embeddedworld2015
Exhibition&Conference
... it´s a smarter world

Continuous Integration for Embedded Systems using Simulation

Jakob Engblom
Wind River
Kista, Sweden
jakob.engblom@windriver.com

***Abstract*— Continuous integration (CI) is a hot topic in software development. CI is a critical enabler for Agile methods and higher software development velocity and productivity. With properly executed CI, developers get feedback on their changes immediately upon committing code, and errors get fixed faster. However, for many embedded systems, lack of hardware, complexity of hardware, or the inconvenience of system setup limits the ways in which CI can be applied. Using a virtual platform and simulation approach, it is possible to use standard PCs and servers to run code destined for even deeply embedded target systems, making an effective CI setup possible that is both more flexible and cheaper than relying on hardware alone. A simulated CI setup also brings benefits that are unique to simulation, such as better bug handling the ability to test code dealing with faults and extreme system setups or situations.**

Keywords—simulation, continuous integration, testing, automation

I. INTRODUCTION

Continuous integration (CI) is an important component of modern Agile software engineering practice. While the details of CI differ depending on whom you ask, a key part is that rather than waiting until the last minute to integrate all the many different pieces of code in a system, integration and most importantly integration testing is performed as early as possible, as soon as code is ready to run. You cannot really do Agile software development fully unless you have automated builds, automated tests, and automated successive integration – continuous integration. Embedded software developers are actively embracing Agile practices, but are often blocked from doing it fully due to the issues inherent in working with embedded hardware [1].

A properly implemented and employed CI system shortens the lead time from coding to deployed products, and increases the overall quality of the code and the system being shipped. With CI, errors are found faster which leads to lower cost for fixing the errors, and less risk of showstopper integration issues when it is time to ship the product. With CI, there is always something that can ship, which is a key part of Agile software development practices.

In CI, each piece of code added to a system should be tested as soon as possible and as quickly as possible, to make sure that feedback reaches the developers while the new code is still fresh in their mind. The most common technique is to build and test as part of the check-in cycle for all code, which puts access to test systems on the critical path for developers.

Testing soon and testing quickly is logistically simple for IT applications where any standard computer or cloud computing instance can be used for testing. However, for embedded systems and distributed systems, it can be a real issue to perform continuous integration and automated immediate testing. The problem is that running code on an embedded system typically requires a particular type of board or even multiple boards. If multiple boards are involved, they need to be connected in the correct way, and the connections between them configured appropriately. There is also a need for some kind of environment – an embedded system rarely operates in isolation, it is rather a system that is deeply embedded in its environment, and depends on having the environment in order to do anything useful. Thus, CI for embedded systems tends to be more difficult to achieve, due to the dependency on particular hardware the dependence on external inputs and outputs.

Using simulation for the computer as well as the environment portion of the embedded system offers a potential solution that allows for true automated and continuous integration even for embedded software developers. In our experience, we can achieve this by using high-speed virtual platforms along with models of networks and simulators for the physical world that the embedded system interacts with.

II. CONTINUOUS INTEGRATION

A continuous integration setup is fundamentally an automatic test framework, where code is successively integrated into larger and larger subsystems. As shown in

