Accelerate DevOps with Continuous Integration and Simulation

A How-to Guide for Embedded Development

WNDRVR

EXECUTIVE SUMMARY

Adopting the practice of continuous integration (CI) can be difficult, especially when developing software for embedded systems. Practices such as DevOps and CI are designed to enable engineers to constantly improve and update their products, but these processes can break down without access to the target system, a way to collaborate with other teams and team members, and the ability to automate tests. This paper outlines how simulation can enable teams to more effectively manage their integration and test practices.

Key points include:

- How a combination of actual hardware and simulation models can allow your testing to scale beyond what is possible with hardware alone
- Recommended strategies to increase effectiveness of simulated testing
- How simulation can automate testing for any kind of target
- How simulation can enable better collaboration and more thorough testing
- Some problems encountered when using hardware alone, and how simulation can overcome them

TABLE OF CONTENTS

WNDRVR

INTRODUCTION

CI is an important component of a modern DevOps practice. While the details of CI differ depending on whom you ask, a key feature 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 adopt DevOps, or even Agile software development, fully unless you have automated builds, automated tests, and automated successive integration—that is, continuous integration. Embedded software developers are actively embracing DevOps, but they are often blocked from doing so fully due to the issues inherent in working with embedded hardware.

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 lower risk of showstopper integration issues when it is time to ship the product.

In CI, each piece of code that is added or changed 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 minds. Ideally, tests should be run and results reported back to the developers within minutes. 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 immediate, automated 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 in which to test the system—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. CI for embedded systems thus tends to be more difficult to achieve due to the dependency on particular hardware, and the dependence on external inputs and outputs (I/Os) and the hardware necessary to drive the I/Os.

Using simulation for the embedded system and its environment offers a potential solution that allows for true automated testing and CI, even for embedded software developers. Wind River® Simics® helps achieve this by using high-speed virtual platform models of the embedded system along with models of networks and simulators for the physical environment that the embedded system interacts with.

> Embedded software developers are actively embracing DevOps, but they are often blocked from doing so fully due to the issues inherent in working with embedded software.

CONTINUOUS INTEGRATION AND SIMULATION

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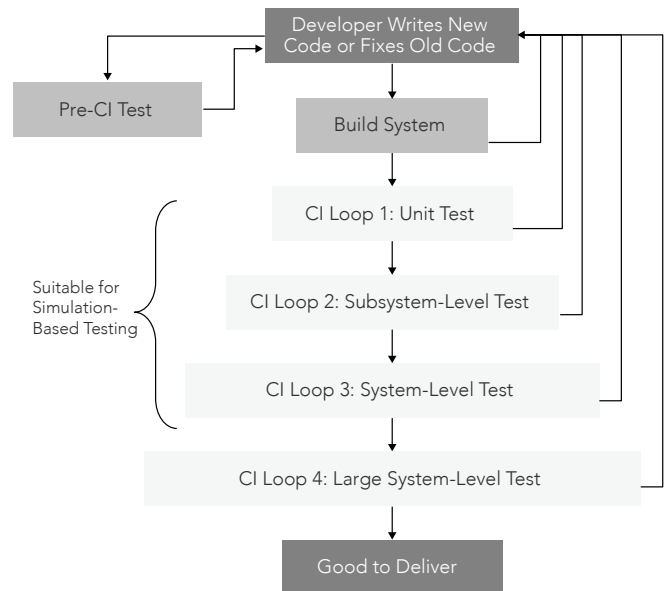

Figure 1. Continuous integration loops

The CI system is typically started when code is checked in by developers. Since code needs to have some basic level of quality before being checked in, there is normally a separate pre-CI test phase where developers test their code manually or by using small-scale automatic tests to make sure the code is at least basically sound and

WNDRVR

probably won't break the build. Once the code seems reasonably stable, it is submitted to the main automatic build system and sent into the CI system proper. The work of the CI system is to a large extent regression testing—making sure that the component that was changed or added does not break existing expected behaviors of the system.

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. Running unit tests is necessary to ensure system-level quality, but it is not sufficient. Integration testing will reveal many types of issues that are not found in unit tests, and each level of integration will reveal its own set of bugs.

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 end of the process, the largest loops can run for days or even weeks.

