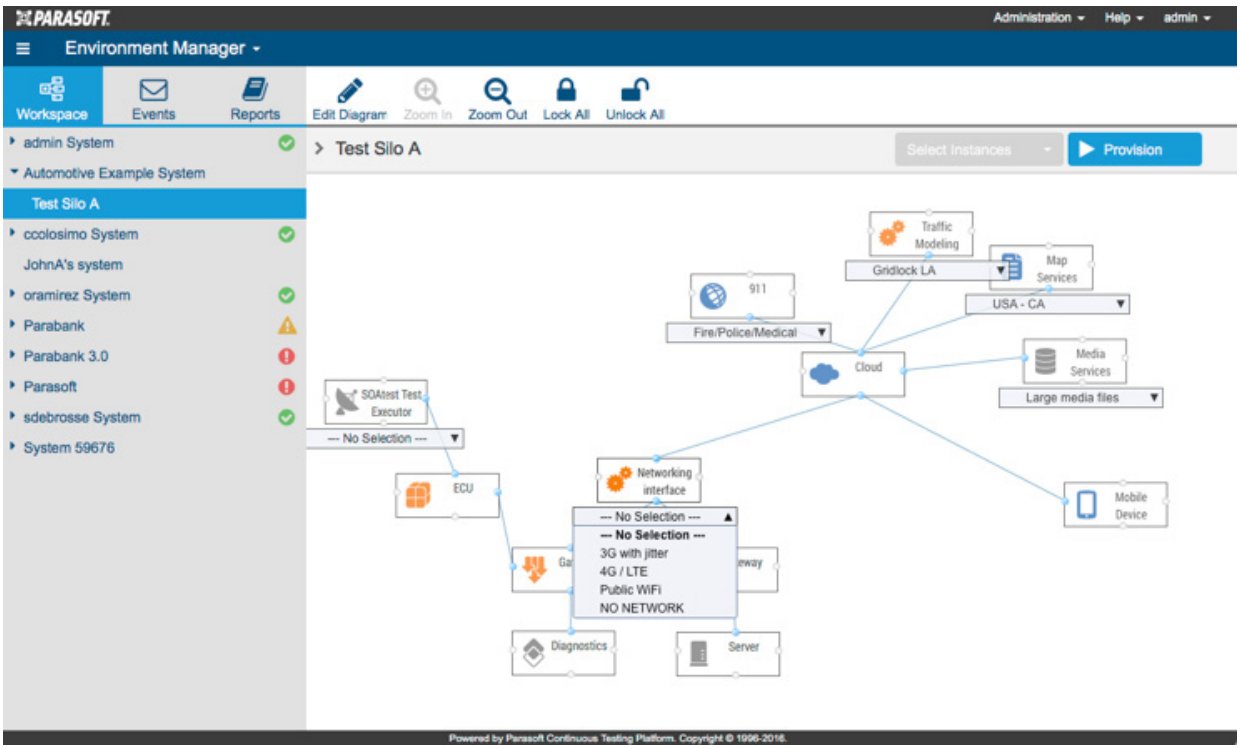


Fig. 4: Graphically managing a virtual test environment.

Just like locking the front door, leveraging the standards of HTTPS, SSL, and encryption is the critical first step. But securing the protocol alone is not enough – you also need to build security into the application itself to ensure that when the system is compromised, the application is not. Many times, unexpected input variables are the culprit, and hackers look at these corner cases to expose limitations and poor coding practices in the underlying code. Several industry standards have been developed to help identify and address these issues, such as CERT-C and MISRA C 2012 Amendment 1, but even with secure coding best practices in place you are not guaranteed to have covered all the scenarios.

A comprehensive security strategy combines both enforcement of best practices and penetration testing to perform automated attacks against an application to expose weaknesses. When penetration testing identifies a vulnerability, it is critical to be able to trace the individual test execution back to the underlying code and identify where the issue was exposed. This can then lead to improvements in the coding practices that can be enforced by automated code analysis much earlier in the development process.