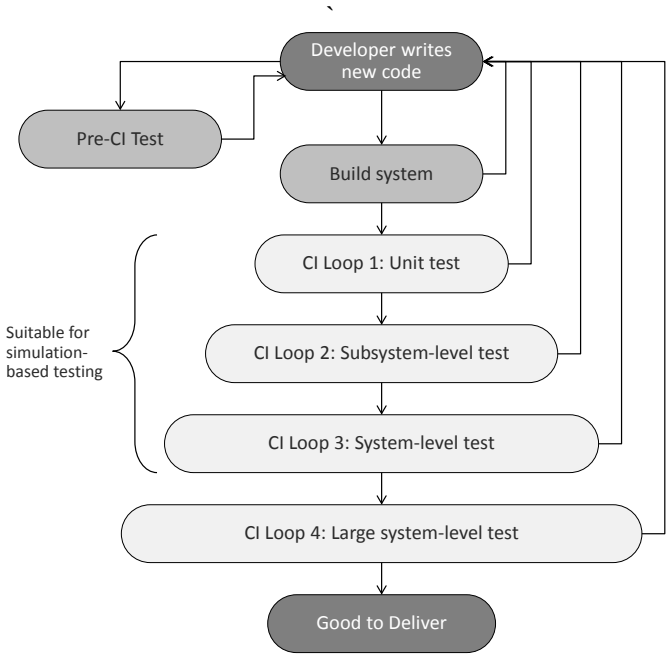

Figure 1. Continuous Integration Loops

Figure 1, the CI setup typically consists of a number of *CI loops*, each loop including a larger and larger subset of the system – both in terms of hardware and software.

The CI system is typically driven by code that is checked in to the main branch by developers – and this means that there is normally a separate *pre-CI test* phase where developers test their code manually or informally. Once the code seems reasonably stable, it is submitted to main automatic build system and sent into the CI system proper.

It is critical to perform testing at multiple levels of integration, since each level tends to catch different types of bugs. Just doing system-level end-to-end testing on a completely integrated system will miss large classes of errors that are easy to find with more fine-grained tests [2]. Running unit tests are necessary to ensure system-level quality, but not sufficient. Integration testing will reveal many types of issues that are not found in unit tests [3].

Each successive CI loop covers a larger scope and takes more time to run. The first-level loops should ideally complete in a few minutes, to provide very quick developer feedback. At the tail of the process, the largest loops can run for days or even weeks, essentially being the final testing before delivery.

In our experience, simulation can be used for all but the largest test loops. In the end, you have to *test what you ship and ship what you test*, and that means that you have to test the system on the hardware that will be shipping.

It is important to note that CI cannot necessarily be applied to any arbitrary existing software stack – in most cases that we have seen, the software architecture has had to be changed to facilitate CI and Agile practices. A key requirement for success is that it is possible to build and integrate parts of a system, and that subsets of the entire system can be tested in isolation. Additionally unit tests and subsystem tests have to be defined, if they do not already exist. Fundamentally, unless the software system is structured to allow CI, no amount of tooling can make a system do CI. Automated testing can almost always be achieved, but true CI requires more than that.

III. Alternatives to Simulation

The obvious way to do testing and CI for embedded systems is to use hardware. Hardware setups, however, are always limited in access, and can be difficult to automate and configure quickly enough for small CI loops. In practice, hardware can be so difficult to set up, control, and fully automate that many companies have given up on using it for CI entirely. Instead, testing on hardware is done only quite late in the process using a mostly complete system, essentially going straight to the largest CI loop without using the smaller ones. This brings with it the well-known effect that defects are expensive to fix, since they are found late in the process.

As shown in Figure 2, a hardware test setup often consists of a board under test, a master PC that loads software onto the board and runs it, and a test data PC equipped with special interfaces such as CAN, Ethernet, FlexRay or industrial buses. To properly test embedded software, it is necessary to generate realistic data over the interfaces. That data is typically generated by models that run in real-time. Such hardware and tester setups tend to be very expensive and few in number – but they are necessary for doing tests on the hardware, and the software models they contain offer a good starting point for a simulated CI system setup. In model-driven development terms, setups like those in Figure 2 are usually referred to as HIL testing.

Unit testing on can be performed on development boards using the same architecture as the target board, as long as tests do not depend on accessing application-specific hardware. Stubs can be used to imitate the rest of the systems. However, once it is time to do integration tests, the actual target hardware is needed.