The largest loops are sometimes considered part of the CI process, and sometimes are handled by a specialized quality assurance or delivery team that makes sure the code truly meets the quality criteria needed to ship. If the code that comes out of the final CI loop is ready to ship, we enter the domain of continuous delivery (CD), which is the next step beyond CI. Simulation can be used for all but the last and largest test loops. In the end, you have to "test what you ship and ship what you test," and that means you have to test the system on the hardware that will be shipping—but that is the last step before release, and most testing up to that point can be done using simulation.

CI cannot necessarily be applied arbitrarily to any existing software stack; in most cases, the software architecture has had to be changed to facilitate CI and DevOps. A key requirement for success is that it be possible to build and integrate parts of a system, and that subsets of the entire system can be tested in isolation—that is, the system must be modular in order to enable CI. Additionally, unit tests and subsystem tests must be defined, if they do not already exist. Using simulation and making testing automated does not automatically mean that you have a CI system.

HARDWARE-BASED CONTINUOUS INTEGRATION

The basic way to perform testing and CI for embedded systems is to use hardware. 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 interfaces such as serial, AFDX, ARINC 429, MIL-STD-1553, CAN, Ethernet, FlexRay, 802.15.4, and other specific buses and networks used to communicate with the real target board.
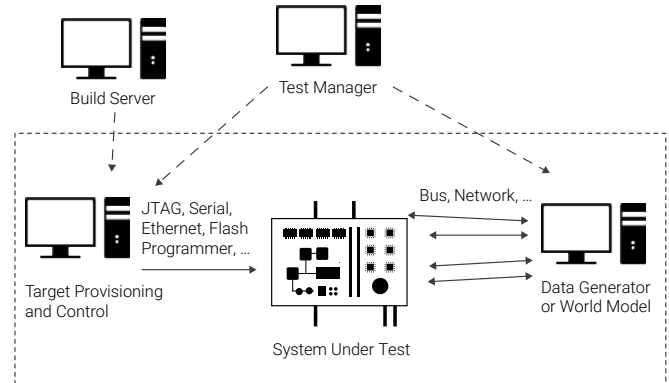

Figure 2.   Typical hardware lab test rig

To test the embedded software on the system under test, it is necessary to have input data to communicate to the target. That is the job of the data generator or world model PC in Figure 2. The input data can come from recordings of real-world inputs, from manually written files of input data, or from models that run in real time. For example, a satellite test bed would have a simulation of how the stars move as the satellite orbits the earth, and it would provide pictures of the sky as inputs to the star tracking system.

While the data generator is shown as a PC in Figure 2, it can also be specialized test hardware, in particular for high-performance systems where the data volumes needed are huge and latency requirements are tight. It is not uncommon to have a whole rack of specialized computer boards connected to a hardware test system over a large number of special cables. Needless to say, such setups can quickly become very expensive and unwieldy—not to mention quite cumbersome to maintain over time.

The target provisioning and control PC is responsible for managing the target system, including loading software on it, resetting it, starting target software, and cleaning up between tests. The PCs directly connected to the target system are controlled by a test management system that often runs on a central server.

Hardware test setups are necessary for doing tests on the hardware and are universally used for at least the final integration testing, and sometimes also earlier integration testing. But access to hardware test setups is typically limited, since there are not that many setups to go around.

WNDRVR

Another common problem is that the hardware test lab setups are so complicated that only a few engineers (or even just one) master it. This unintentional specialization, with each team only really knowing how to run a few types of tests, in turn leads to bottlenecks, long turnaround times, and inefficient communication. Such specialization runs counter to the whole Agile collaborative spirit, where flexibility, velocity, and quick feedback loops are essential.

Furthermore, hardware test setups can be difficult to automate and configure quickly enough for small CI loops. The result is that 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 classic big-bang waterfall integration rather than a gradual CI process. And with this practice comes the well-known effect that defects are expensive to fix, since they are found late in the process.

> Another common problem is that the hardware test lab setups are so complicated that only a few engineers (or even just one) master it. This unintentional specialization, with each team only really knowing how to run a few types of tests, in turn leads to bottlenecks, long turnaround times, and inefficient communication. Such specialization runs counter to the Agile collaborative spirit, where flexibility, velocity, and quick feedback loops are essential.

To work around the inconvenience and lack of access to hardware, companies have tried various solutions.

Unit testing 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. This solution gets around the need to have real target boards, but at the cost of not really running the final integrated software stack. Once it is time to do integration tests, the actual target hardware is needed. Development boards are also hardware resources and will be limited in availability too.

