The Six Core Aspects of an Ultimate CI/CD Machine for Automotive Safety-critical Software

Ingo Nickles

PCT
Vector Informatik GmbH
47906 Kempen, Germany
ingo.nickles@vector.com

Abstract: In software development with "Continuous Integration", code changes are continuously, almost immediately, inserted into a project and automatically checked. The aim of the check is to ensure "Continuous Deployment", i.e. to keep the code base in a state that would in principle allow the creation of a customer release at any time.

As many aspects as possible in a continuous integration pipeline with continuous deployment should be automated: Integrating the code change into the existing code base, static and dynamic checking of source code and test cases through to continuous monitoring and the creation of corresponding reports. Depending on the results of the check further steps can also be taken automatically, such as creating error tickets.

To ensure that the continuous integration of code changes does not unnecessarily affect the day-to-day work of the development team, all results must be made available quickly and reliably. How a corresponding work environment should theoretically be set up is shown below. Finally, a practical implementation is described using concrete tools as an example.

Keywords: Change Based Testing; Test Automation; Regression Testing; Continuous Integration; Continuous Deployment, DevOps

1 INTRODUCTION

The ever-increasing code bases of modern embedded software projects pose immense difficulties for many development teams. Integrating new code has never been easy. However, while in the past it used to be "sufficient" to create the code and to integrate all code changes at the end of the development and then test them, this is almost impossible with project sizes of many millions of lines of code.

The increased code size includes increasing team sizes which raises additional difficulties in terms of collaboration. In addition to the increasing size and complexity, the development process is also under pressure from ambitious deadlines, which in the worst case leads to a neglect of software testing activities and thus to reduced software quality and increasing software erosion.

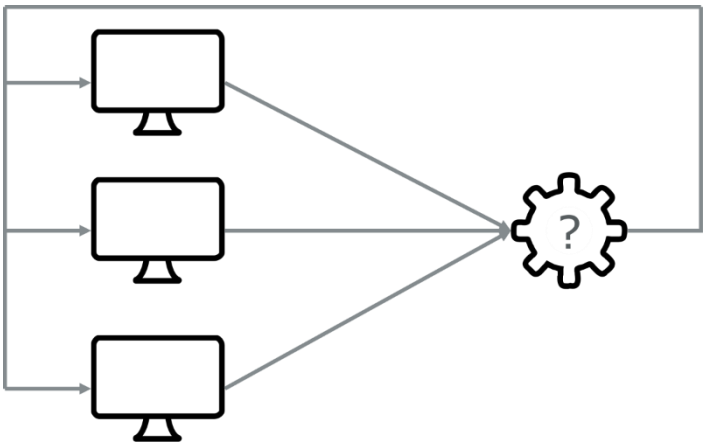

Figure 1: Several developers change the same code base

To circumvent this problem, many teams opt for continuous integration with continuous testing and continuous deployment: changes are continuously inserted into a centrally managed repository and checked automatically. Continuous monitoring allows the early detection of software erosion and makes it possible to take countermeasures in good time.

2 CONTINUOUS INTEGRATION

2.1 Source Code Management

The basis of the CI/CT/CD (or CITD for short) process is the source code management (SCM), in which all files are stored centrally and monitored. "All files" obviously means the actual source code files; however, development-related documents, functional requirements, design documents, test or error data can also be stored in SCM. Centralized filing of files is essential,

particularly in the case of cross-team development work. However, sharing access to the same files can lead to problems, as shown in Figure 1: When multiple people are modifying a file at the same time, whose changes win? And how can an editor who already has a file open be notified of a change in the central file?

This problem can be avoided by exclusive access to central files. This means that before a file can be changed, it can first be reserved and only then changed.

However, this approach leads to problems in large projects with many teams: For example, there are often dependencies between different files in the source code. A change in one header file may cause changes in many other files that otherwise would not compile. In such a scenario, a developer must reserve a larger number of files at the same time and only then can he/she make his/her changes. In turn, reserving many files hinders the work of other team members who must wait for the files to be available for their own development activities.

From the author's point of view, this way of working has therefore not gained acceptance. Instead, modern SCM programs allow the files to be modified without prior reservation. It is checked whether changes to be saved collide with the changes of another team member, which can then require a complex merging of both versions at the time of central saving. Despite the sometimes-considerable additional work involved in processing collisions, this method of working, which does not require reservations, is now state of the art.

2.2 Continuous Testing and Continuous Deployment

The continuous saving of changes, especially of the source code at a central location, e.g. with the help of an SCM program, is therefore referred to as "Continuous Integration" or CI. For "Continuous Deployment", i.e. for a workflow in which the centrally stored software status can be delivered to a customer at any time in the form of a release (or is indeed deployed; e.g. in the form of an immediate update on a server), continuous quality management is also required, but at least "continuous testing", i.e. the continuous execution of all (necessary) test cases.