Since hardware is hard to use, a common solution is to develop an *API-based* or *shim-layer-based* simulator. In such a setup, the software is compiled to run on a Windows- or Linux-based PC, and the target hardware and operating system is represented by a set of API calls. An implementation of the APIs that work on the host is used to let the software run. This provides an environment where application code can run, but it will not be compiled with the real target compiler, it will not be integrated in the same way that software is for the real system,

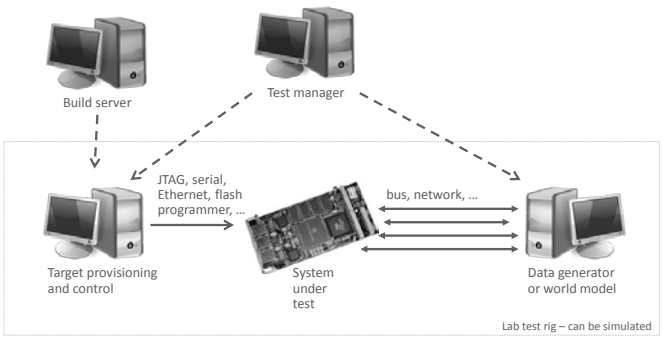

Figure 2. Hardware test rig

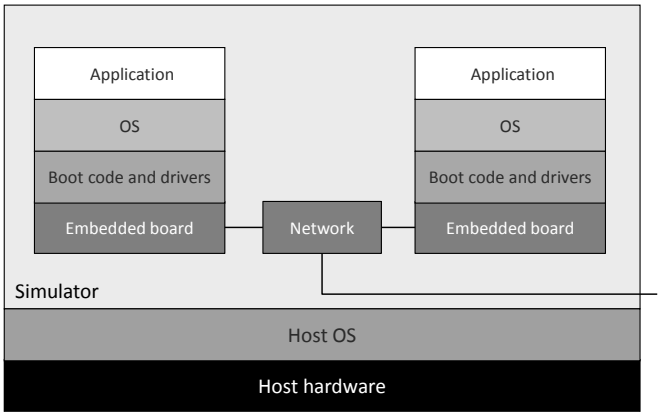

Figure 3. Virtual platform simulation of computer boards

and it will not run the real operating system kernel. Such a setup offers a quick way to do initial testing on the development host, but also tends to hide errors related to the real target behavior and build tools. It also involves creating and maintaining an additional build variant and the simulation framework. As such, it is most often used to test a few cooperating applications, but extending it to the full system is very rare – and also quite complicated. It is most useful as a quick pre-CI test.

Yet another approach is to run user-level software on the host with no API layer at all. This assumes that the target software is largely independent of the details of the target system. Such solutions miss issues related to building and integration code for the real target system, however, and cannot involve libraries only available in binary form. The net result tends to be a growing library of stubs that quickly turn it into something very similar to the API-level simulator.

Many companies evolve a hybrid of several of these approaches. One common hybrid is to combine a PC modeling the environment with a development board. Any differences between the development board and the end system are then addressed with software changes or shim layers on the target. In a few cases we have seen a hybrid system end up more costly than simply using the production hardware that was eliminated as a cost saving measure.

IV. SIMULATION FOR CI

The simulation type that we propose for application to CI, and that have successfully seen applied to real systems for the past ten years is the *fast functional transaction-level simulator* [4], also known as a *fast virtual platform*. Virtual platforms can run the same binaries that will run on the real system, and run them fast enough to be useful for software developers and massive testing. In our work, we have used the virtual platform system called Simics [5], integrated with various external simulators. However, most of the techniques we describe here are applicable to any virtual platform tool.

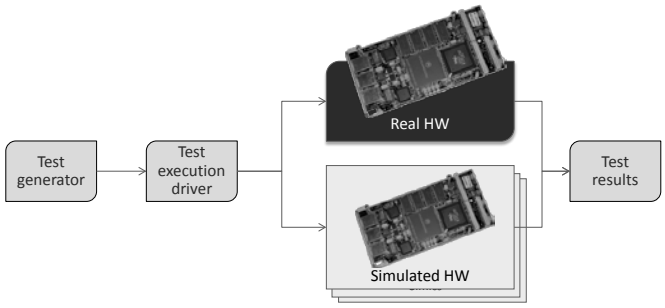

Figure 4. Simulation in a typical testing framework

The target software running on the virtual platform includes low-level firmware and boot loaders, hypervisors, operating systems, drivers, middleware, and applications. To achieve this, the virtual platform accurately models the aspects of the real system that are relevant for software, such as CPU instruction sets, device registers, memory maps, interrupts, and the functionality of the peripheral devices. A fast virtual platform typically does not model the detailed implementation of the hardware, such as bus protocols, clocks, pipelines, and caches. This provides a simulation that runs fast enough to run real workloads, and that can typically cover between 80% and 95% of all software tests and issues. To cover the tests that depend on real-world timing and absolute performance, hardware will have to be used. This is expected and normal.