Combine data from coverage analysis and message layer testing technologies

Functionalities that are implemented and deployed in multiple layers require special attention to test. The server side isn't just one system, but often a system of systems. The embedded side frequently also consists of several sub-modules. Automated API tests should be executed against each of the layers. The health of given functionalities can be assessed only after reviewing test results from each layer. But how do we know if each layer was adequately tested? In the end, the chain is only as strong as the weakest link.

The first step to answer this question requires building a reasonable structure of requirements. Top-level functional requirements that describe the behavior expected by the user need to be broken down into lower-level requirements describing API behavior in constituent layers. Once we have a definition of expected behavior at every layer, we can start building API tests for requirements verification. It is essential to create the association between requirements and the test cases. Without tests-to-requirements traceability, it will be almost impos-

sible to assess the impact of one failing test on the entire functionality. Fig. 5 illustrates an exemplary report showing requirements tests-coverage:

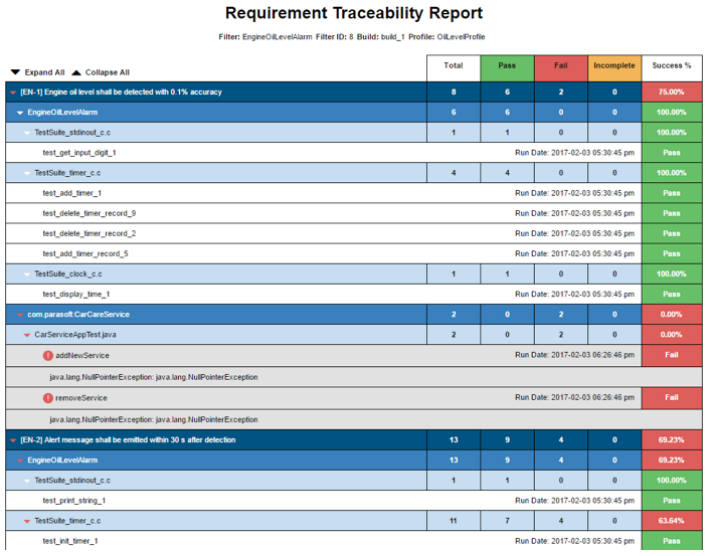
With a reporting system providing this level of visibility into the project, the team can focus on important things and save precious time. Those in charge of the system quality do not need to review pages of test execution results, trying to understand what a given result means to an entire system – they can get a quick overview of the solution health state by looking at the testing process from the requirements perspective and limit the reaction only to hot spots where something went wrong.

Test results alone are not enough to assess the state of a system. Even if we have a complete test-coverage of requirements, we still don't know how thorough our testing process is. Did we really examine what was implemented? Is there some corner case for which we do not have a verification? This question can be answered only with code coverage tools. In the ideal world, you would have detailed low-level requirements describing every possible behavior of the system, including nonfunctional requirements, error handling requirements, and unexpected input requirements. Full test-coverage for such a specification would imply 100% source coverage, but this is almost never the case. It is common to lack low-level nonfunctional requirements, and code coverage tools are the only option to provide the feedback necessary for the testing process (and a de facto must-have for any organization trying to deliver quality products on time). For the sake of simplicity, we'll skip the discussion of which coverage metric is superior because it is a subject for a separate discussion.

Implementing code coverage measurements in combination with API testing for all services and modules involved in a given functionality helps us estimate where we need to improve the quality of the testing process, and informs us about the business risk. If we do not have an adequate level of testing for even a single subsystem of tested functionality, the entire solution is at risk. Code coverage tools provide this information, where testing process is not sufficient, and links from tests to requirements help us understand the impact of insufficient testing.