Another 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 are represented by a set of API calls that can be used on both the target and the host. This solution 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, and it will not run the real OS kernel. Such a setup offers a quick way to do initial testing on the development host, but it also tends to hide errors related to the real target behavior and build tools. In many cases, tests just cannot be run on this type of simulation, since they need a larger context than is available. Thus, API-based tests are most often used to test a few well-behaved applications, but extending them to the full system is very rare, and also quite complicated. They are most useful as quick pre-CI tests. API-based simulators also require the development organization to create and maintain an additional build variant as well as the simulation framework itself. This cost can be quite significant in practice, even if it seems small initially.

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 target end-system are then addressed with software changes in the code or in a shim layer on the target. Sometimes the hybrid system can end up being more expensive than simply using the production hardware that was eliminated as a cost-saving measure.

Overall, hardware solutions have various issues that prevent companies from moving fully to a CI flow that is as smooth and efficient as that experienced by general IT companies.

> Overall, hardware solutions have various issues that prevent companies from moving fully to a CI flow that is as smooth and efficient as that experienced by general IT companies.

USING SIMULATION FOR CONTINUOUS INTEGRATION

To get around the problems caused by using hardware for CI, companies have turned to simulation based on Simics. Using simulation, testing can be performed using standard PCs and servers, reducing the reliance on hardware and expanding the access to hardware virtually. With simulation, the test setup shown above in Figure 2 would look like the one in Figure 3. The PCs servicing and controlling the target board are replaced with simulation modules, and the target board is replaced with a virtual platform.
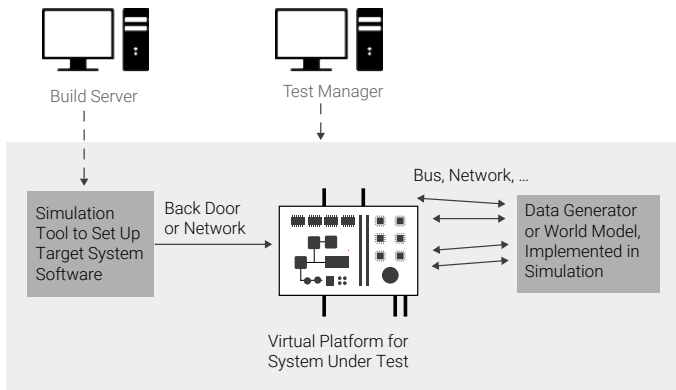
WNDRVR

Figure 3. Simulation-based lab test rig

Compared to hardware, managing a simulated test system is much easier. Because 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, as shown in Figure 3.

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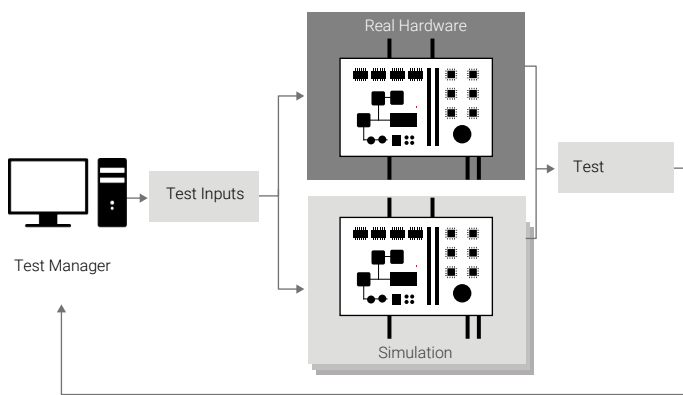

Figure 4. Test management and simulation

As shown in Figure 4, the simulator augments the availability of physical boards, removing the constraints that hardware availability places on both developers' spontaneous testing and structured CI testing. With simulation, each user can have a system of any kind to run 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 by renting time on a cloud computing service.

In contrast, with physical labs, hardware availability is almost always an issue. The number of physical systems available is limited, and time on them tightly controlled, forcing developers to limit testing or test when their time slot comes up rather than when their code is in good shape to be tested. It is also common to see test campaigns becoming longer and longer on hardware, as tests are added over time while the number of labs remains the same. The time from the point when a job is submitted for execution to the point when it is completed gets longer and longer, as it has to wait for a hardware unit to become available. With a simulation-based setup, test latency is shorter, and thus it is possible to provide faster—and therefore better—feedback to the developers.

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 more quickly, and fewer errors will make it out in the field, reducing development costs and increasing product quality.

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, as some testing is still infinitely better than no testing. But with virtually unlimited hardware availability, 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. This includes creating virtual setups that have no counterpart in the physical lab, as well as dynamically varying the hardware setup during a test. Thus, the attainable test matrix is expanded beyond what is possible with the physical labs.

