Jaakko Nahkala

Integration Test Framework for ACS880 Drives

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

9 September 2024

Abstract

Author:	Jaakko Nahkala
Title:	Integration Test Framework for ACS880 Drives
Number of Pages:	43 pages + 3 appendices
Date:	9 September 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart IoT Systems, Embedded Systems
Supervisors:	Tuomas Pulli, Design Manager
	Saana Vallius, Senior Lecturer

This thesis introduces a software testing framework for integration tests intended for the ABB ACS880 inverter control unit software. Previously, this framework was utilised in ACS880 supply drive software for integration testing. In the final year project, three integration tests utilising the integration test framework were adopted for the ACS880 inverter software.

In the thesis, integration tests are examined through model-based testing as well as white box and black box testing. A system overview of the ACS880 software is presented to examine the integration test framework as it is in-use on the supply software. Current state analysis is included to further explore the motivation for the project.

The goal of the project was to complement the ACS880 inverter software test coverage, to optimise test feedback and to encourage the utilisation of software testing tools in firmware development. To facilitate these goals, the integration test framework was incorporated into continuous integration (CI) processes. The initial test cases are explored in detail both in the user manual and the thesis.

In practice, existing software components such as build scripts, the integration test framework and continuous integration scripts were modified. New additions for the inverter software included a motor model interface, integration tests and a user manual published on the case company internal database.

Adopting the integration test suite to CI improved the test coverage of the ACS880 software noticeably. As evidence of this, the integration tests were some of the first tests on the inverter software to signal a fault relating to software virtualisation.

Keywords: Integration tests, software testing, testing framework, embedded systems, electric drives, variable frequency drives

---

Tiivistelmä

Tekijä:	Jaakko Nahkala
Otsikko:	Integrointitestiympäristö ACS880-perheen taajuusmuuttajille
Sivumäärä:	43 sivua + 3 liitettä
Aika:	9.9.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Information and Communications Technology
Ammatillinen pääaine:	Smart IoT Systems, Embedded Systems
Ohjaajat:	Suunnittelupäällikkö Tuomas Pulli Lehtori Saana Vallius

Tämä opinnäytetyö esittelee ohjelmistotestauskehyksen, jota käytetään integrointitestauksessa ABB ACS880 -taajuusmuuttajan ohjausyksikön ohjelmiston kehityksessä. Aikaisemmin tätä kehystä on hyödynnetty ACS880-virransyöttöyksikön ohjelmiston integrointitestauksessa. Tässä työssä esitellään kolme integrointitestiä, jotka käyttävät kyseistä testauskehystä ACS880-taajuusmuuttajan ohjelmistossa. Opinnäytetyöhön sisältyy myös käyttäjän opas integrointitestijärjestelmästä.

Opinnäytetyön tavoitteena on täydentää ACS880-taajuusmuuttajan ohjelmiston testauksen kattavuutta, nopeuttaa testauksen palauteprosessia ja kannustaa ohjelmistotestauksen työkalujen hyödyntämiseen ohjelmistokehityksessä. Näiden tavoitteiden saavuttamiseksi integrointitestauskehys on liitetty osaksi jatkuvaa integraatiota (CI), ja lisättyjä integrointitestejä tarkastellaan yksityiskohtaisesti.

Opinnäytetyön toteutus sisälsi osittain olemassa olevien ohjelmistokomponenttien muokkaamista. Näitä kohteita olivat rakennusskriptit, integrointitestauksen kehys ja jatkuvan integraation skriptit. Uusia lisäyksiä taajuusmuuttajan ohjelmistoon ovat moottorimallin rajapinta, integrointitestit ja käyttäjän opas, jotka on tallennettu kohdeyrityksen sisäiseen tietokantaan.

Integrointitestisarjan sisällyttäminen CI-prosessiin paransi tuntuvasti ACS880-ohjelmiston testauksen kattavuutta. Työssä lisätyt integrointitestit olivat ensimmäisiä testejä, jotka havaitsivat ohjelmiston virtualisointiin liittyvän vian taajuusmuuttajan ohjelmistossa.

Avainsanat: Integrointitestaus, ohjelmistotestaus, testauskehys, taajuusmuuttajat, sähkömoottorit

Contents

List of Abbreviations

ACS880: Family of industrial electric drives by ABB.

AC / DC: Alternating Current / Direct Current.

API: Application Programming Interface. Software interface connecting multiple parts of software together.

ASD/VFD: Adjustable Speed Drives / Variable Frequency Drives. Electric drives that allow for the adjustment of speed and torque of a motor by varying the frequency and voltage of the power supplied to it.

ATF: Automated Test Framework. Software automation framework for streamlining task execution.

CES: Critical Embedded System. Embedded systems that are critical for some operation. Term used in automotive, industrial and medical fields where failures can be catastrophic.

CI/CD: Continuous Integration / Continuous Deployment. Development process associated with agile development paradigms incorporating software regression tests and validation as integral part of software development.

CPU: Central Processing Unit. The main processor of a computer or a sophisticated embedded machine. Unlike most microcontroller (MCU) embedded devices a CPU can utilise a large number of peripherals.

DSP: Digital Signal Processing. A form of digital data processing that utilises algorithms for signal manipulation or data extraction.

DSU: Supply unit. Used to rectify AC to DC voltage.

GUI: Graphical User Interface. Software interface utilising graphical device to show information regarding the software operation.

HIL: Hardware In the Loop. Testing method that simulates behaviour of essential hardware elements which communicate with the software model.

ID Run: Motor Identification Run. During an ID run, the drive attempts to measure properties of a connected motor model to ensure optimal control.

INU: Inverter or variable frequency drive. Forms AC voltage from a DC voltage.

MBT: Model-based testing. A testing approach based on a model of system operation.

MSB: Most Significant Bit. The bit with the largest value in multiple-bit data type.

PR: Pull Request. The process of taking changes from one branch to another can be facilitated with a formal event called Pull Request. This can be used for triggering branch policies for testing as well as reviewing the code.

PWM: Pulse-Width Modulation. Way to transfer information by modifying the on/off phase of a square wave cycle. Continuous waveforms such as sine can be approximated with PWM.

SUT: System Under Test. The target system of the model-based testing approach.

ULP: Units in the Last Place. A type of error bound for comparing floating-point numbers. By subtracting the integer representation of the figures, a difference between them in float-space can be established for comparison.

UML: Universal Markup Language. Visual modelling language used to visualise system designs. Commonly used in software engineering to map system overview.

VD: Virtual Drive. ABB internal software for simulating ACS880 electric drives.

VS: Visual Studio. Microsoft’s software development platform that contains support for multiple build tools and debuggers.

V&V: Verification and Validation. Aspect of software development where software is ensured to function correctly while also addressing customer needs.

1 Introduction

Integration tests are a way to inspect software component interactions by building a relational model of the software and its architecture. This model can be subjected to changes in parameters, events, and the environment, depending on the software and its interfaces. (Zander, et al. 2011.) In doing so, integration tests interface individually tested software components together to test and assess larger aggregates of functionality. Integration tests are used for finding and rooting out design flaws in the regression testing phase of software development (Mili, et al. 2015). This testing paradigm can expose problems with compatibility, dependencies, and interactions of software components. Lower levels of software tests consisting of unit testing verify the smallest singular variable- and function-level correctness while static code analysis ensures adherence to standardised practices. Integration tests interface these already tested software components together to evaluate more complex software interactions consisting of multiple functions and interfaces. (Zander, et al. 2011.)

The aim of integration tests is to form a feedback loop from the software operation to the developer. This feedback loop reduces the need to manually assess software operation, enabling automation for continuous integration (CI) and regression testing. Reducing the amount of required manual inspection saves the developer time from the work those manual processes necessitate. In addition, the automated processes have potential to be less prone to clerical errors than checking the software performance manually.

Electric drives are essential modern technologies that play a pivotal role in wide-ranging applications from power plants to industrial production. Frequency converters can convert power from one frequency to another, enabling devices designed for one frequency to operate with a power source of a different frequency. Electric drives, also known as variable frequency drives (VFDs) or adjustable speed drives (ASDs), are systems that control the speed and torque of an electric motor by varying the frequency and voltage of the power supplied to the motor.

Accelerating decarbonisation and electrification efforts across the globe create incentives to enable and maintain these processes efficiently and securely. Advancements in software and embedded control algorithms have enabled the development of electric drives in the last several decades in ways that would not have been possible without such technology. For instance, reliable control algorithms can simplify the hardware design by reducing the number of sensors needed to know the state of the drive. (Rik W, et al., 2011)

ABB is a global industry leader in variable-frequency drives, offering products for multiple drive topologies with market segments spanning a wide range of industrial and commercial applications. These drives include both inverter and line supply drives that are used, for instance, in power industry, robotics, motion services and manufacturing processes. ACS880 is a modern family of industrial single and multiple drive systems capable of acting as both a variable frequency drive, i.e. inverter (INU), and supply (DSU) configurations. These electric drive systems are equipped with a control board that acts as a regulator and monitor of the drive with embedded control algorithms. This software implementation allows precise digital control as well as parametrisation of the drive operation.