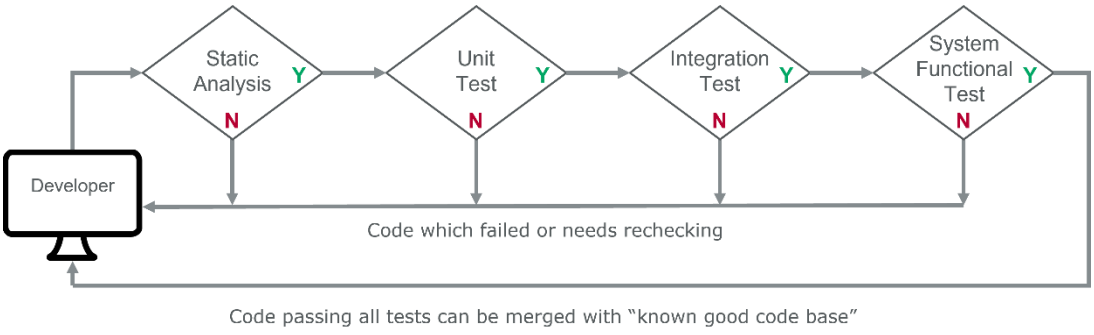

Figure 2: Exemplary continuous testing process

Executing test cases is often seen as a downstream step when changing source code. As can be seen in Fig. 2: A code change can only be saved/committed centrally if all quality gates pass, e.g. all test cases have been successfully executed and passed.

But that is too short-sighted: a test case should check a functional requirement for the code. In the best case, a test case (at least in the design) is created before the code change, e.g. in test-driven development (TDD), but at the latest with the code change to check the correctness of the changed code.

If the code changes due to a new or changed functional requirement, then every associated test case should change at the same time and/or new test cases should be created at the same time. Code changes to bug fixes should be secured using additional test cases. But that means:

- Almost every source code change also implies a test case change.
- Test cases should also be managed in the SCM and stored in parallel with the appropriate code. I.e. it is easiest if source code and test cases share the same repository.
- In the case of central changes, both the source code and the test cases should be checked automatically.

The management of the test cases in the SCM, i.e. a central storage of the test data for automated execution, is a decisive factor for cross-team access to the test data and test results. It helps when introducing a "continuous testing" workflow. Any involved test tools must be able to re-create test setup and re-execute test cases with a convenient set of files that are stored in SCM. Due to collisions when multiple users access the same data all files must be saved in human-readable form. And with just these files the test tools must be able to

- automatically generate test environments
- execute test cases
- generate corresponding reports

A restrictive commit workflow as shown in Figure 2 refuses a commit if any quality-check fails. This workflow causes annoyances and slows the development process down. On the other hand, we want to maintain a high-quality code base that is continuously ready for deployment. To avoid issues with the central repository but still allow developers to commit changes without passing all quality gates we can use feature branches as shown in Figure 3

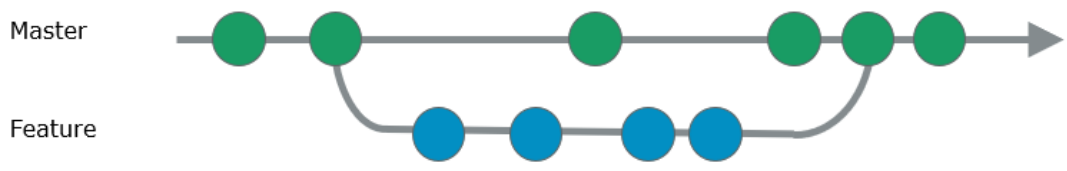

Figure 3: SCM with feature-branch

Feature branches allow all development that is related to a specific feature (or bugfix) go to a dedicated branch in the central repository. Neither each commit nor the artifacts in the feature branch themselves need to always pass all quality gates.

Only then when the feature branch is merged back into the master branch (usually called merge-request or pull-request) the resulting repository needs to pass all checks. This will usually be done in two steps:

- First Step:
  At the end of the feature development run all test cases and verify that all quality gates are passed on the feature branch. Adapt source code and test cases as needed.

- Second Step:
  When including the feature branch into the Master branch (which will very likely require certain files to be merged) run all resulting test cases and verify that all quality gates are passed on the unified master-feature repository

2.3 Different CI requirements for developers and testers

To get to the point: A developer expects quick results, since CI is only a means to an end. Testing and collision resolution is commonly perceived as an impediment to a rapid development process. A tester, on the other hand, attaches more importance to the completeness of the testing activity. Even if time aspects must not be neglected in the complete test run, longer test runtimes are more acceptable for the test team than for the development team member. But what does this mean for CI? It obviously makes sense to take a more differentiated look at the structure of an ultimate CI/CD machine.

First, there are basic requirements that should always be met and most of which have already been discussed:

- Central storage of source code and test data in SCM

- Automatic executability of all test cases

- Automatic analysis of the results including reporting

- Automatic monitoring

- Everyone in the team should be able to "easily" execute all test cases (e.g. by pressing a button)

CI considerations are often limited to precisely these aspects, but this does not consider the different requirements of the different team members. The problem with this is that in most cases indeed all test cases are integrated into a CI workflow.