Implementing code coverage measurements for connected embedded system poses a challenge because of the technology mix which is usually applied in this kind of solution. A testing framework should support code coverage measurements for all languages that are used, such as Java, C#, C/C++. In addition, the testing framework should support associating test cases with code coverage results generated when the given test was executed. In the example that was set at the beginning of the section, we can distinguish at least four different modules, implemented in different technologies:

- Oil level sensor: microcontroller-based system, programmed in C, communicating to the CAN bus
- Infotainment system: implemented in C++, communicating via 3G/4G/Wi-Fi
- Cloud-based server application: implemented with C#, receiving notifications from the car, associating them with user profiles and sending notifications
- Mobile phone app: implemented in Java, receiving messages from the cloud



Requirement Traceability Report

Filter: EngineOilLevelAlarm Filter ID: 8 Build: build_1 Profile: OilLevelProfile

Expand All ▲ Collapse All	Total	Pass	Fail	Incomplete	Success %
[EN-1] Engine oil level shall be detected with 0.1% accuracy	8	6	2	0	75.00%
EngineOilLevelAlarm	6	6	0	0	100.00%
TestSuite_stdinout_c.c	1	1	0	0	100.00%
test_get_input_digit_1			Run Date: 2017-02-03 05:30:45 pm		Pass
TestSuite_timer_c.c	4	4	0	0	100.00%
test_add_timer_1			Run Date: 2017-02-03 05:30:45 pm		Pass
test_delete_timer_record_9			Run Date: 2017-02-03 05:30:45 pm		Pass
test_delete_timer_record_2			Run Date: 2017-02-03 05:30:45 pm		Pass
test_add_timer_record_5			Run Date: 2017-02-03 05:30:45 pm		Pass
TestSuite_clock_c.c	1	1	0	0	100.00%
test_display_time_1			Run Date: 2017-02-03 05:30:45 pm		Pass
com.parasoft.CarCareService	2	0	2	0	0.00%
CarServiceAppTest.java	2	0	2	0	0.00%
addNewService			Run Date: 2017-02-03 06:26:46 pm		Fail
java.lang.NullPointerException: java.lang.NullPointerException					
removeService			Run Date: 2017-02-03 06:26:46 pm		Fail
java.lang.NullPointerException: java.lang.NullPointerException					
[EN-2] Alert message shall be emitted within 30 s after detection	13	9	4	0	69.23%
EngineOilLevelAlarm	13	9	4	0	69.23%
TestSuite_stdinout_c.c	1	1	0	0	100.00%
test_print_string_1			Run Date: 2017-02-03 05:30:45 pm		Pass
TestSuite_timer_c.c	11	7	4	0	63.64%
test_init_timer_1			Run Date: 2017-02-03 05:30:45 pm		Pass