In modern electric drives, embedded software plays a role in control and communications functions that regulate drive operation, monitor system status, and achieve networking between clusters of devices (Rik W, et al., 2011). Software testing is important in ensuring deterministic performance of control algorithms and in ensuring they function within accepted tolerances.

This thesis introduces a framework of integration tests for ACS880 inverter drive software that has been previously used in ACS880 supply drive software testing. This previous adaptation of the testing framework was utilised for the inverter implementation as well. In addition, the shared components of the integration test system were harmonised between the inverter and supply software where it made sense. The goal of the project was to complement test coverage and optimise test feedback. Emphasis was placed on the need to separate integration tests from system level tests and unit tests. Likewise, the advantages of running the

same test framework locally as well as in CI pipelines were acknowledged. The state of the software and software testing before the project is presented in the current state analysis section of the thesis. This analysis highlights the motivation behind the thesis and the aim behind the introduced changes.

2 Background

The ACS880 electric drive control board holds an embedded system incorporating many interworking and interdependent software technologies that enable extensive parameter control of the drive operation. Multiple combinations of drive topologies, hardware platforms and specialised software form a diverse group of projects under the ACS880 umbrella. This versatile portfolio in combination with industrial grade reliability requirements necessitates robust software testing processes.

Software development for the ACS880 control unit includes multiple stages of regression and stability testing that validate the source code functionality and adherence to standards such as MISRA C2012. In addition, agile software development practices are incorporated including peer reviews and CI in accordance with feature-driven development. Development processes are facilitated by common toolsets that pool the requisite software tools in respect to the requirements of each project.

This section introduces concepts that are beneficial for understanding the work carried out in the project. These concepts range from software testing and its properties to software functions and electric drives. Software concepts and testing paradigms that relate to the project are given greater emphasis.

2.1 Embedded Systems

Embedded systems often operate in environments where failure can lead to significant harm or loss, like in transportation control systems or industrial applications, highlighting the importance of thorough testing (Broy et al., 2007). These systems are often referred to as critical embedded systems (CES). As CES have advanced in functionality and incorporate ever more sophisticated technology, their complexity also increases. (Cheddadi et al., 2022) This creates incentive to model the CES using a software model. Modelling the CES via software minimises external factors such as problems of hardware dependencies. (Cheddadi et al., 2022) Aiding in this, various software mocking tools have been developed. One such tool is the C++ Google Mock. Google Mock creates a dummy structure of the C++ program that helps the CES software run in isolation. This is often used in the unit testing of the individual functions but can be incorporated into higher level integration testing as well. (Cheddadi et al., 2022)

Additional complexity in embedded systems stems from real-time operation. This means that the program execution happens concurrently rather than consecutively. Concurrency introduces nondeterministic behaviour to the software execution, as external events can and will influence the internal functioning of the program (Parnas & Lawford, 2003). This makes embedded software testing more challenging, as edge cases and rarely occurring events need to be addressed to maintain necessary testing coverage.

2.2 Software Testing

Software testing in embedded systems is usually distributed among four levels of tests in increasing levels of complexity: low level testing, software integration testing, hardware software integration testing as well as systems integration testing. (Kokila, et al. 2016) To generalise this division, software testing can be thought of as consisting of three major pillars of testing: low-level tests, integration tests, and system-level tests. Low-level tests, such as unit tests, are simple in

terms of complexity but the most numerous of all test cases. Unit tests target function-level correctness of the software. Integration testing targets the software-software or software-hardware component compatibility. Lastly, at the highest level are system level end-to-end tests, which are assessing the total system operation in terms of inputs and outputs. (Kokila, et al. 2016)

Integration testing is a phase of testing that ensures individual software modules work together as intended when they are integrated together. Challenges include managing the complexity of interactions, real-time constraints, and hardware-software interactions, all while adhering to regulatory standards (Vardanega & Wellings, 2008). Tools and techniques such as simulators, emulators, and Hardware-in-the-Loop (HIL) testing are often employed, along with formal methods to mathematically prove the correctness of integrations (Kopetz, 2011).

The need to separate integration tests from the system level tests and unit tests can be thought of as the need to granulise the software targets of testing. Dividing a complex set of interactions into individual components and their interactions increases the visibility into software behaviour. While a robust model for unit tests forms the basis for secure and well understood software, the jump in complexity to end-to-end system-level tests is large enough that fault resolution is likewise more difficult. Thus, a robust model of integration tests can be seen as the basis for a system level test architecture.

2.2.1 Model-Based Testing

Model-Based Testing (MBT) is an approach to software testing where test cases are derived from a formal model that describes the operation of the System Under Test (SUT). Since a model of the SUT contains relevant interactions of the system, the interactions contained in the model can be subjected to testing. (Utting et al., 2011)

In MBT, a model is created that represents the expected behaviour of the system. This model can be based on several different formal representations of the

system such as state machines or Universal-Markup Language (UML) charts. Different formal models can be combined. An example of this could be Matlab, where models of embedded real-time systems can use a combination of data-flow notation in Simulink block diagrams and transition-based notation in Stateflow charts (Utting et al., 2011). However, the resulting model needs to accurately reflect the SUT, at least for the function the tests are targeting. Test cases are generated automatically by the model, where inputs and their expected outputs form a sequence of events that can be asserted against. This can improve test coverage since systematically deriving tests from the SUT model can yield test cases otherwise missed by testers (Utting, et al. 2011).

Automation of test case generation and execution reduces manual labour and can improve the efficiency of the test system. (Utting et al., 2011). This is beneficial in embedded systems where testing and formulating test cases can be resource intensive. The SUT model can serve as a valuable form of documentation that aids in understanding system behaviour, which can be advantageous for maintainability and feature development (Broy & Jonsson, 2010).

The model of the SUT holds information the behaviour of the system that is relevant for the individual test. Formulating such a model can be challenging for complex systems with numerous interactions. The effectiveness of MBT thus depends on the availability of robust tools for modelling and test generation. Additionally, embedded systems often operate under real-time constraints which can complicate the modelling further (Belli, et al. 2011). To accommodate this complexity, the models can become large making them more difficult to manage and understand. Ensuring that the modelling and testing processes remain scalable is important for test development and maintenance.

Model-Based Testing can increase test coverage and early defect detection because of the wide range of tests that the modelling approach can generate. However, the challenges such as model complexity, scalability, and real-time constraints might limit the applicability of this testing paradigm. Because of the

wide range of test cases that can be generated with MBT, it is a useful conceptual tool to form test cases for integration tests and simulated software environments. While MBT was not used when creating example test cases for the project, this attribute of the MBT approach shows promise in combination with an integration test framework and warrants further exploration.

2.2.2 Testing Pyramid

The testing pyramid is a conceptualisation of the different stages of software testing. The pyramid distinguishes between tests based on the examined interactions as well as the tools used for the tests and the quantity of individual test cases. (Cohn, 2009.) Illustration of a testing pyramid is shown in Figure 1. On the left side of the diagram, the arrows indicate how the complexity and number of test cases relate to the stages of testing.

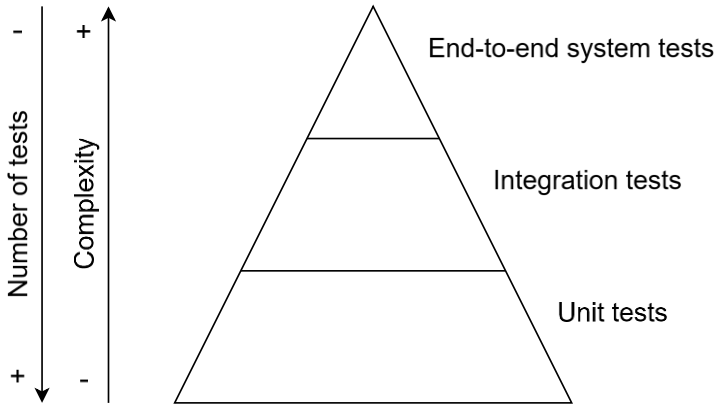


Figure 1: Software testing pyramid

Thought of as a pyramid, the basis of all software testing and the majority of tests are formed by unit tests that target a single software component such as function behaviour under specific conditions. Integration tests appear in the middle of the test pyramid above unit tests. Integration tests target software component interaction and compatibility with one another. Integration tests can be conceptualised as a superset of unit tests that interface single unit tests together to inspect larger software component interaction. Topmost, above integration

tests are the most architecturally complex and numerically limited test cases in system level tests or end-to-end tests that take a more holistic approach to software testing. (Lidwell, et al. 2010)

In the testing pyramid, both the complexity of the individual tests as well as the test coverage of an individual test grow towards the top of the pyramid. End-to-end system-level tests are the most time-consuming tests to develop and maintain as they cover the most interactions (Lidwell, et al. 2010). If the test configuration is too heavily focused on either end of the pyramid, it can affect the development and test processes negatively. Too many end-to-end tests can slow down the development feedback cycle and affect problem solving negatively, while too much low-level and not enough higher-level testing can reduce the view on introduced changes and affect testing coverage. Hence, a balanced approach where each segment of the pyramid is addressed with a suitable number of tests is beneficial. This results in the coverage of the software functionality from individual components to the end-to-end system behaviour.