At the same time, the simulation setup does not have to correspond to the complete physical hardware system to be useful. Rather, the

WNDRVR

most common way to enable CI using simulation is to design a set of configurations that are useful for particular classes of test cases, and that do not include the entirety of the 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. The simulation setup scales with the tests to be performed.

SIMICS VIRTUAL PLATFORMS

The virtual platforms suitable for use in CI are fast functional transaction-level models such as Simics. A fast virtual platform such as Simics typically does not model the detailed implementation of the hardware, such as bus protocols, clocks, pipelines, and caches. In this way, Simics 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, which is expected and normal. There is a basic choice to be made between running a lot of software with a simplified timing model, and very little software with a high level of detail. In today's systems, it is usually the case that more issues are found by running a lot of code rather than by cranking up the detail level.
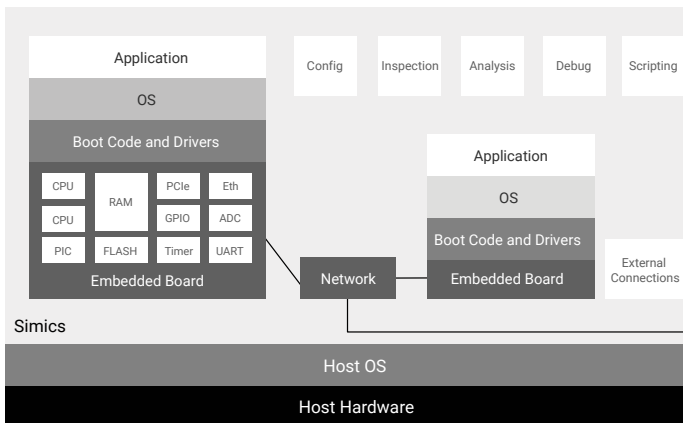

Figure 5. Simics simulation

A typical Simics target setup is shown in Figure 5. The target software running on the simulated hardware boards includes low-level firmware and boot loaders, hypervisors, operating systems, drivers, middleware, and applications. To achieve this, Simics 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.

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. Simics has proven to be fast enough to run even very large workloads, including thousands of target processors.

Figure 5 also shows that Simics provides features such as configuration management, scripting, automated debugging, and analysis tools that help when constructing simulated CI and software development environments. When using Simics, the entire state of the simulated system can be saved to disk as a checkpoint for later restoration, which enables issue management workflows and optimizations for starting runs from a known good and reusable state, as illustrated in Figure 7.
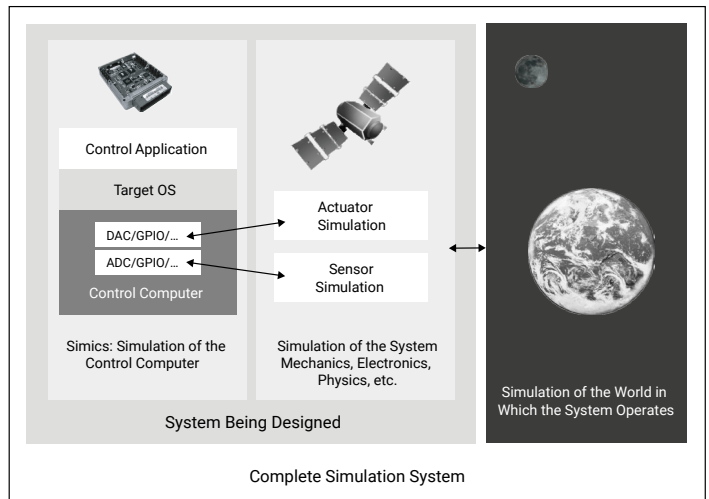

Figure 6. Connecting the virtual platform to the environment

In Figure 3 we see a simulation-internal connection between the data generator or world model and the system under test. With simulation, you could potentially do this in various simulation-specific ways, but for most integration tests it is usually a good idea to connect the virtual platform running the control software to a simulation of the environment in the same way that they are connected in the real world. The recommended structure of such simulations is shown in Figure 6. There is a simulated control computer board featuring simulation of the hardware I/O ports, and running an integrated software stack including the device drivers for the I/O hardware. The modeled I/O devices connect to models of the sensors and actuators that are part of the system being designed. There are also cases where the simulation of the rest of the world is run on another virtual platform (one of the machines in Figure 5 would actually simulate the environment for the other).