Fig. 5: Example Requirement Test Traceability Report

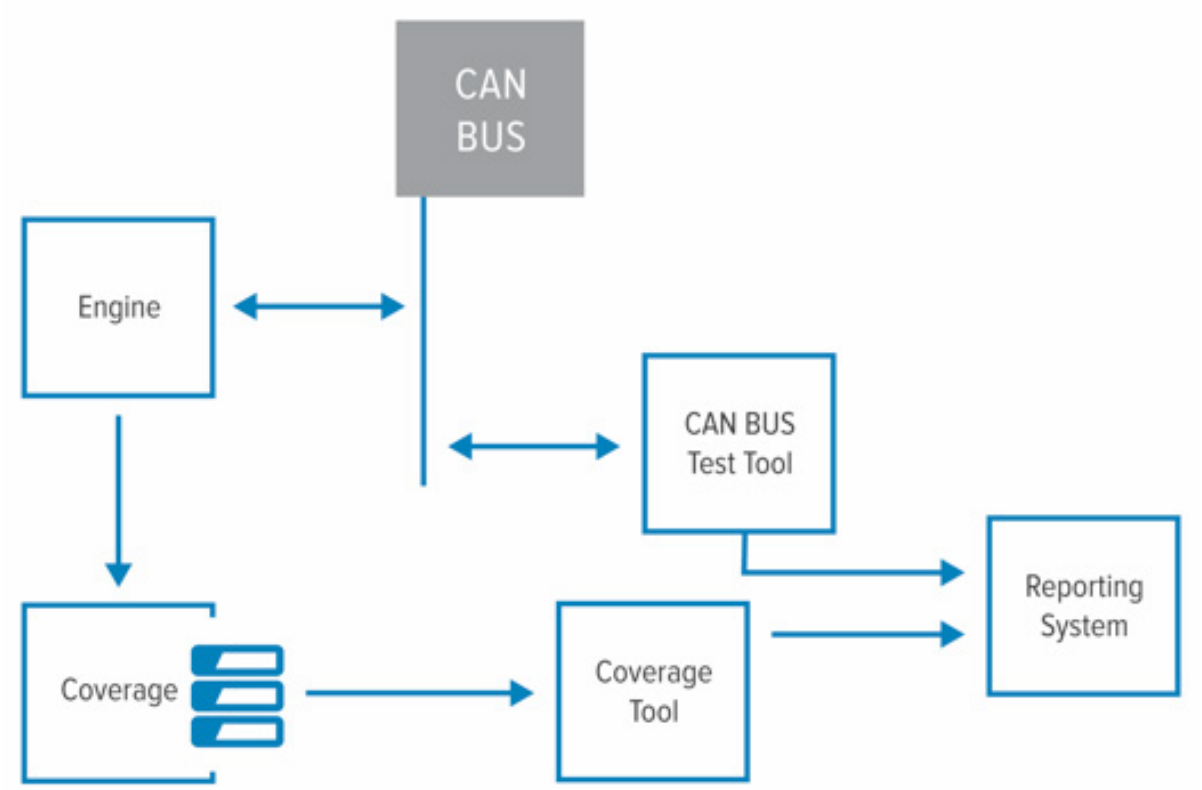


Fig. 6: Testing of on vehicle services

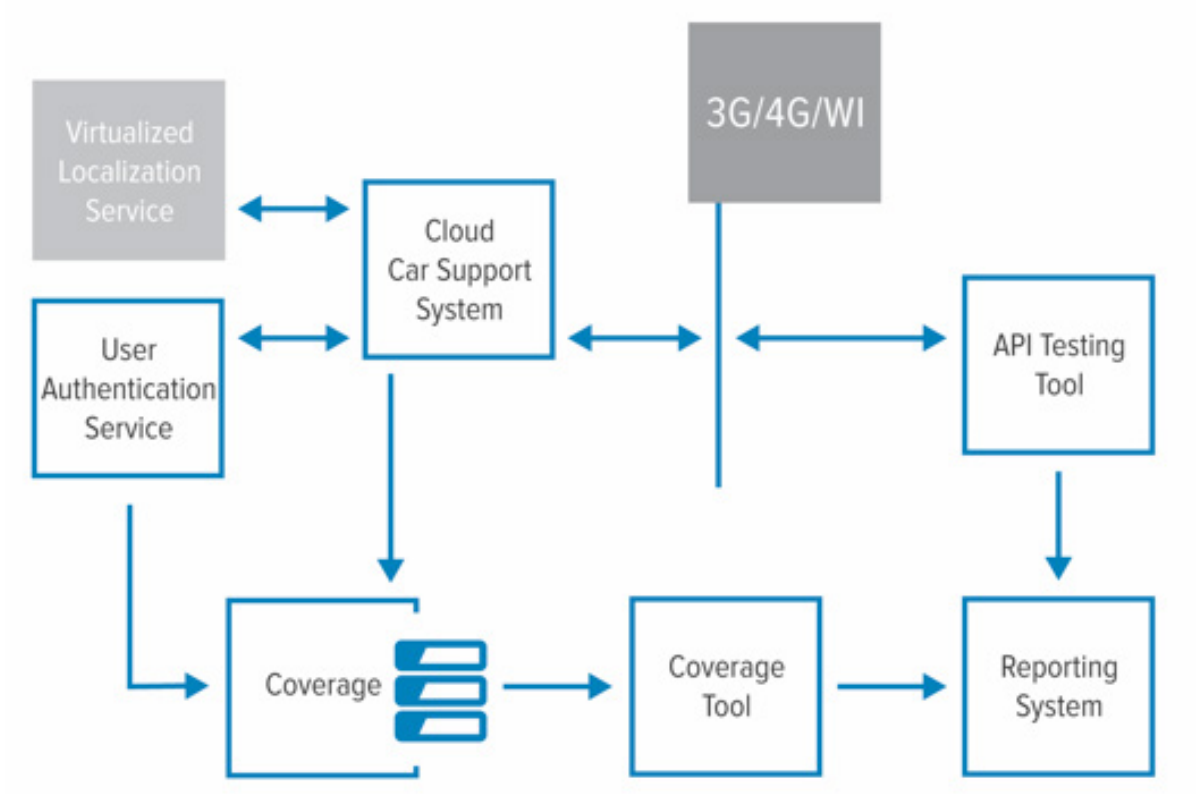


Fig. 7: Testing of cloud-based service

Test reports collected from the test setups as presented above should provide important pieces of information about the code coverage generated by each individual test case, as well as any arbitrary selection of test cases and how they correspond to requirements.

 © 2017 Parasoft Corporation | Perfecting Software

Data can be presented in a variety of forms, the most usable of which are interactive reports (Fig. 8 and Fig. 9) that start with a high level overview of data, and allows you to drill down for investigating specific hot points. It is critical, however, that these reports enable the user to navigate test results from throughout the testing pyramid.