2.2.3 White Box and Black Box Testing

An additional perspective into software testing is the concept of black box and white box testing. This conceptualisation divides software tests into white box and black box tests based on their scope and focus (Verma, et al. 2017). In order to maintain test coverage in complex projects, a combination approach to testing the utilisation of both black and white box testing techniques is beneficial (Cole, 2000).

In white box testing, also known as structural testing, the software under testing is known to the developer, so the internal structure and functions informs the design of the tests. White box testing can be viewed as deriving tests from the internal structure of the program. (Galler & Aichernig, 2014) Examples of white box testing include unit tests and integration tests. Unit tests verify software correctness on a function-level basis, while integration tests verify it on a component-level basis. (Williams, 2006.)

Black box testing, otherwise known as functional testing, tests the software operation in terms of its inputs and outputs. (Imperva, 2024) As the software is known only through its inputs and outputs, it acts as a black box with no visibility inside it. In black box testing the internal composition and function of the software can be assessed through the test results, but there is no view for the internals of the system as in white box testing. (Khan & Sadiq, 2011)

Black box tests can be divided into functional and non-functional tests. Functional tests assess that the functions and features of the software work as expected. On the other hand, non-functional testing assesses the software through its quality such as user interface, performance, or security aspects. (Imperva, 2024) Figure 2 illustrates the functional black box testing. It asserts that the output of the software correlates with the given input.

Figure 2: Functional black box testing

While the end customer often sees software through a black box, relying solely on black box testing is not enough. Black box testing does not offer visibility on internal software components, which makes the assessment of efficiency and correctness of specific implementations substantially more difficult. (Cole, 2000) White box testing is required to bring visibility to the internal components, their correct implementation, and properties. Thorough white box testing can prevent errors arising in higher levels of complexity in black box tests, which improves maintainability and development. Complex software projects incorporate both white and black box testing paradigms.

Integration tests are typically considered as white box testing as they require interfacing of specific internal software components together. (Williams 2006)

Tests targeting smaller granules of software functionality and form, such as unit testing and static code analysis are part of white box testing as well. The integration testing phase itself relies on the robust implementation of these lower-level tests. These lower-level white box tests target the internal implementation and functionality of the software. While static code analysis assesses the code's adherence to agreed-upon standards, the unit testing phase inspects individual software functions and their robustness. The integration testing phase can be thought of as higher-level white box testing, as it targets the interactions between software components.

2.3 Verification and Validation

Verification and validation are aspects of software development informing the design, testing and quality assurance lifecycles. Verification refers to assuring the correctness of a software implementation by verifying it for functional correctness and against regulatory constraints or other requirements. Validation is a process where the specification of the software is assured to meet customer needs specifically. (Pressman, 2010.) This process can be expressed simply as two questions that arise during the software development lifecycle (Boehm, 81):

> Verification: "Are we building the product right?"
>
> Validation: "Are we building the right product?"

Whereas the development process aims to create software that satisfies both verification and validation needs, testing can act as a tool for assessing the software from both perspectives. Hence, these perspectives are present in the larger software development processes to ensure that the final product meets the standards, required objectives, and set expectations.

The process of verification and validation (V&V) relates to multiple aspects of software development from testing to quality assurance cycles. This relationship can be expressed through the V-model of software development, where each stage of software testing from the lowest to the highest level corresponds to a

specification of the same level. Figure 3 illustrates the V-model in software development.

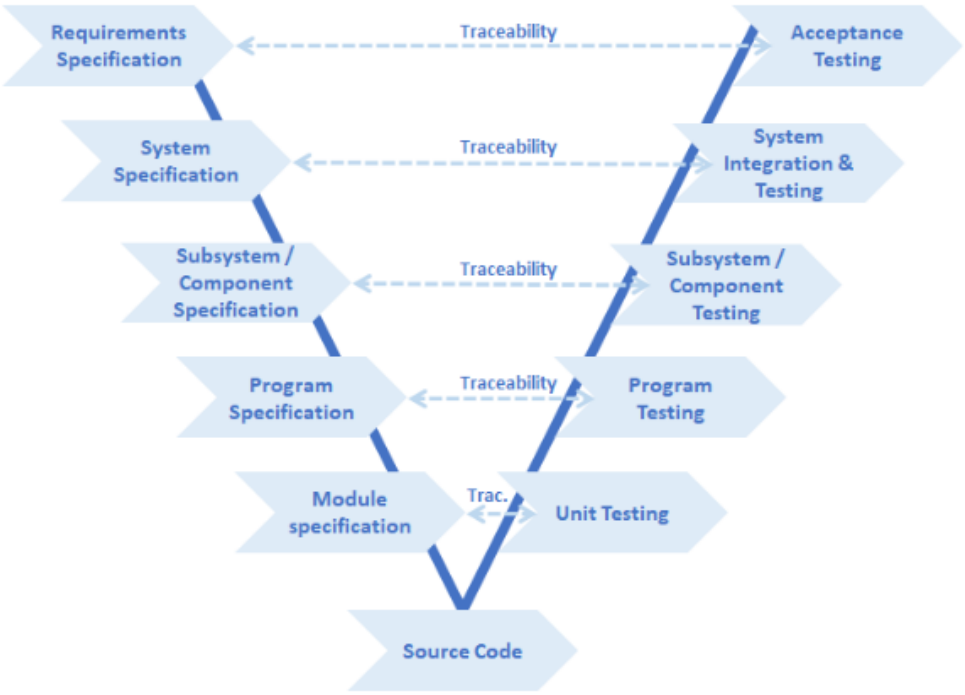


Figure 3: V-model in software development (N. Rajabli, et al. 2020)

While each level of testing correlates to a specification for the given layer of abstraction, the tests can be also seen as addressing the specifications of the higher levels, as each action of verification and validation in higher level of abstraction relates to earlier actions and specification in the lower level. (Pressman, 2010.)

2.4 Static and Dynamic Libraries

A static library is a set of compiled object files that are linked directly to the program at compilation time. When the program is compiled, the code from the static library is copied into a singular executable. This means that the code from the static library becomes part of the program, and the resulting executable

contains all the code it needs to run, loaded into program memory. (A. Ostrowski & P. Gaczkowski, 2021.)

A dynamic library contains compiled code that is linked to the executable program during runtime and not compile time. When the program runs, it loads the necessary code from the dynamic libraries into memory. (A. Ostrowski & P. Gaczkowski, 2021) This means that multiple programs can share the same dynamic library, saving memory because the code is loaded only once into memory and shared among all the programs that use it.

Static libraries can yield better runtime performance compared to dynamic linking as there is no overhead associated with loading dynamic libraries during program execution. (A. Ostrowski & P. Gaczkowski, 2021) In the case of integration tests, this can be advantageous as the test suites often need to be executed repeatedly and concurrently in a resource-constrained local environment.

2.5 Software Harmonisation

Problems in software development can often be attributed to problems in the core architectural model of the software. Failing to adhere to an architectural vision for the software can result in multiple problems during later stages of development, such as excessive redesigns, delays and decrease in software quality in terms of robustness or maintainability. In addition, the lack of a robust architectural model can result in implicit architectural decisions that are made based on local needs and less informed practices. This can result in several non-optimal design decisions to be implemented. (Hu, 2023.)

In software development, non-optimal design decisions can accrue over time into what is known as technical debt. Technical dept means solving a problem but having to deal with negative consequences later. (Suryanarayana, et al. 2014) More specifically, design dept can make the software architecture needlessly complex and increase the amount of labour required to manage it in the future. (Suryanarayana, et al. 2014.)

One aspect of software design where dept can accrue is generalisation. Ignoring sufficient generalisation can transform the hierarchy of the software architecture too wide or with unnecessary duplication. (Suryanarayana, et al. 2014) This “unfactored” hierarchy means that there is duplication in implementations or interfaces of types in the hierarchy that could be simplified into a supertype. (Suryanarayana, et al. 2014) Unfactored hierarchy can arise from copying of subtypes without addressing the commonality of the components. To address this, the software components that are shared as-is should be harmonised to a common source. (Suryanarayana, et al. 2014)

When common functionality is developed between two or more software components or projects, harmonisation or generalisation can yield improvements in maintainability and development time. Shared components and implementation improve maintainability by reducing the amount of software changes needed to realise a feature or adapt to a changing requirement. (Suryanarayana, et al. 2014)

On the other hand, harmonisation between software projects can lead to increased complexity, as it can increase checks in the codebase to differentiate functionality between the projects to allow special cases and different implementations. (Hu, 2023) Testing coverage needs to adapt to this increase in complexity when the same component is shared between multiple projects. However, this can also lead to a simplified adaptation process for new tests, as the different software components can utilise a shared, overarching test coverage.

2.6 Electric Drives

Electric Drives utilise the force experienced by a current-carrying conductor in a magnetic field to generate transferable force. (Drury & Hughes, 2019) When electric current passes through the conductor such as a coil in a magnetic field, this force generates torque that pushes the coil to turn. (Nave, 2024) This same principle can be utilised to generate electric current as well. When a conductor moves physically in relation to a magnetic field, an electrical current is generated through the conductor that can be used to power other devices. (Huang, et al. 2019).