WNDRVR

WORKFLOW OPTIMIZATION USING CHECKPOINTS

Using Simics for virtual platform simulation makes it possible to optimize the test workflows, including new ways to provide feedback to the developers from test runs. Simics checkpoints capture the entire state of the simulated system to disk and allow the saved state to be instantly brought up in Simics on the same or a different machine, at any point in time and at any location. The first use of checkpoints is to save intermediate points in the test flow, such as the point after a system has finished booting, or after the software to test has been loaded. Figure 7 shows a typical Simics-based workflow where the system is first booted, then the booted state is saved and used as the starting point for loading software. 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. Since checkpoints should be handled as read-only items, it is possible to base many test runs off the same checkpoint. 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 the test. In a simulator, each run can start from a known consistent and good state, with no pollution from other tests. By removing this overhead, checkpoints can save a lot of time when starting tests, as well as avoid spurious results by ensuring a consistent initial state across batches of tests.
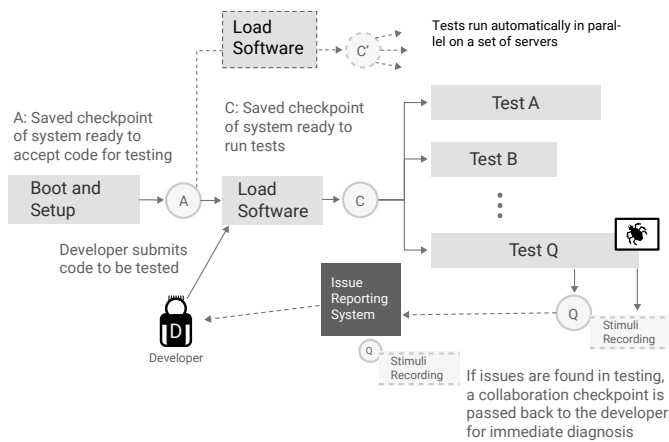

Figure 7. Workflow with Simics checkpoints

Figure 7 also shows how checkpoints are used to manage issue reports from testing. In addition to the traditional 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 responsible for the code that broke the test with the precise hardware and software state at the time the issue hit. This ability removes the guesswork in understanding what the test did and how the software failed and is a tremendous boost for debugging efficiency.

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.

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. Once a recording has been replayed in a Simics session, reverse debugging can be used within that session to quickly and efficiently diagnose the issue.

TESTING FOR FAULTS AND RARE EVENTS

Since the goal of CI is to ensure that code keeps working, it is important to test as many different scenarios as possible, and to keep doing so in an automated fashion every time a piece of code is changed and reintegrated. This is particularly tricky for code that handles faults and erroneous conditions in a system. Testing such code using hardware is difficult, and yet it is critical to ensuring system reliability and resiliency. Hardware test rigs for fault injection tend to be expensive, and testing is often destructive, which limits how much testing can realistically be performed.

In a simulator, in contrast, 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 practice will ensure that fault handling remains functional over time and will increase system quality. Often, the

WNDRVR

fault and error handling code in a system is the least tested, and a constant source of issues. Using simulation and injected faults, such code can be tested to a much higher extent than is possible using hardware.

One example of the type of testing that simulation allows is the pulling of a board from a system, and checking that the system detects that the board is removed and rebalances the software load to the new system configuration. In the context of CI, doing so makes it possible to test that the platform and middleware perform as designed when integrated with the hardware and each other.

Simulation also enables the introduction of varying environmental conditions as part of CI and testing. In the end, an embedded system is integrated into the world, and that integration needs to be tested—not for "faults" exactly, but rather for behavior that is expected from an uncooperative physical world. Testing how a system responds to various environmental conditions is an important use case for simulation, and one where simulation is being used extensively for physical systems already. For example, for a wireless network system such as the one shown in Figure 8, the integrated software behavior should be tested in the presence of weak signals and asymmetric reachability. Such testing is easy to perform using a model of the network, but difficult to perform in the real world. Each network link is available for change in the simulation, while trying to jam a real-world radio signal in a controlled way is very difficult.

Simulation is often the only practical way to systematically and continuously perform testing of system scaling. For example, in sensor systems in the Internet of Things, 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.

As shown in Figure 8, a simulation can be scaled from a small unit test network (1) to a small system test (2), and finally to a complete system including multiple types of nodes and a very large number of small sensor nodes (3). In Simics, each such configuration can be programmatically created by selecting the number of nodes of each type and their connectivity.