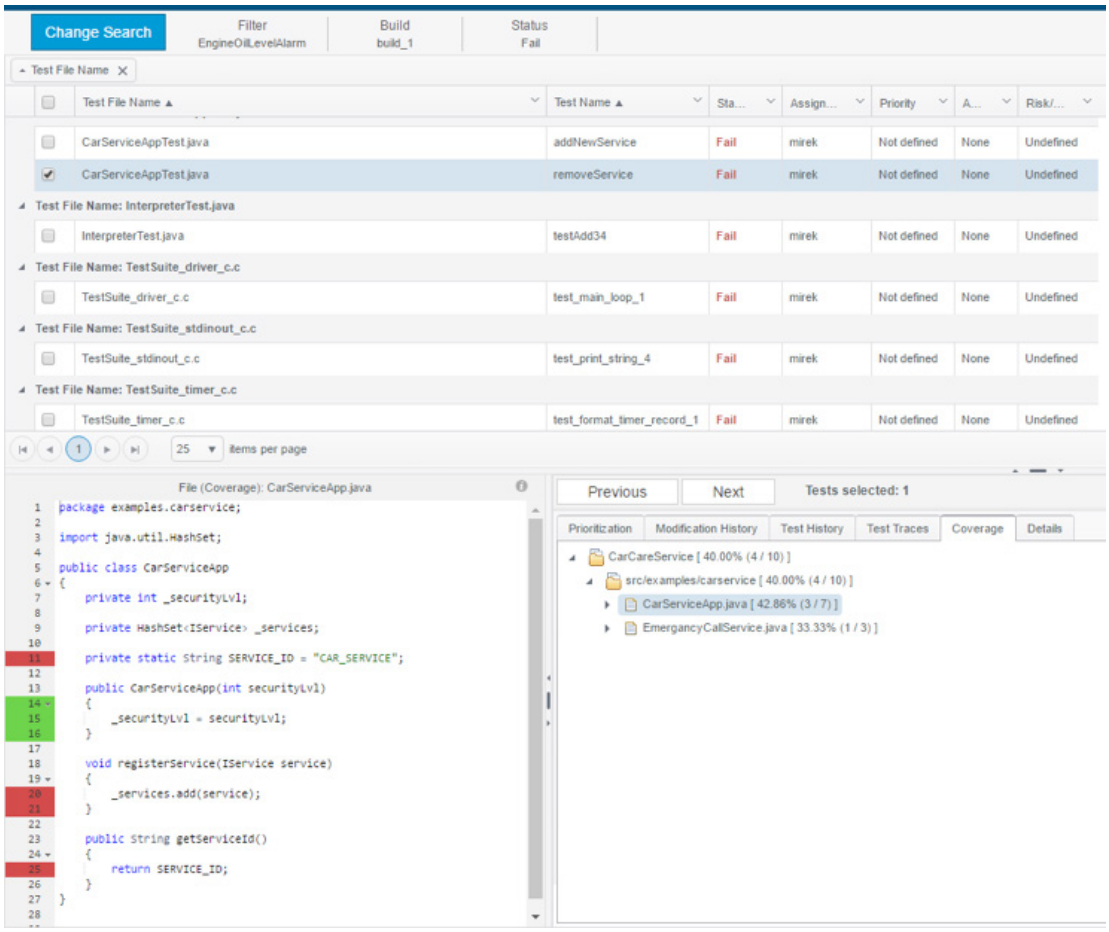


Fig. 8: Interactive test coverage navigation

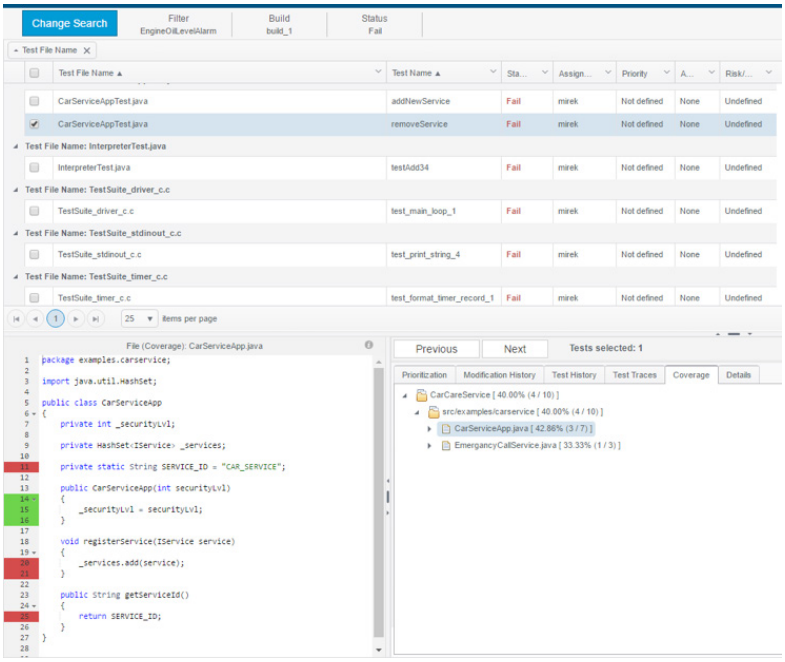


Fig. 9: Interactive test navigation

Summary

The drive for delivering market-leading innovation is the key differentiator in both enterprise IT and embedded software industries. Organizations are looking to service-oriented architectures and agile development practices to reduce time to market and deliver a compelling user experience.

Accelerated schedules and system complexity makes testing of these deliverables difficult and time consuming. In this paper, we discussed the following techniques that can be leverage to address these challenges:

- **Automated API Testing:** Moving up the testing pyramid enables a cost-effective way to validate components individually and as part of the system as a whole.
- **Service Virtualization:** Using service virtualization eliminates system constraints and enables automated tests to run continuously as part of CI workflows.
- **Automated Penetration Testing:** Expose vulnerabilities that can be traced back to the underlying code – where the exploit can be identified and used to improve coding best practices to prevent similar future exploits.
- **Test, Requirement and Code Traceability:** Capturing coverage for individual tests throughout the testing pyramid and correlating back to original requirements enables organizations to choose the most cost-effective testing techniques.

When used in conjunction with each other, these techniques provide organizations the ability to accelerate and improve accuracy of market delivery and do so in a way that is secure, scalable, and reliable.

About Parasoft

Parasoft helps organizations perfect today's highly-connected applications by automating time-consuming testing tasks and providing management with intelligent analytics necessary to focus on what matters. Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software, by integrating static and runtime analysis; unit, functional, and API testing; and service virtualization. With developer testing tools, manager reporting/analytics, and executive dashboarding, Parasoft supports software organizations with the innovative tools they need to successfully develop and deploy applications in the embedded, enterprise, and IoT markets, all while enabling today's most strategic development initiatives — agile, continuous testing, DevOps, and security.

PARASOFT®

Perfecting Software

Sales: 1-888-305-0041
Int'l: +1-626-256-3680

All rights reserved. Parasoft and all Parasoft products and services listed within are trademarks or registered trademarks of Parasoft Corporation. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.