Frequency converters or variable-frequency drives (VFD) harness electrical energy into motion or motion into electrical energy. Depending on the operating point of the electric drive, it may convert DC voltage into variable frequency AC voltage, or vice versa. (ABB Drives, 2024) Inverter units act as a rectifier circuit. This rectification can either create periodic waveform from a DC input or convert a periodic waveform into a DC output. The DC supply may be a line supply drive connected to an electric grid or an energy storage, such as a battery. (ABB Drives, 2024)

Figure 4 displays how the rotation of a turbine can be harnessed to generate electrical current. Depending on the type of the generator, either the conductor or the magnetic field itself is animated by the rotation of the turbine. (Huang, et al. 2019) In a DC generator, the magnetic field is static, and the conductors are rotated producing a DC voltage. In an AC generator that is pictured in Figure 4, the magnetic field is rotated in relation to the stator coils. This produces a voltage that varies as the north and south poles of the magnetic fields excite the electrons of the stator elements cyclically. (Huang, et al. 2019)

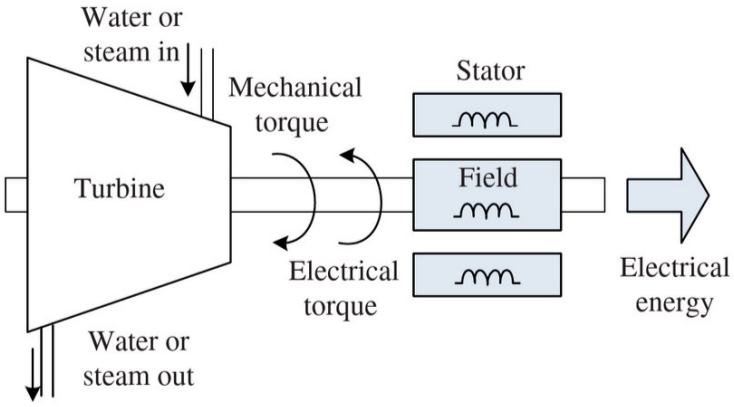


Figure 4: Turbine generator (Huang, et al. 2019)

ACS880 inverters control the speed or torque of electric motors by varying the frequency and amplitude of the voltage applied to the motor from a DC link voltage. The conversion from DC voltage into AC voltage is achieved by using pulse-width modulation (PWM). (ABB Drives, 2024.) Figure 4 outlines this conversion between DC supply and the AC output of the inverter driving a motor.

Diode supply units or active rectifiers are devices used to convert AC voltage into DC voltage. This is a common practice in power supplies that are connected to AC power outlets. In normal operation, diodes pass electrical current with very high resistance in one direction and close to no resistance in the other. This property of diodes can be used to rectify an AC sinusoidal or square waveform, converting the two-way moving voltage into a steady desired voltage.

Figure 5 pictures a simple VFD with feedback control. The feedback control of the speed of the motor is enabled via speed detector implementation. This can vary from measurement sensors to advanced control algorithms in modern VFDs.

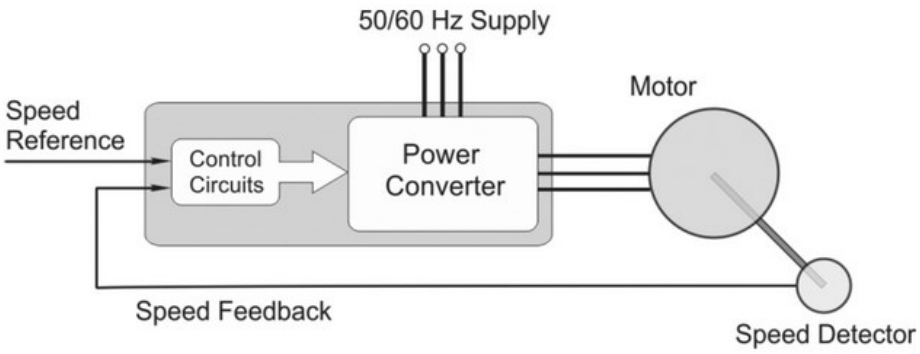


Figure 5: Variable-frequency drive with feedback control (Drury & Hughes, 2019)

The VFD pictured in Figure 5 takes AC current in its power inlet and converts it to a desired speed based on the control circuit. The conversion of AC to DC voltage happens by first rectifying the bipolar AC signal into a DC component, smoothing the DC voltage with a series of capacitors, rectifying this DC voltage into a bipolar form and finally smoothing the rectified signal into a sinusoidal waveform. (ABB Drives, 2024)

3 System Overview

This section introduces the ACS880 software and architecture to which the integration tests were incorporated. Attention is given to the integration test framework, how it is used in ACS880 supply software and how it interfaces with the rest of the software. The larger software architecture such as the virtual drive, continuous integration practices and the Google Test framework utilised by the integration tests are explored as well.

The integration test framework consists of several software components interfaced together to create a feedback loop from the firmware application to the developer. This feedback loop can show problems of the firmware with submitted changes, as well as help with the integration of several separately modified software components. Integration tests are incorporated into CI regression tests to validate software changes.

A final state and usage diagram of the system is illustrated in Appendix 1. It displays an overview of the integration test system emphasising the usage of the system and how the components act together when engaging the tests. The diagram displays the three major fields of the implementation, existing components, and how these components are used together as an integration test suite. Additionally, the diagram shows how the existing components and additions are used while running the integration tests.

The existing components in the Appendix 1 diagram include the VD wrapper and API. These components were used for the inverter software as they were present in the supply software. This meant they could be moved to a shared submodule location for harmonising the components of the VD integration tests. As can be seen in Appendix 1, the local build of the tests as well as the CI implementation utilise the same implementation for integration tests.

3.1 ACS880 Inverter Software Testing

Software testing in the ACS880 projects utilises tests in all the levels of software testing described in Section 2.3.3. The lower-level testing consists of static code analysis and unit testing. Prior to the work introduced in this thesis, integration testing was performed through an HIL setup consisting of a control board connected to a hardware emulator controlled by ATF. Finally, system level or end-to-end testing is performed much in the same way with ATF. These system level tests include hardware smoke tests and CPU load tests. Aspects of all these testing paradigms are part of multiple regression tests as well as tests for the stability of the software development tools and platform.

3.2 Virtual Drive

Virtual drive (VD) is an ABB software tool used for simulating ACS880 electric drives. VD runs the firmware code of the control unit and simulates the drive operation. The other simulated hardware is connected to the control unit. This includes sensors, plants, motors and grids among other components. The VD grants access to the parameter system of the drive, which can be used to formulate example tests.

The virtual drive is an internal software build that allows running the firmware of the control board locally on developer PCs. This emulation enables using the same firmware code across development environments and on the target hardware. While the simulated software of the control unit is running the firmware application, everything else meaningful to the operation of the drive, such as the connected motor, needs to be simulated with software. The virtual drive can be used as a tool to model the operation of the firmware under known circumstances and is likewise used in supply software integration tests that are part of regression testing. The virtual drive exposes a graphical user interface (GUI) that can be used to monitor parameters of the drive.

The virtual drive has hundreds of parameters that are exposed through a parameter interface, altering the operation of the VD. A virtual drive wrapper interfaces this software, allowing for interaction between the test cases and the simulation running the VD. This allows calling the same parameter system as is used in the real drive firmware program. The parameter system acts as a known entry point to the drive function, and this property can be used for test development.

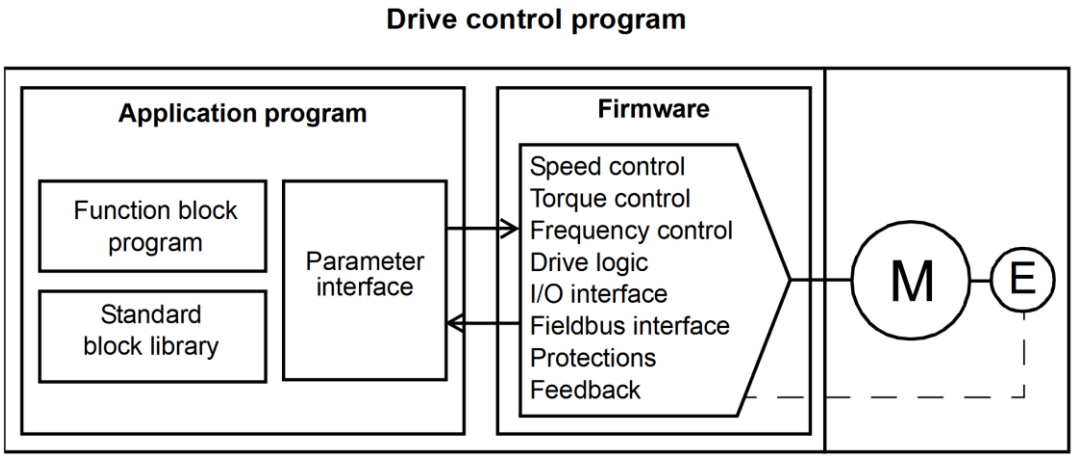


Figure 6: Parameter interface (ABB, 2024)

Figure 6 displays the parameter interface as it controls the drive firmware. The parameter interface acts as an interface between the application program and the firmware of the drive controller, which has control over the connected motor. An additional system that gives visibility to the processes of the VD is the event system. It is closely related to the parameter system with event handles signalling states of the drive. This can be used to check, for instance, external or internal faults or warnings and whether there are ongoing active events.

