Towards Zero Defect Software

Automating Verification and Validation

For testing embedded code, which is becoming more complex and is increasingly being deployed in critical systems, automated testing is the only way forward. The two main technologies used in this solution framework are static analysis and model checking. In embedded systems, automated V&V technologies are also being applied to check properties other than coding correctness, such as estimation of resource consumption, worst-case execution time, battery discharge, and heat emission. Some of these problems are open. AI, app culture, and cloud pose new problems. Future testing and verification systems will have to be intelligent, autonomous gatekeepers working round-the-clock. Perpetual testing will have to be done post-release as well. For testing future intelligent systems, new techniques that combine analytical and fuzzing techniques will have to be used to generate stronger, more comprehensive test cases.

IN BRIEF

Software bugs cost the global economy USD 1.7 trillion in 2017.¹

The total number of security failures studied constitute 26% of the bugs. The cost increased by several multiples, with retail and consumer tech being the most affected. A key cause behind this increase is the exponential growth in the size and complexity of software programs. The advent of machine learning (ML) only adds to this complexity, making it much harder to mitigate faults and avoid failures.

The end of manual verification

Today, software programs are not only powering traditional business systems, but also smartphones, drones, cars, airplanes, healthcare devices, and even nuclear reactors. In each of these systems, the number of programs usually comprise a few million lines of code to a couple of billion. For example, modern high-end cars contain more than 100 million lines of code. While this signifies growth in the scope of business, it also brings into question the risks, safety, security, and privacy concerns associated with the same. For instance, in the automotive sector, the number and percentage of vehicle recalls due to software glitches are increasing year over year. In 2018 alone, Fiat Chrysler recalled more than five million cars due to a software


Reimagining Research 2018
Towards Zero Defect Software

Fact File

TCS Research: Verification and Validation
Outcomes: TCS Embedded Code Analyzer, Expressive Decision Tables, and Loop Abstraction for Bounded Model Checking
Principal Investigator: R Venky
Academic Partners: Professors at Oxford University; Technical University of Munich; Chennai Mathematical Institute; Indian Institute of Technology Bombay
Techniques Used: Static Analysis, Model Checking, Requirement Analysis
Industries Benefited: All industries that use embedded software
Patents: 14
Papers: 70+

glitch that could prevent drivers from deactivating the cruise control in order to slow down their cars. Studies have shown time and again that these defects could be up to a 100 times less expensive if they were fixed during the software development life cycle (SDLC). Traditionally, software testing and reviews were done manually. As a result, they were unable to scale in terms of size and complexity of modern software systems. Manual testing was time-consuming and effort-intensive. Moreover, limited availability of skilled reviewers and testers was an additional challenge. Standards, such as ISO 26262, mandate formal verification and validation of all safety-critical systems. In this context, automated verification and validation (V&V) is a promising technological innovation that has the potential to resolve challenges posed by modern software programs.

Two automation techniques

Currently, state-of-the-art automated V&V solutions focus on the functional correctness of software specification and code. The two main technologies used in these solutions are:

1. Static analysis: This refers to a class of techniques that analyze a software program without executing it. These techniques usually rely on over-approximation of program behavior to the scale of multimillion lines of code. As over-approximations are a source of imprecision, static analysis tools tend to report false positives. Potential errors reported by this model of analysis have to be manually reviewed with a lot of effort going into eliminating the false positives.

2. Model checking: This refers to a class of techniques that analyze every path in the program without executing it. The model generates precise answers about the program behavior. However, this precision comes at the cost of scalability. Model checking tools scale to a few thousand...
lines of code. Subsequently, these sophisticated tools can be used to verify only the most critical parts of the program.

Software analysis tools based on these methodologies offer the following abilities with a varying degree of scalability and precision:

- **Defect detection:** Each type of defect that requires identification has to be specified as a property of the code on the analysis tool. This is followed by checks on the program to verify if such defects exist in the code.

- **Test case generation:** With a coverage criterion, these tools can generate test cases that cover the program according to the specified criterion.

- **Report generation:** These tools can be programmed to generate reports of regulatory compliance in line with standards such as ISO 26262.

Several advances have been made toward improving the precision and scalability of these aspects, leading to an increase in the adoption of these technologies. Many highly scalable and fully automated commercial and academic tools are available to support and simplify the implementation process. A number of automotive, avionics, software, and hardware companies also have in-house V&V research and development (R&D) teams.

### Problems to be solved

In the context of embedded systems, these technologies are also being applied to check software programs for properties not limited to coding correctness, such as estimation of resource consumption, worst-case execution time, battery discharge, and heat emission. However, not all problems can be resolved using V&V techniques. There are additional challenges that need to be addressed in order to leverage the full potential of these tools.

- **Certification:** While these tools are effective in the identification of defects, they cannot be used to certify whether a given software program is scalable or precise.

- **Usability in practice:** Several cases call for manual inspection in addition to tool reporting and analysis. A lot of ongoing research focuses on minimizing this manual effort.

- **Defect fixing:** Once these tools report the errors, fixing them requires manual effort. Research indicates that performing a causal analysis helps developers speedily resolve such defects.

Moreover, with the advent of autonomous software systems, such as those in self-driving cars, determining what constitutes an erroneous behavior and reproducing the same is another major challenge.

### Challenges ahead

In the near future, software systems will have distinct attributes that include:

- **Artificial intelligence (AI) or ML capabilities**

- **Continual evolution of systems through downloadable applications developed by an ecosystem of engineers**

- **Cloud hosting of software or upgrade over the air—making it easy to distribute patches with bug fixes or implement new features**
Towards Zero Defect Software

devlopment and implementation of a verification engine. This will act as a gatekeeper for quickly certifying new features. To enable rapid enhancements, these systems will need to be designed as platforms on which engineers can develop new functionalities. These platforms will offer developers well-defined interfaces in the form of APIs or services that impose certain protocols needed to be followed by their users. For instance, a platform that offers audio-video support may require a video-streaming application that first checks the availability of both audio and video before streaming, and then explicitly releases both once the streaming starts.