As shown in Figure 3, you can run multiple boards inside a single simulation, along with the networks connecting them. It is also possible to connect the simulated computer boards (virtual platforms) to the outside world via networks or integrations with other simulators. Virtual platforms have proven to be fast enough to run even very large workloads including thousands of target processors [6].

When using Simics, the entire state of the simulated system can be saved to disk as a checkpoint for later restore, which enables issue management workflows and optimizations for starting runs from a known good and reusable state [7], as illustrated in Figure 7 below.

It should be noted that the virtual platform is not a test generation system or a test management system, but rather a system for *executing tests* and *collecting test outputs*. As shown in Figure 4, simulation is typically used with test automation frameworks alongside hardware, using the same tests as are run on the hardware – but running them in a more convenient way. This reuses existing assets in terms of test designs and test scripts, increasing flexibility and ensuring consistency. The problem we are trying to solve is not how to define the tests, but to efficiently execute them in spite of the challenges posed by embedded hardware. As shown in Figure 4, since the simulator is just a standard software program (or several, in the case that multiple separate simulation environments are used together) with no hardware dependencies, many instances can be run in parallel using a batch processing, cluster, grid, or cloud systems.

Most tests will be used on both simulation and hardware, but it also possible and recommended to extend the testing with tests that are only realistic to do when using a simulator, as discussed at the end of Section V. When there is a working CI system based on hardware, using simulation is a valid and valuable strategy to extend the testing. By using simulation, more tests can be run, making it possible to find more errors in testing and have fewer escape to the released software. With more test bandwidth available, more aspects of the product can be tested, increasing overall quality.

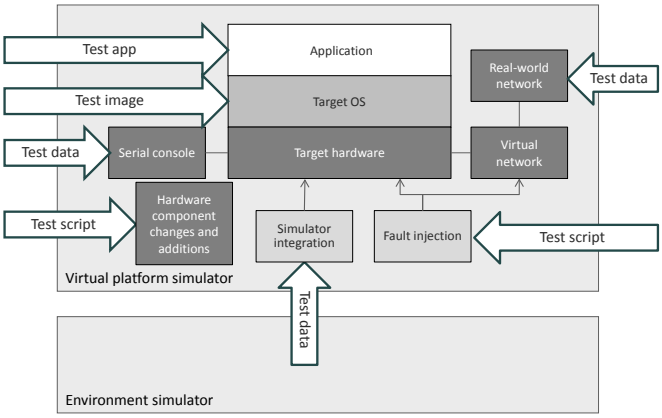

Figure 5. Getting Data into the Simulated System

It is also important to point out that a virtual platform model does not have to correspond to the complete physical hardware system to be useful. Rather, the most common way to enable CI using simulation is to design a set of model configurations that are useful for particular classes of test cases, and that do not include the entirety of the hardware system. If some piece of hardware is not actually being used, it can be skipped or replaced by a dummy in the model, reducing the work needed to build the model and the execution power needed to run it. Simulation setups must always be designed with the use cases in mind [4].

A. *Running Tests on the Simulator*

Given a set of software tests to run, the next question is how to get these tests to run on the simulator. Starting a virtual platform simulator from another software program is easy to automate. Following that, there are many different ways to feed in test cases, as illustrated in Figure 5.