The simulator of the virtual drive allows monitoring of the virtual drive as if it was running the target firmware. The simulator enables monitoring the firmware application and the virtual drive under specific conditions, such as when the virtual drive is subjected to changes in parameters altering its operation. The simulator models the mechanics of the drive and can be run at a forced speed or

simulated load torque. When simulating in the speed mode, the load is the torque of the drive, and vice versa in the torque mode, the load of the system is the speed of the drive.

3.3 Motor Identification Run

During the motor identification run (ID run), the drive attempts to measure properties of the connected motor model to ensure optimal control. The ID run is engaged by a parameter handle that toggles the run. Initiating the run occurs by writing a positive toggle to the identification run mode handle. The motor properties measured by the ID run include stator resistance, direct-axis inductance, quadrature-axis inductance, and the permanent magnet flux of the simulated motor. These properties of the motor are measured by the drive during the ID run operation. During the ID run simulation, the drive should approximate the data present in the motor model database.

The motor identification run presents an optimal case for an integration test, as it is simple to engage by writing the ID run parameter, but the performing the run requires the integration of multiple software systems for the simulation and the VD to work. In addition, this simulation measurement data can be contrasted and compared against real-world measurements for the same motor model. This measurement data is contained in the motor database, consisting of real-world measurement data for specific motors.

For the inverter software, a simulator model interface was implemented to allow for changing the motor type of the simulator. This topic will be explored further in sections 5.3 and 5.5.2 of the thesis.

3.4 Google Test

Google Test is a C++ library for unit testing. While its original use-case is for unit testing, the library can be extended and modified to enable use cases outside of unit testing. This allows for inspection and assessment of more complex software functionality that is needed for integration tests. Even when the implementation relies on a more complex architecture, the individual asserts follow the same evaluation process that are used in Google Test -based unit tests. (Sen, 2010.)

The asserts that Google Test enables which were of most interest for the integration tests in the final year project included operational value comparisons, verification of string values and floating-point comparisons with tolerance allowances. The operational value comparisons consider the expected relation between the values (such as less than, greater than, equal, not equal). The string assertions consider the contents of the string (same content, different content, both either ignoring case or being case sensitive). (Cheddadi et al., 2022.)

Perhaps the most interesting of these is the float comparisons Google Test offers. The floating-point comparison functions EXPECT_FLOAT_EQ (val1, val2) and EXPECT_DOUBLE_EQ (val1, val2) use Units in the Last Place (ULPs) -based error bounds. This method of comparing floating point numbers removes the same-sign integer by subtracting the figures and establishes a difference between the figures in float-space. (Dawson, 2012.) The EXPECT_FLOAT_EQ and EXPECT_DOUBLE_EQ compare by default with 4ULPs of accuracy. However, EXPECT_NEAR (val1, val2, abs_error) allows for defining an absolute floating-point error allowed for the comparison. (Cheddadi et al., 2022.)

The Google Test framework enables advantages in comparison to other testing frameworks, such as direct support for the mocking platform Google Mock. In addition, the Google Test built-in assertions are deployable even in software where exception handling is disabled, which enables the assertions to be used in destructors. (Sen, 2010.)

3.5 Integration Test Framework

The integration test platform utilises a locally run virtual drive to interact with the firmware of a control unit. A wrapper for the virtual drive is used for the integration tests that allows the test cases to inherit some functionality from Google Test common API. To separate the functionality into different drive topologies or targets, the source code is first compiled into a static library.

The testing framework inherits the test structure, asserting logic and its supporting functions from the Google Test framework. A chain of inheritance follows from this shared test framework to the individual test classes. A simplified diagram of this inheritance is outlined in Figure 7.

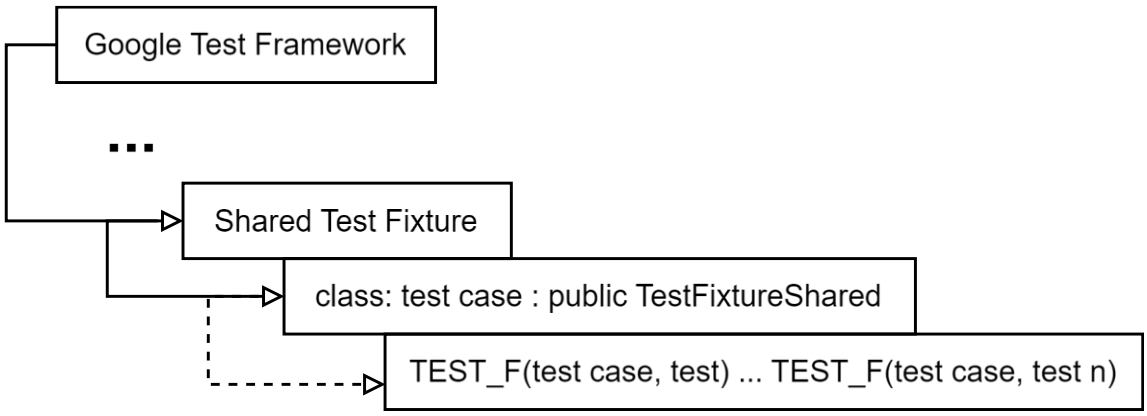


Figure 7: Inheritance of the test framework

The bottom segment in figure 7 contains the individual test functions inside the test case. The methods of the Google Test framework can be used inside the test functions, and in this way the test functions act as an implementation of the common test framework.

The basic structure of an integration test consists of test case source files written in C++ that are compiled and linked to a static library of the virtual drive source code. This linking uses a virtual drive wrapper to generate an executable for the individual test case. The virtual drive library is a static library of the source code of the target firmware. Thus, the executable is generated based on the source

code of the target, but individually for the test cases that allow for close monitoring and changing of environment variables or the state of the drive parameters.

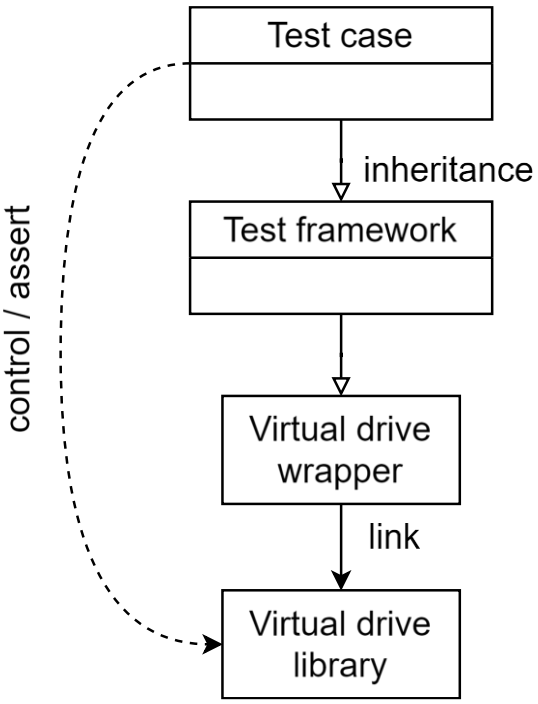


Figure 8: Integration test structure

Figure 8 shows how the test case source code .cpp file controls the VD, inheriting from a test framework that uses the VD wrapper to link the individual test to the static library.

The integration tests utilise a different build system from the virtual drive firmware build. These separate build systems optimise the build for separate aspects and generate a static library from the target firmware code instead of a regular executable. The system is optimised for speed and efficiency to reduce the amount of overhead created by waiting a build to finish during test development. The build system of the firmware optimises reliability and error logging that increase the build times in comparison to the integration test build even during no-change builds.

3.6 Continuous Integration

Continuous integration takes software assessment through testing as an integral part of the development process. To achieve this, branch policies are in place that target the introduced changes to a suite of regression tests. These tests can reveal problems in software operation and interfacing between several components. These regression tests incorporate integration tests among other types of tests together with build stages for the firmware applications.