Conformity to such protocols can be checked using a code analyzer customized to encode those protocols that need verification. Operating system vendors can employ customized code analyzers to certify device drivers. OEMs and other systems developers can employ the same tools to validate vendor software. In addition to protocol violations, other critical defects to guard against are system crashes and security vulnerabilities. Automatic test generators that implement concolic testing and evolutionary fuzz testing can help locate most of these defects. A gatekeeper software, as shown in Figure 1, includes a code analyzer

Verification gatekeepers

The sheer complexity of these systems and rising demand from users for software enhancements at a very rapid pace will require the development and implementation of a verification engine. This will act as a gatekeeper for quickly certifying new features. To enable rapid enhancements, these systems will need to be designed as platforms on which engineers can develop new functionalities. These platforms will offer developers well-defined interfaces in the form of APIs or services that impose certain protocols needed to be followed by their users. For instance, a platform that offers audio-video support may require a video-streaming application that first checks the availability of both audio and video before streaming, and then explicitly releases both once the streaming starts.

Conformity to such protocols can be checked using a code analyzer customized to encode those protocols that need verification. Operating system vendors can employ customized code analyzers to certify device drivers. OEMs and other systems developers can employ the same tools to validate vendor software. In addition to protocol violations, other critical defects to guard against are system crashes and security vulnerabilities. Automatic test generators that implement concolic testing and evolutionary fuzz testing can help locate most of these defects. A gatekeeper software, as shown in Figure 1, includes a code analyzer

Verification gatekeepers

The sheer complexity of these systems and rising demand from users for software enhancements at a very rapid pace will require the development and implementation of a verification engine. This will act as a gatekeeper for quickly certifying new features. To enable rapid enhancements, these systems will need to be designed as platforms on which engineers can develop new functionalities. These platforms will offer developers well-defined interfaces in the form of APIs or services that impose certain protocols needed to be followed by their users. For instance, a platform that offers audio-video support may require a video-streaming application that first checks the availability of both audio and video before streaming, and then explicitly releases both once the streaming starts.

Conformity to such protocols can be checked using a code analyzer customized to encode those protocols that need verification. Operating system vendors can employ customized code analyzers to certify device drivers. OEMs and other systems developers can employ the same tools to validate vendor software. In addition to protocol violations, other critical defects to guard against are system crashes and security vulnerabilities. Automatic test generators that implement concolic testing and evolutionary fuzz testing can help locate most of these defects. A gatekeeper software, as shown in Figure 1, includes a code analyzer

Figure 1: Gatekeeper Architecture

The only way to check whether an autonomous vehicle is detecting obstacles correctly is to test it against those obstacles, as there is no alternate specification of correctness that can be used as a standard reference.
and an intelligent test generator that can minimize risks related to frequent upgrade releases.

**Perpetual testing**

Software systems need to be tested for functional correctness besides standard properties such as security vulnerabilities and crashes. Functional correctness is tested by writing test cases and executing them on the system. Although test-case execution is automated by harnesses, writing test cases is largely a manual process. Writing effective test cases requires high domain expertise, and is effort-intensive. This process can be automated by writing the requirements using a high-level specification language to generate test cases automatically. Automation leads to the systematic generation of test cases.

Automation of test case generation has a favorable consequence—testing post release. Future software systems will either be hosted on the cloud or updated over the air. This simplifies the process of deploying patches that fix bugs post-release and before users encounter them. Future DevOps processes will include automated perpetual testing even after a release, as shown in Figure 2. Bugs found during the post-release testing phase will be sent to the maintenance team to be fixed and patched over the air. Not only will this reduce the cost of fixing these bugs at a later stage, but also contribute towards upholding the overall brand value of the organization concerned.

**Testing machine learning algorithms: the way forward**

Modern systems will implement ML algorithms to enhance user experience. Examples of such systems include image-processing algorithms in cameras embedded in autonomous vehicles, chess playing software programs, and audio-processing software systems with voice control interfaces, among others. Testing these algorithms will need remarkably different techniques, as traditional definitions of correctness and coverage are not applicable.

They need to be tested in terms of the boundaries of their classification. For instance, an image-processing algorithm that is supposed to detect obstacles can be tested for its boundaries by taking an obstacle that it classifies correctly, and changing it to determine the boundaries of alterations up to which it continues to be classified as an obstacle. Techniques similar to fuzz testing used for the detection of security

---

**Figure 2: Perpetual Testing**
Towards Zero Defect Software

tools will increasingly play the role of a software gatekeeper even after a product has been shipped and implemented. They will implement a combination of analytical and fuzzing techniques to generate newer test cases that will examine the boundaries of future intelligent systems.

vulnerabilities and analytical techniques that determine boundaries of a given neural network will be the way forward. These new testing algorithms will have to be implemented in the perpetual test engines so that bugs can be found even after a product is shipped. Therefore, future testing

R. Venkatesh

R. Venkatesh is a Chief Scientist at TCS Research and heads its Verification and Validation Program. He has been with TCS for more than 25 years, working primarily in the areas of software development, formal methods, and verification. During this tenure, he has led several tool development projects, including TCS ECA, a static analysis tool that is sold commercially by the company. Other tools include MasterCraft and more recently a formal specification notation called EDT.

Ravindra Metta

Ravindra Metta is a Technical Architect and a Distinguished Engineer at TCS Research and Innovation. He holds two Masters Degrees: one in Mathematics and one in Computer Science. He has over 20 years of experience in the research and development of software engineering tools based on formal methods.