The software provisioning that needs to happen before tests are run will usually follow the way software is loaded on the hardware. In some cases, this means building a single target binary with both an RTOS and its applications, and putting that binary into the memory of the simulated target system. In other cases, the target runs a dynamic OS like Linux, and uses networking to load new application binaries. Applications could also be provisioned by using disk images or simulator-specific back doors. The method used ultimately depends on the nature of the target system and its software. As discussed below and shown in Figure 7, it is a good idea to save checkpoints along the way to avoid having to repeat the same setup operation multiple times.

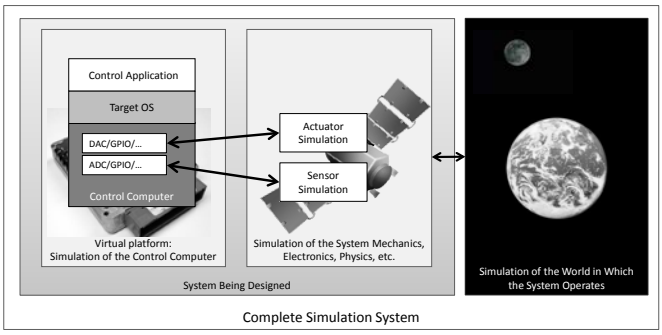

Figure 6. Simulation of the Entire System

Sometimes, the target software itself is the test data. A fresh build of an operating system would be loaded onto the virtual platform to test that it boots and initializes hardware correctly. A test application with its own built-in test data would be loaded on the target system and executed with no external interaction.

When tests are driven by external stimuli, it is best practice to use the same input paths as with a physical board. Most commonly, network connections or serial ports are used to push test data into the system – be it simulated or real.

In addition, using a simulator makes it possible to inject other types of test stimuli that are hard or even impossible to automate on hardware. In Figure 5, we include the example of using the hardware configuration as a test stimulus. In simulation, changing the hardware setup is merely a software operation that constructs a different hardware system. In the real world, this would mean manually reconfiguring the hardware in a lab, or possibly using a robot to perform the same.

B. *Integrating the Environment*

The environment outside of the computer board or control unit is a very important part of an embedded system. The specific environment differs from system to system; for a mobile phone or base station, the environment is a cellular network. For an automotive control system, it could be a hybrid drive train. For a satellite, it could be the position of the earth, sun, and stars, alongside traffic from mission control. Regardless of the details, the environment needs to be brought into the CI system at some point and in some way.

Unit-tests and subsystem tests can often be run using fixed inputs or a simple stub or test driver representing the outside world. Such tests drivers are typically part of the tested software stack, essentially compiling the test data into the system under test.

For system-level tests, it is usually a good idea to connect the virtual platform running the control software to a simulation of the environment. Such environmental simulations are commonly found in model-driven and simulation-driven design workflows, and they can be reused for simulation-based CI. Simulations can also come from hardware-based tests, as discussed in Section III above. Figure 6 shows how integrated simulation systems are typically structured. Existing environment simulation systems such as CAN stimuli generators can also be used to provide the external world model for the system under tests.

There is a simulated control board featuring simulation of the hardware input and output ports, and running an integrated software stack including the device drivers for the input and output hardware. The modeled inputs and output devices connect to models of the sensors and actuators which are part of the system being designed. Finally, the system being designed interacts with a model of the world in which it operates. When all these components are software simulations, they can all be part of an automated CI workflow, making the environment part of the continuous integration.

In some cases, the real world is directly connected to the simulated system under test in a Hardware-in-the-Loop (HIL) setup, allowing the software to sense and actuate the real world. That does require that the simulation runs fast enough to keep up with the real world.

We have seen cases where the simulation of the rest of the world was actually run on another virtual platform (one of the machines in Figure 3 would actually simulate the environment for the other).

C. Managing Intermediate State

Using virtual platform checkpoints, it is possible to save intermediate points in the test setups, such as the point where a system has been booted and after software has been loaded. Figure 7 shows a typical workflow where the system to use for tests is first booted, the booted state is saved, and used as the starting point for loading software. Since checkpoints should be handled as read-only, it is possible to base many runs off of the same checkpoint. Once software is loaded onto the system, another checkpoint is saved, and this checkpoint is used as the starting point for a series of tests.