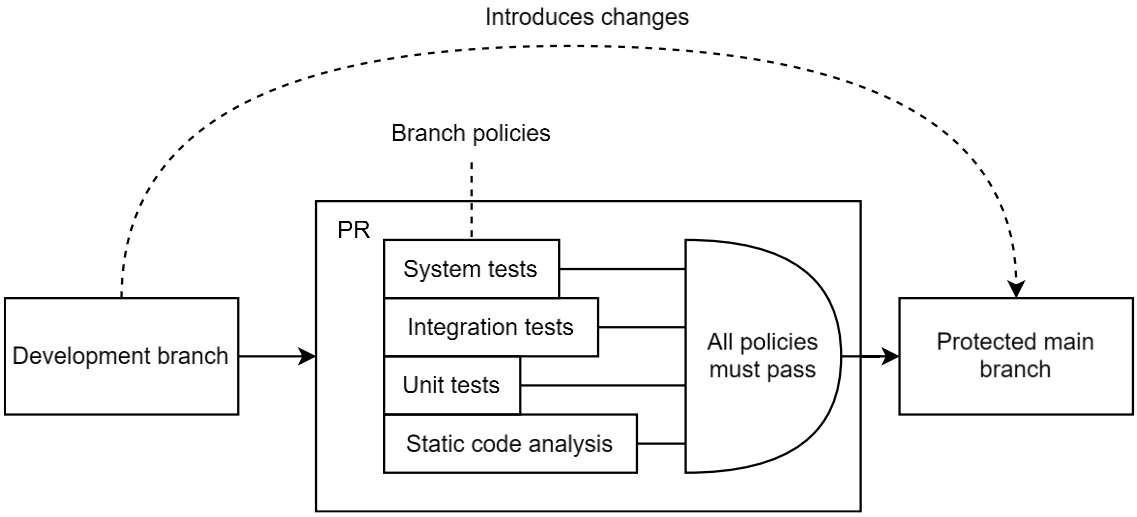


Figure 9: Pull request event with branch policies

Figure 9 outlines how branch policies protect a branch during a pull request (PR) event, where changes from a development branch are merged into a protected branch. Integration tests are efficient tests to increase test coverage of regression tests during PRs, as they combine different components and interfaces of the software together.

4 Current State Analysis

Prior to this project, integration tests have been performed as a component of the ACS880 inverter software testing using the system level Automated Testing Framework (ATF). These ATF tests run on target hardware in a laboratory test environment. While this approach is useful for assessing complex software interactions on real hardware, drawbacks include a slower feedback loop, increased test pipeline duration as well as complexity of test development that can impair test coverage. Utilising both ATF system level tests and a local PC or a server simulator model to engage in lighter software-only integration tests enables deepening the test coverage efficiently as the dedicated software integration tests can target more specific component interactions. The software integration test layer benefits test case development and reduces the duration of the testing feedback loop.

The ACS880 supply unit software uses an integration test framework written in C++. This framework is inherited from a common third-party testing framework Google Test, which is widely used in software testing. The Google Test library contains common test functionality such as various assertions for test case development. In addition, some common structures of the test are inherited from the Google Test such as test setup and teardown methods that the individual test implementation overloads. (Sen, 2010)

The variable frequency drives of the ACS880 family utilise extensive unit and system level tests for software development. The software projects for INU and DSU have developed into partially different testing strategies because of their internal structure and functions. Whereas the variable frequency drives have been traditionally more straightforward to test in ATF system level tests, the supply drives have benefitted more from a dedicated integration test framework.

In discussion with the case company, the goal of harmonisation to simplify the upkeep of the software between the INU and DSU projects was identified. Harmonising the shared code base makes introducing changes to the projects

simpler. Over time, the two projects have drifted apart in terms of build scripts and target specific implementation while significant similarity persists in the overall structure and build processes. Sharing common code helps condense the codebase and alleviates maintenance by reducing the number of identical changes required between the projects.

The existing ACS880 supply side (DSU) integration test framework is used as a model for the inverter side implementation. This allows for running integration tests locally on any machine or server with fast feedback times.

5 Implementation

This section describes the changes that were implemented to the ACS880 inverter software to introduce the integration test framework and initial integration tests. The system overview diagram in Appendix 1 displays three major aspects as parts of the implementation:

- Build scripts for creating static library (.lib) of the source code
- Setter for simulator motor data in control submodule
- Integration test fixture and test cases specific for inverter software

However, this is a simplified overview based on the usage of the integration test system itself and of the changes that were needed to run the integration tests in the ACS880 inverter software. The diagram does not account for every aspect of the work carried out in this thesis, such as CI pipeline modifications and the user manual. In its entirety, the implementation consists of the following items:

- Build scripts (1)
- Motor database model interface (2)
- Settings accessor of the drive model (2)
- Integration test script (1) (2)
- Test framework (1) (2)
- Example tests
- Jenkins pipeline scripts (1)

Furthermore, the changes can be divided into new additions, modifications of existing components and relocations of existing components of the supply side software that were refactored and moved to a shared location. Items denoted with (1) are modifications to existing components, while items denoted with (2) have been moved to shared submodules between the ACS880 inverter and supply repositories.

Shared sources and scripts were refactored and relocated to a shared submodule where it made sense to do so. This refactoring enabled running the shared software components in both the inverter and supply software, which harmonises

the architecture between them. To illustrate this, Appendix 2 displays the shared components of the integration test system and their relation to the build process of the integration tests. These shared components are in separate submodules contained in the two repositories, and they divide the components into scripts, tools and source code.

The shared source code submodule contains the supply software integration test virtual drive wrapper. The structure of the test fixture in the wrapper was refactored to allow for both inverter and supply usage. In practice, this meant switching the supply side specific naming to a general naming scheme and separating the library inclusions between the inverter and supply repositories. The additions to the script submodule contain build scripts used for building the VD source code as a static library, as well as integration test scripts for engaging the VD build and test suite. These scripts were refactored to enable use in both inverter and supply software.

Placing the shared components into submodules where both inverter and supply software has access to them harmonises the software and architecture between the two repositories. This harmonisation of the sources in part facilitates the future adoption and use of the test framework by keeping the structure of the codebase closer between the inverter and supply software. This eases the maintainability of the test setup by reducing the number of changes required between the repositories to adopt architectural changes. Test development can also benefit from a similar source structure by leveraging existing test cases from the other repository.

5.2 Build Scripts

The changes introduced to build scripts enable the building of a static library (.lib) of the target source code. The library build of the target firmware is a combination of all the source code that forms the firmware. This static library is used by the virtual drive wrapper of the integration test platform to link the individual test cases. Where the static library presentation of the source code differs from a regular dynamic executable is that the process takes all the sources and compiles a static library file as the output instead of an executable with dynamic memory. Figure 10 outlines how the regular VD build process generates an executable that can be used by the simulator environment.

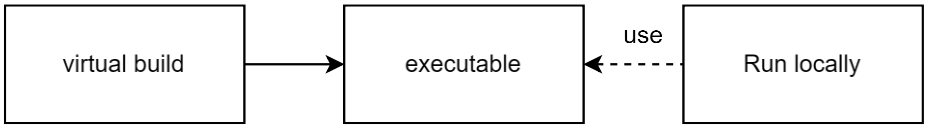


Figure 10: Regular VD build

The regular VD build generates an executable from the source code that enables developers to engage it in a simulator. This regular build is contrasted by a static library build that is required by the integration test framework. The static library build does not generate an executable itself, but a static library (.lib) file of the source code. The integration tests generate an executable by linking the integration tests case object files to the static library of the source code. Figure 11 displays how the integration tests generate the final executable by linking the test files to the source library.

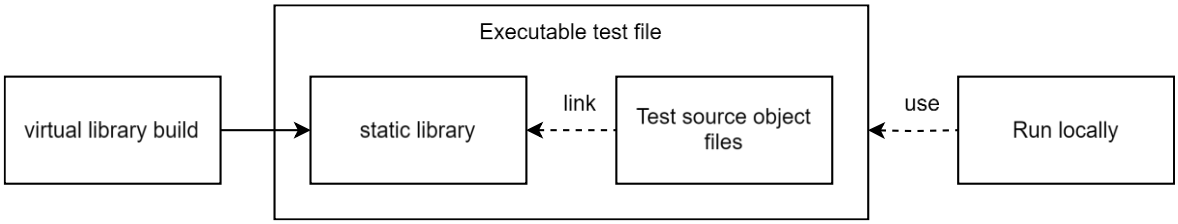


Figure 11: Executable test file

The executable generated by the integration tests can be run on a local simulator in the same way as regular VD, which allows for detailed monitoring of the software during the test operations. The difference with a regular VD executable is that the integration test files can now interface with the operation of the VD. This enables the test files to control of the simulator and collect data based on it. This data is used by the tests to assert that the VD is behaving within allowed variance from the expected values.

The builds for the inverter and supply firmware rely on different libraries and build processes. The build process has to accommodate separate implementations for regular firmware builds, virtual library builds and VD executable builds for both inverter and supply software. Primitives are used in the build process to separate the build processes between these targets. These primitives include tags for inverter, supply, virtual library and VD that can be checked against in the build script to enable different implementations between the targets.

5.3 Simulator Model Interface

Setting the VD motor data for the integration tests was implemented through a simulator model interface. As the default motor data are empty, they need to be set if the VD is used to run a motor model. To enable this, a simple interface was implemented between the motor database and the firmware application. This interface enables submitting data files to the simulator that alter the external properties of the simulated system. The model interface holds a function for setting the path to the data file used for the simulator that calls the simulator API. A relative path to the data files is passed through to the API by the interface function. The loading of the data file uses the built-in settings file loading tool of the simulator API.

The database file uses custom data types for many of the drive settings. These types are not defined in the scope of the integration test source code. This means the data points need to be casted to standard data types before they can be passed to the integration tests.

5.4 Integration Tests

The integration tests introduced in this thesis serve as guidelines and examples for ways to utilise the testing framework itself. While the test coverage at first looks centred on the parameter and event systems – considering the larger picture of the test coverage within each test case separately yields an additional perspective. This leads to the understanding that some tests, while interfacing with the parameter or event systems, have implications beyond strictly those areas. The integration tests are taken as active components of regression testing.