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

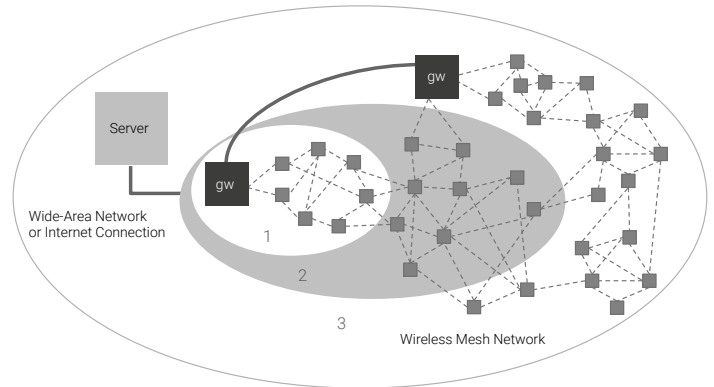networked systems and distributed systems is just not possible.

Figure 8. Example of scaling up the simulated target system

SIMULATION-BASED CI AND THE PRODUCT LIFECYCLE

The use of simulation to support CI means that it will be used during most of the product lifecycle. Figure 9 shows that CI (and thus CI using simulation) is applicable from platform development all the way to deployment and maintenance.

In platform development, hardware is integrated with the OS driver stack and firmware, and middleware is integrated on top of the operating system. Once the platform is sufficiently stable to allow application development to begin, integration testing also includes applications. Applications integrate with the target OS and middleware, as well as with each other. The platform tests are also part of the integration testing even as applications are added; there might be several different sets of CI loops that start at various points in the system integration.
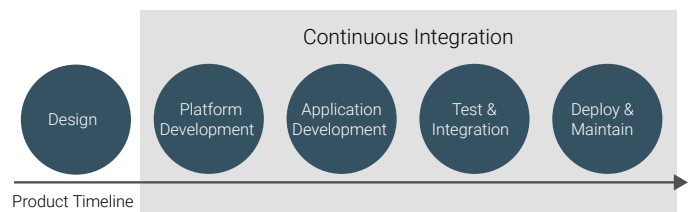

Figure 9. Traditional product lifecycle

CI means that integration testing is being pulled into earlier development phases—the whole point is to avoid waiting until the standard test phase to do integration. Indeed, as shown in Figure 10, test and integration morphs from a separate phase to a parallel track of development, where tests are designed and executed from very early on in the software lifecycle. Testing and test development become part of the development effort, supporting the evolution of the system and its software over time.

WNDR\/R

When using simulation for integration testing, the simulation setup is useful even after the first release of the integrated system has shipped. As the software is maintained with bug fixes, and new software is developed and software functionality expanded, CI is a key part of development practices. As the software continuously evolves, it has to be continuously integrated and tested so that existing functionality keeps working, and new functionality integrates correctly into the system.
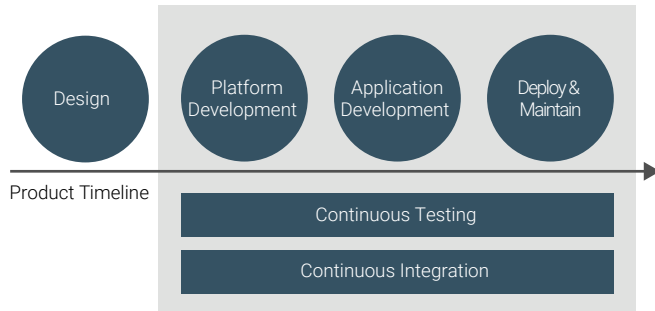

Figure 10. Continuous testing and integration

In addition to development, simulation can also be used to support other organizations within the company dealing with deployment of the system, such as support and training departments. A simulation setup can be used to reproduce issues from the field, and once an issue is reproduced, the bug reporting workflow illustrated in Figure 7 and discussed above can be applied. The simulation can also be used to support training of operators on a system.

CONCLUSION

CI is an important part of modern software engineering practice. By using CI, companies achieve higher quality and enable further enhancements, such as continuous delivery or 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 the environment surrounding an embedded system can enable CI for systems that seem "impossible" to automatically test. Simulation can also bring other benefits, such as faster feedback loops with better information to developers for issues discovered in testing, and expansion of testing to handle faults and difficult-to-set-up configurations.

Using Simics, many companies have successfully turned to simulation to augment their testing hardware setups and realize unprecedented development efficiencies.

WNDRVR