On a hardware system, each test would have to start by booting the system or cleaning it in some way to remove the effects of previous tests. In a simulator, each run can start from a known good state, with no pollution from other tests. Checkpoints can save a lot of time in starting tests by removing this overhead.

We have also seen customers use checkpoints to manage the setups used for testing in a more proactive way. For example, by creating a checkpoint of each nightly build of the basic software platform, all developers and all tests run on a particular day will start from a known and well-defined state, rather than relying on whatever software happens to be loaded on the target systems from the previous day [4].

D. Reporting Issues

Figure 7 also shows how checkpoints are used to manage issue reports from testing. In addition to the classical information in an issue report (text describing what happened, collections of logs and serial port output, version and configuration data, etc.), checkpoints (containing a recording of all asynchronous inputs) can be used to provide the developer with the precise hardware and software state at the time that the issue hit [7]. This removes the guesswork in understanding what the test did and how the software failed, and is a tremendous boost for debugging efficiency.

The checkpointing methodology works with external simulators or data generators, by simply recording the interaction between Simics and the external simulator. When reproducing the issue, the data exchange is simply replayed, without the need for the external simulator or data source. Such *record-replay debugging* is a very powerful paradigm for dealing with issues that appear in complex real-time and distributed systems with many things happening at once.

This type of efficient feedback loop from testing to development is especially important for CI, since the developer is expected to deal quickly with issues that are found, while being quite removed from the actual testing going on. In manual interactive testing, the distance is typically much smaller as the developer is doing the testing just as the code is being developed. Using checkpoints and automated issue generation brings down the time needed to get back to a developer, and provides more information to make it easier to understand what happened.

V. Problems Solved by Simulation-Based CI

By using a simulator to run code for CI of embedded code, we solve a number of problems that typically affect hardware-based setups.

A. Hardware Control

Compared to hardware, managing a simulated test system is much easier. As the simulation is just software, it will not run out of control, hang, or become unresponsive due to a bad hardware configuration or total target software failure. The simulator program itself will always remain in control, and allow runs to be started and stopped at will. It is also easier to manage multiple software programs than multiple hardware units. Where a physical test system will need to coordinate multiple pieces of hardware and software as shown in Figure 2, a simulation-based setup has the much simpler task of coordinating a few software programs.

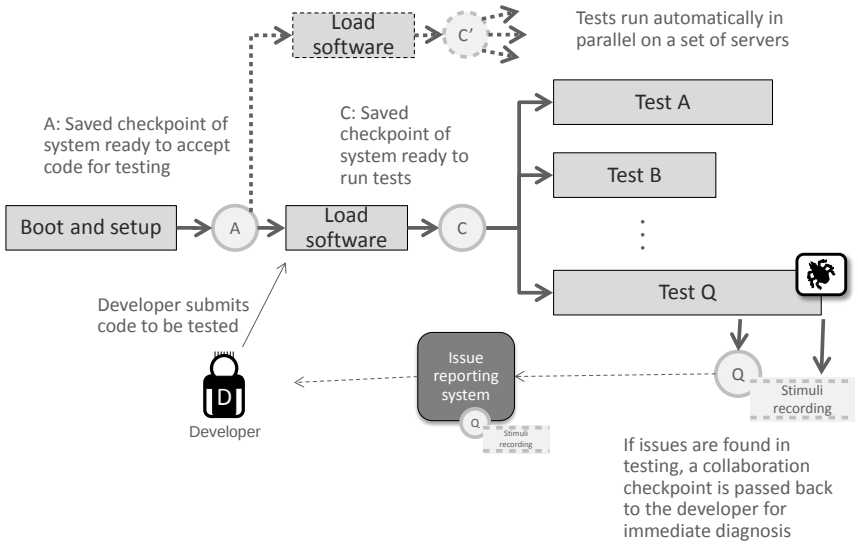

Figure 7. Using Checkpoints to Reuse Intermediate States and Communicate Issues

B. Hardware Asset Inflexibility

With a simulation, the same physical hardware box – a generic PC or server - can be used to run software for a wide variety of embedded systems. This provides much more flexibility than hardware labs, since one hardware system cannot be repurposed to test software build for another system.