Applying the concept of white box and black box testing to integration tests introduced in this thesis, the integration test framework itself can be viewed as a black box component since it is ported from an existing implementation. The integration tests introduced can be viewed as black box testing for the limitations of this project. The focus of this project is on introducing the testing framework to the inverter software. In the project, the VD itself, excluding the parameter and event systems, acted as a black box. The interface to the VD is the parameter system of the drive that acts as a white box component for the test cases, as the test cases read from and write to it to perform the tests. The internal processes of the VD act as a black box and are only revealed through writing and reading the parameter system. The parameter system can be thought of as a window for the test cases to the otherwise black box of the VD system. This relation of the parameter system is described in Figure 12.

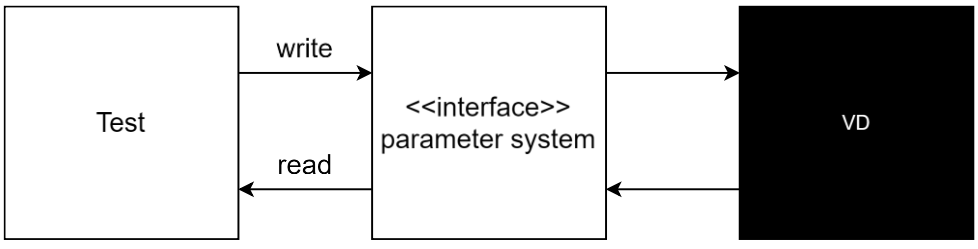


Figure 12: Integration tests as black box testing

There are other testing paradigms for conceptualising the integration tests beyond white box and black box testing. Inspecting the integration tests

introduced in this thesis through model-based testing (MBT) heightens the importance of the simulated application acting as a model of the system under test (SUT). While the introduced tests are not generated using a formalisation of the SUT, the MBT approach can be used as a future tool to facilitate more thorough integration test coverage.

5.5 Example tests

This subsection examines the integration tests that were incorporated as initial test cases for the integration test framework. The integration tests examine the virtual drive operation through the parameter interface and the event system. These tests include commencing a drive initialisation sequence, translating the binary data for firmware names contained in the parameters of the virtual drive simulation and an event vector test.

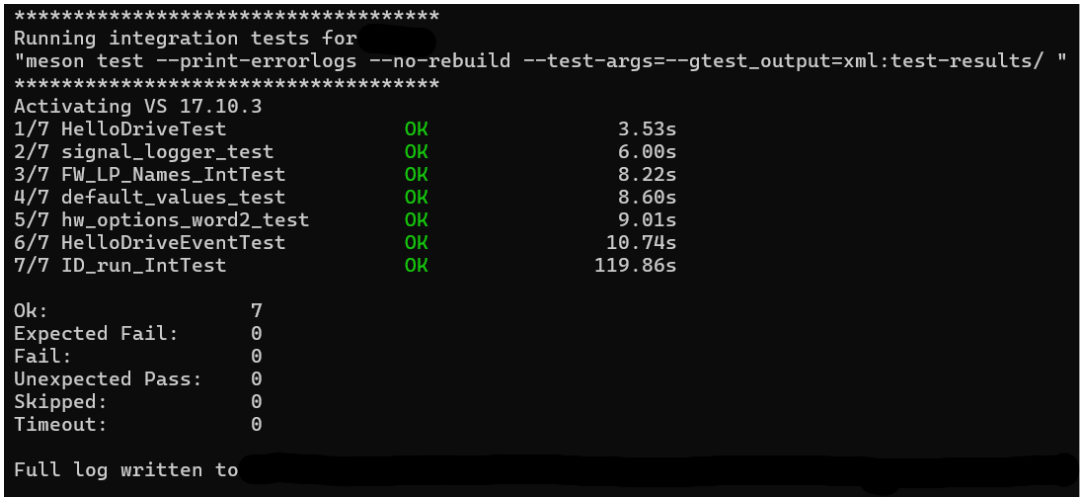


Figure 13: Console output of the integration test suite

Figure 13 displays the console output when engaging the integration tests through the command line interface. When issuing the command that generates and runs integration tests, the program first generates the static library for the target if it is not yet present in the repository. Afterwards, the static library is linked to form the test case object files, and finally the tests linking to the library are run.

Running the tests activates the VS build system to engage and control the VD operation.

5.5.1 Identification Run Test

The ID run test initiates the ID run for a permanent magnet motor by writing to the parameter interface. The parameter interface regulates the run via a parameter handle which forms the stop condition of the program. Timeout duration of 100 seconds is given for the run, which forces the program to end after the timeout. After the run has commenced, the motor model interface accesses the motor database for the expected values of the motor parameters. If the values correlate within accepted variance, the test passes successfully. If the values differ beyond the accepted variance limit, the test fails. In this way, the motor database forms the expected values that the simulation approaches during the ID run.

In the test, four data points are compared between the simulated values and the database entries for the motor. These include stator resistance, direct-axis inductance, quadrature-axis inductance, and permanent magnet flux of the motor. Tolerances for each of the four data points were identified through testing and set close to failure so that even low deviations are recognised by the test.

These motor properties that the ID run identifies in the simulator are set for the inverter software by a motor model interface introduced in section 5.3. This interface enables submitting data files to the simulator that alter these external properties of the simulated system. The data files are contained in a database submodule holding measurement data for different motor types. The model interface holds a function for setting the path to the data file used for the simulator that calls the simulator API. A relative path to the data files is passed through to the API by this interface function.

5.5.2 Firmware / Software Name Test

The firmware and software name test reads two tags for the software and firmware of the drive. These unsigned integer tags are accessible by parameter handles on the VD. The tags must be processed to a string format before comparison to the expected values for the target software and firmware package names. The handles in the parameter system for the firmware and software names hold 32-bit unsigned integer values that can be converted to a 5-character identifier for the firmware and software. These names are assigned to the VD parameters based on the firmware and software target names that the VD is running. Figure 14 displays an overview of the process that the test undertakes to convert the uint32 tags of the software and firmware into a string format.

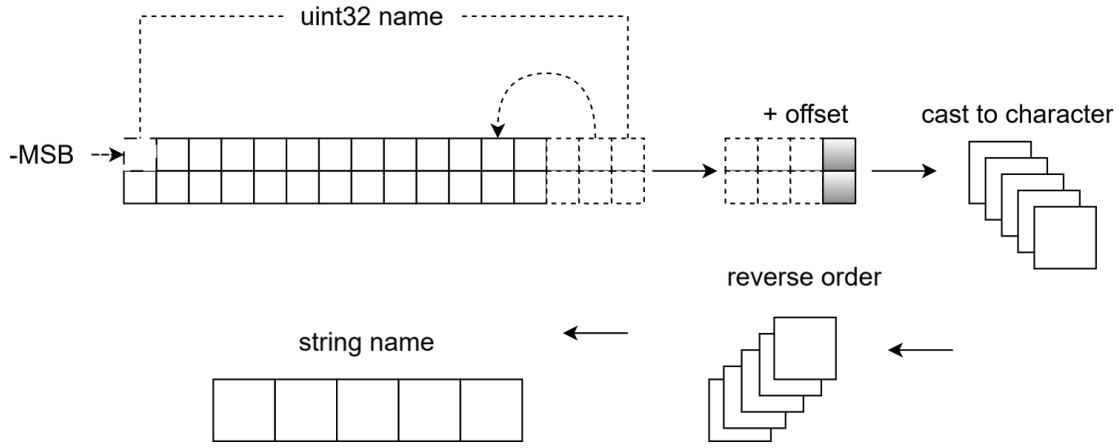


Figure 14: Name parsing from uint32 into string

The conversion of the uint32 values into string format begins by subtracting the most significant bit (MSB) from the binary value. Afterwards, the binary tag is processed sequentially in 6-bit increments, extracting the lowest 6 bits for processing. For each 6-bit segment, a predetermined 2-bit offset is added and this binary data is cast into a character type. After a single character is converted, the remaining binary data is shifted six positions to the right. This process carries out until five characters are processed. This results in the final string going from right to left, meaning that the last character is the first element in the string array. After the characters are processed and added to the string, its order can be

reversed by iterating from the last character to the first one to get the correct firmware and software package names.

5.5.3 Event Vector Test

The event vector test examines the event system of the VD to assert the correct operation of the drive. This is done by writing parameter handles for different software event statuses and checking that the correct events are active. There are differences in the parameter and event handles between the inverter and supply projects, which had to be considered when designing the test case for the ACS880 inverter software. In the test case code, the inverter and supply tests are separated by checking against the primitives outlined in section 5.2.

Appendix 3 displays a single assertion of the event vector test. During the test, an external fault is written into the parameter system, and an active source is written for the event. As can be seen in Appendix 3, even a simple assertion combines several different elements of the firmware, increasing test coverage.

5.6 Continuous Integration

Continuous integration was addressed in the thesis by introducing the integration tests as mandatory testing pipelines during multiple testing phases. These phases of testing include regression testing, main branch daily testing as well as CI stability testing.