But running all test cases can take a lot of time considering what "all test cases" means:

- Unit, integration and system test cases

- Testing of all meaningful variants of the code ("conditional code")

- Testing of all target systems used

All 3 points of this list are multipliers when determining the runtime of all test cases (TC):

$$\sum(AlleTF) = \left(\sum(UnitTF) + \sum(IntTF) + \sum(SysTF)\right) * \#Varianten * \#Zielsystem$$

The sum of all test case execution times is equal to the sum of all runtimes of all dynamic test cases (Unit-, Integration and System-Test Cases) multiplied by the number of source code variants and multiplied by the number of target variants.

The runtime of all test cases can often take several weeks. Fortunately, modern CI systems allow test case execution to be run in parallel, which means that acceptable runtimes of a maximum of 2 days can usually be achieved, which of course is not suitable for a continuous execution.

Nevertheless, we should keep hold of the complete test run of all test cases as the first use case of a CI job. This complete test run ensures that the code works in all variants and can be used as test proof in the environment of developing safety-critical software. The runtime of up to 2 days is generally acceptable because the complete test run is usually done only once before deployment or when a customer release is delivered. It is also unlikely that any issues are to be expected when executing the complete test run due to the continuous testing described below.

Even if the realization of the complete test run in a CI tool environment is not a classic "continuous integration" use case, its implementation ensures that everyone in the team can run all test cases automatically at the push of a button. The actual use for a continuous integration of code and test cases is then only an easy-to-implement special case of the complete test run with a reduced scope. Since the exact form of the reduction for the implementation of a CI job for the test team depends on many other factors, we will not go into detail about them, but instead turn to the other extreme: the minimal test run, initiated by a developer during a code change for rapid validation.

As already mentioned, the shortness of the test runtimes paired with the greatest possible certainty that the centrally changed code is correct is the balancing act that needs to be realized when implementing CI. Since not only source code but also test data changes, the developer should have both the complete code base and the complete test environment available locally in order to be able to adapt test cases of all kinds or at least to notice their failure before the change ends up in SCM.

Simulation or virtualization should be used whenever possible to reduce runtimes and keep the amount of hardware required as low as possible. In most cases the results of the test case execution on the simulator will not differ from the execution on the real hardware. For rapid validation it should also usually be sufficient to test the code changes in just one variant.

Finally, the test case execution runtimes can be reduced through change-based testing: Only those test cases that are affected by a code change (or a test case change) are executed.

If the test runtimes are too long despite reduction to one variant, simulation, parallelization of execution and change-based testing, the test runtime can be further reduced by increasing the resources and/or by reducing to a prioritized selection of test cases. In the context of this article, I do not want to go into any further detail about strategies for prioritizing test cases. However, it is not recommended, for example, to only run unit tests and to ignore test cases from higher test levels. Each test level finds different types of issues, so it is important to ensure that all test levels are continuously represented.

3 CURSE AND BLESSING OF AUTOMATICALLY GENERATED TEST CASES

Of course, it would be nice if the source code could be checked using automatically generated test cases. Unfortunately, this can only be achieved in the rarest of cases, at least for functional requirements for the code, since the functional requirements are usually not specified in machine-readable form.

Other code requirements, e.g. robustness to extreme values (e.g. minima, maxima, null-pointer) can be checked in a useful and automated way. For example, unit test cases can be generated automatically through value range analyses of the underlying data types, or the system can be challenged with limit values based on a standard protocol. A proper reaction to male-formed messages or out-of-bounce elements is most likely also defined in the standard protocol, which can be checked automatically accordingly. At the unit test level, a non-crash of the unit test task would be considered a success.

In addition to these meaningful robustness tests, a path analysis of the source code for automatic test case generation with high path coverage can also be carried out. Since these test cases are derived from the coding and not from the functional requirements, they are only recommended in rare cases.

Figure 4: Tools: It depends on what you do with them

4 SUMMARY AND TOOL EXAMPLES

Finally, the core aspects of building an ultimate CI/CD machine should be outlined here once again. I am also often asked that while these ideas sound great in theory, they are difficult to implement in practice. In the following, I will therefore also give examples of tools that can be helpful in implementing the theory. I would like to point out that my choice is of course influenced by what my employer offers and my very personal experiences. But there are plenty more fish in the sea.

6 key aspects of an ultimate CI/CD machine:

- A central repository that supports the integration process through automated jobs.
  Example: git/github
  Jenkins

- Allow developers to test whenever they need it.
  Example: Integration with Jenkins
  Emulation by QEMU
  Test cases in VectorCAST, CANoe4SW, CANoe

- Provide an overview of the completeness of the tests.
  Example: Code coverage by VectorCAST/QA
  Check metrics and KPI with Squore
  Requirement traceability

- Automatically generate test cases where it makes sense.
  Example: Baselining test cases with VectorCAST/C/C++
  Checking the diagnostic protocol with CANoe.DiVa

- A parallelization and scaling of the test architecture in order to achieve the fastest possible build/execute time
  Example: Pipelines in Jenkins, possibly workers (executors) on different PCs
  Cloud services with Docker containers

- A way to perform the minimum number of tests required by a change to source code
  Example: Change-based testing with VectorCAST