C. Hardware Availability Bottlenecks

When working with physical labs, hardware availability is almost always an issue. The number of physical systems

available is limited, and time on them rationed in some way. With a simulated setup, hardware availability is not an issue, since the simulator can make any computer simulate any embedded board. The simulator augments the availability of physical boards, removing the constraints that hardware availability puts on both developer spontaneous testing and structured CI testing. With simulation, each user can have a system of any particular kind to run on whenever they need it.

It is also possible to temporarily increase the testing pool by borrowing computer resources from other groups within the same company, or even renting time on a cloud computing service.

D. Test Run Latency

When hardware is the bottleneck for testing, it is common to see test campaigns becoming longer and longer. The time from the point that a job is submitted for execution to the point that it is completed gets longer and longer, as it has to wait for a hardware unit to become available. With a simulation-based setup, much more virtual hardware is available for the quick tests, and this leads to shorter test latency. Using checkpoints as illustrated in Figure 7, the time to get a particular test running can be much shorter as the setup is essentially cached for reuse.

Test latency is also reduced by the potential for more parallel testing, making it possible to run through a particular set of tests in shorter time than on hardware. We have seen users previously limited by hardware greatly increase their test coverage and frequency thanks to parallel testing – if you can run your test suites daily rather than weekly, errors will get found earlier, regressions will be caught quicker, and fewer errors will make it out in the field, reducing development costs and increasing product quality.

E. Test Design Reflecting the Hardware, not the Problem

When limited by hardware availability, real-world tests are often designed to fit into available testing resources rather than to detect problems. This is a necessity in practice, as some testing is still infinitely better than no testing. However, with virtually unlimited hardware availability, this is no longer as much of an issue. Tests do not have to be scaled down or modified to match available hardware; instead, the virtual hardware can be set up to match the tests that need to be performed.

F. Limited Configuration Space

In a hardware lab, there is normally a single or a few different hardware configurations available for testing. These might not represent all the different configurations actually found in the real world, but rather a compromise between expense and breadth of testing. With a simulation, there is no need to limit the configurations available, since there is an infinite pool of boards available. It is very easy to create and save and reuse configurations, since they are all just software configurations and setup scripts.

Configuration richness and variety is important to capture *omission errors* in testing. Such errors are quite common in code that interacts with a complex environment and that consists of many separate units that are integrated together [3]. To trigger them, it is necessary to run as many different hardware and software configurations as ever possible.

G. Using Development Boards rather than the Real Thing

Another aspect of hardware availability that affects test design is the common use of development boards rather than production boards. In many types of systems, the real boards are rare, expensive, and hard to communicate with. With a simulation, a model of the real board can be employed, removing the need for a variant build for the development board. This makes the testing have higher fidelity, and saves the cost to maintain an extra build variant to support testing.

We have seen real systems with no connectors available for testing – no serial, JTAG, or utility Ethernet at all. Instead, a development board was used for automatic basic testing, and then real tests were painstakingly carried out on the real systems with a fairly low frequency. With a simulation of the board, it was possible to inject software and run test cases without modifying the software so that it could run on the development board. Instead, the simulator opened up a virtual back door into the system to inject software and test the real thing with a simulation of the real mechanical system being controlled.

H. Partial System Integration

It is often desirable to integrate some subsystem or other part of a larger system for deep testing, without having to integrate it with the whole system. Indeed, when working with hardware in development, the rest of the system might well not exist at all. Such partial integration is much easier to do using a virtual platform and simulation, since it is simple to capture the interaction between the subsystem under test and the rest of the system, and to inject information that convinces the subsystem under test that the rest of the world is there.

Such stubbing is often more difficult to perform in hardware, as the interfaces to be intercepted can be difficult to get at. In some cases, it is easy, as for a CAN bus, but for a wireless communications system, stubbing things is much harder. Stubbing a rack back plane is also hard in hardware. When it gets down to components on a board, hardware stubbing is pretty much impossible today.

I. Very Large Setups

There are cases where hardware is just impossible to manage when scaling towards the theoretical limits of a system. For example, in Internet-of-Things (IoT) sensor systems, you often need to have hundreds or even thousands of nodes in a single system to test the software and system behavior. In a simulated setting, it is possible to automatically create very large setups without having to spend the incredible amount of time it would take to set up, maintain, and reconfigure such a system in hardware form. Even when hardware is very cheap, configuring and deploying hundreds of separate hardware units is expensive.

Another example would be testing software for hardware that is in development or in prototype state - such hardware is usually very limited in quantity and getting tens or hundreds of nodes for testing networked systems and distributed systems just is not possible.

J. *Testing Fault Handling*

The code that handles faults and erroneous conditions in a system can be very difficult to test on hardware [8], and yet it is critical to ensuring system reliability and resiliency. Hardware test rigs tend to be expensive [9], and testing is often destructive, which limits fault injection testing on hardware to fairly rare cases.

In contrast, in a simulator, injecting faults is very easy since any part of the state can be accessed and changed. Thus, systematic, automatic, and reproducible testing of hardware fault handlers and system error recovery mechanisms can be made part of the CI testing. This will ensure that fault handling remains functional over time, and increase system quality. Often, the fault and error handling code in a system is the least tested and a constant source of issues [10]. Using simulation and injected faults, such code can be tested to a much higher extent than is possible using hardware.

A related issue is to simulate naturally varying environmental conditions as part of continuous testing. For example, for a wireless network system, it is very valuable to test software behavior in the presence of weak signals and asymmetric reachability. Such testing is very easy to perform using a model of the network, but very difficult to perform in the real world. Network conditions are not really faults per se, but rather expected behaviors from an uncooperative physical world. In general, testing how a system responds to various environmental conditions is a very valuable use case for simulation – and one where simulation is being used extensively for physical systems already.

VI. SUMMARY

Continuous integration (CI) is an important part of modern software engineering practice. By using CI, companies achieve higher quality and enable continuous deployment, among other benefits. However, implementing CI for embedded systems can be a real challenge due to the dependency on particular processors, particular hardware, and particular environments. Using simulation for both the computer hardware and environment used in an embedded system can enable CI for systems that seem "impossible" to automate. Simulation can also bring other benefits, such as better feedback loops to developers for issues discovered in testing, and expansion of testing to handle faults and difficult-to-setup configurations.

REFERENCES

[1] Andre Girard, *Agile and DevOps for Embedded Systems – Engineering Trends Analysis*, VDC Research, 2014.

[2] Mike Bland, Finding More than One Worm in the Apple, *Communications of the ACM*, Volume 57, Number 7, July 2014.

[3] Sigrid Eldh, *On Test Design*, Doctoral Dissertation #105, Mälardalens Högskola, Västerås, October 2011.

[4] Daniel Aarno and Jakob Engblom, *Software and System Development using Virtual Platforms - Full-System Simulation with Wind River Simics*, Morgan Kaufmann Publishers, 2014.

[5] Stuart Douglas (ed), *Simics Unleashed – Applications of Virtual Platforms*, Intel Technology Journal , volume 17, September 2013.

[6] Grigory Rechistov. "Simics on the shared computing clusters: the practical experience of integration and scalability", *Intel Technology Journal*, Volume 17, Issue 2, 2013.

[7] Jakob Engblom. "Transporting Bugs with Checkpoints," *Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2010)*, Southampton, UK, September 15-16, 2010.

[8] Steve Chessin. "Injecting Errors for Fun and Profit", *Communications of the ACM*, Volume 53, Issue 9, September 2010.

[9] Masafumi Matsuo, Yuji Uchiyama, Yuichi Kurita, "Quality Assurance for Stable Server Operation", *Fujitsu Science and Technology Journal*, Vol 47, No 2, April 2011.

[10] Jack Ganssle, *The Firmware Handbook*, Newnes, 2004