Regression testing was implemented through branch policies protecting the ACS880 inverter repository main branch. Branch policies trigger on every pull request event where new code is being incorporated into the main branch. This means that during every PR, the CI pipeline will perform the integration tests introduced in this thesis as part of server-side regression testing. Briefly after the adoption of the regression testing branch policy, the integration test suite revealed an incoming fault in a virtualisation layer of the virtual drive which resulted in erroneous operation of the virtual drive. Notably, the integration test suite was crucial in revealing this as the software operated normally outside virtualisation.

In addition to regression testing, the daily testing of the main branch and stability testing ensure that the software and the testing environment are stable and functional. While the implemented tests are simple and do not fully utilise the capabilities of virtual drive testing, the CI architecture permits future additions to the test setup to run automatically. Once a new test is added to the suite, it will be incorporated into these testing phases automatically.

5.7 User Manual

To facilitate the continued usage and adoption of the integration tests as part of ACS880 inverter software testing, a manual for running, maintaining and developing the integration tests was created in the final year project. Confluence was used for this purpose, storing relevant information about the usage of the integration test framework. The manual discusses prerequisites for building and running the integration tests, modifications needed to run the simulation GUI as

part of the integration test, steps for making a new test case and common problems faced when setting up tests on a new setup.

The integration tests introduced in the thesis aim to introduce the test framework for continued use rather than form a rigorous test setup. The thesis aims to wrap the tests into a presentable package for the developers of the ACS880 inverter software. The manual and the included documentation serve the purpose of guiding new users to the test framework. The manual documents the steps required to get a new test setup running and addresses some common pitfalls during this process. New test development is encouraged in the manual by presenting ways of interfacing with the simulator via the parameter and event systems.

6 Conclusions

The aim of this project was to incorporate an existing integration testing framework used in ACS880 supply software testing into use on the architecturally related ACS880 inverter software. The goal was to have a functioning integration test framework utilising the virtual drive on the ACS880 inverter repository and continuous integration pipelines and to present initial test cases for the newly introduced test framework. The future adoption and use of the test framework was facilitated with a manual documenting the usage and development of the test framework. As changes were made to the inverter repository when the project progressed, an additional benefit of harmonising the sources between the two repositories was identified. Thus, by the end of the project, the scope of the work had widened from the initial estimate, with changes to the supply side and shared components as well.

The integration test framework was adopted for use in the ACS880 inverter software successfully. Three initial test cases were presented as part of the work, and this test suite was incorporated into continuous integration testing as planned. The requisite manual was written on the case company internal wiki in Confluence. Harmonisation was achieved for most of the integration test framework sources such as build scripts, the virtual drive wrapper and supporting Python scripts.

The thesis examines integration tests through model-based testing as well as white box and black box testing. An overview of these paradigms in software testing is presented. This perspective into software testing can facilitate future developments of integration tests. The ACS880 system is presented, with greater emphasis on the needs of the software development, testing and continuous integration processes.

Regarding the validity of the findings presented in this thesis, it remains to be seen whether the harmonisation of the shared components yields expected improvements in maintainability. However, the integration tests that were taken

into CI already performed their task in filtering the incoming changes. Briefly after their addition to the CI, the tests revealed problems in a PR introducing changes to the virtualisation layer of the virtual drive. As this was used by used by the testing framework to interface the VD, the tests prevented merging the changes. Thus, the integration tests have already proven their usefulness in expanding the test coverage of the ACS880 family inverter drives.

Further development of the integration test framework can, on one hand, be viewed from the perspective of the test cases and, on the other hand, the future of the testing framework itself. From the perspective of the test cases, approaches such as model-based testing explored in this thesis can prove fruitful as methods of further test case development. From the perspective of the testing framework, the harmonisation of the build scripts and components can be further monitored and improved. Whenever there is a significant correlation between software projects, harmonisation and separation are important perspectives to the development of the software architecture.

References

ABB Drives, 2024. ACS880 Firmware Manual, Revision C (2024-02-07) https://library.e.abb.com/public/6b0e2b9eba86e68bc1257b0c0053dab2/EN_ACS880_FW_Man_C.pdf Referenced 20.5.24

ABB Drives, 2024. What is a variable speed drive? https://new.abb.com/drives/what-is-a-variable-speed-drive Referenced 23.5.24

Belli, F., Hollmann, A., Padberg, S., 2011, Model-Based Integration Testing with Communication Sequence Graphs. Boca Raton, Florida: CRC Press

Boehm, B. W., 1981. Software Engineering Economics. New Jersey, Englewood Cliffs: Prentice-Hall

Broy, M., Jonsson, B., 2010. Model-Based Testing of Reactive Systems. Heidelberg, Germany: Springer Berlin

Broy, M., Krüger, I. H., Pretschner, A., Salzmann, C. 2007. Engineering Automotive Software. Proceedings of the IEEE, vol. 95, no. 2. New Jersey, Piscataway: IEEE.

Cheddadi, H., Motahhir, S., Ghzizal, A., 2022, Google Test/Google Mock to Verify Critical Embedded Software. Fez, Marocco: SMBA University

Cohn, M., 2009. Succeeding with Agile. Boston, Massachusetts: Addison-Wesley Professional.

Cole, O., 2000. Dr. Bobb's Journal, White-Box Testing. Trade journal publication, March 2000. London, UK: Informa PLC.

Dawson, B., 2012. Random ASCII: Comparing Floating-Point Numbers, 25.2.2012. Referenced 11.8.2024. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

De Doncker, R. W., Pulle, D. W. J., Veltman, A., 2011, Advanced Electrical Drives: Analysis, Modeling, Control. Dordrecht, Netherlands: Springer Dordrecht

Galler, S. J., and Aichernig, B. K., 2013. International Journal on Software Tools for Technology Transfer, vol. 16, iss. 6. Survey on test data generation tools: An evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java. Berlin, Germany: Springer Nature.

Hinkkanen, M., Harnefors, L., Kukkola, J., 2024, Fundamentals of Electric Machine Drives. February 2, 2024, ver. 1. Geneva, Switzerland: Zenodo repository. https://doi.org/10.5281/zenodo.10609166

Hu, C. 2023. An Introduction to Software Design: Concepts, Principles, Methodologies, and Techniques. Cham, Switzerland: Springer Cham. https://doi.org/10.1007/978-3-031-28311-6

Huang, Q., Khawaja, A. H., Chen, Y., Li, J., 2019. Magnetic Field Measurement with Applications to Modern Power Grids. New Jersey, Piscataway: Wiley-IEEE Press.

Hughes, A., Drury, B., 2019. Electric Motors and Drives, 5th Edition. Amsterdam, Netherlands: Elsevier Newnes

Khan, M. A., Sadiq, M., 2011. Analysis of black box software testing techniques: A case study. UAE, Dubai IEEE Conference 16.-17.10.2011. New Jersey, Piscataway: IEEE. https://doi.org/10.1109/CTIT.2011.6107931

Kokila, J., Ramasubramanian, N., Indrajeet, S., 2016. Advanced Computing and Communication Technologies. Singapore: Springer Singapore https://doi.org/10.1007/978-981-10-1023-1

Kopetz, H., Steiner, W., 2011. Real-Time Systems: Design Principles for Distributed Embedded Applications. Cham, Switzerland: Springer Cham https://doi.org/10.1007/978-3-031-11992-7

Imperva, 2024. Learning Center: Black Box Testing. https://www.imperva.com/learn/application-security/black-box-testing/ Referenced 11.6.2024.

Lidwell, W., Holden, K., Butler, J. 2010. Universal Principles of Design, Revised and Updated. Beverly, Massachusetts: Rockport Publishers LLC

Mili, A., Tchier, F., 2015, Software Testing: Concepts and Operations. Hoboken, New Jersey: John Wiley & Sons, Inc.

Nave, R., 2024. HyperPhysics. Atlanta, Georgia: Georgia State University. http://hyperphysics.phy-astr.gsu.edu/hbase/hph.html Referenced 13.8.2024

Ostrowski, A., Gaczkowski, P., 2021. Software Architecture with C++. Birmingham, UK: Packt Publishing

Parnas, D. L., Lawford, M. 2003. The Role of Inspection in Software Quality Assurance. IEEE Transactions on Software Engineering, 29(8), 674-676. New Jersey, Piscataway: IEEE

Pressman, R. S., 2010. Software Engineering: A Practitioner’s Approach, 7th Edition. New York City, New York: McGraw-Hill Education

Rajabli, N., Flammini, F., Nardone, R., Vittorini, V., 2020. Software Verification and Validation of Safe Autonomous Cars: A Systematic Literature Review. IEEE Access Volume 11, 2023. New Jersey, Piscataway: IEEE https://doi.org/10.1109/ACCESS.2020.3048047

Sen, A., 2010, A Quick Introduction to the Google C++ Testing Framework. Published 11.5.2010. New York City, New York: IBM developerWorks https://web.archive.org/web/20160316134644/http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html Referenced 20.7.2024.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells. Amsterdam, Netherlands: Elsevier https://doi.org/10.1016/c2013-0-23413-9

Utting, M., Pretschner, A., Legeard, B. 2012. A taxonomy of model-based testing approaches. Software Testing, Verification & Reliability, 22(5), 297-312. Hoboken, New Jersey: Wiley. https://doi.org/10.1002/stvr.456

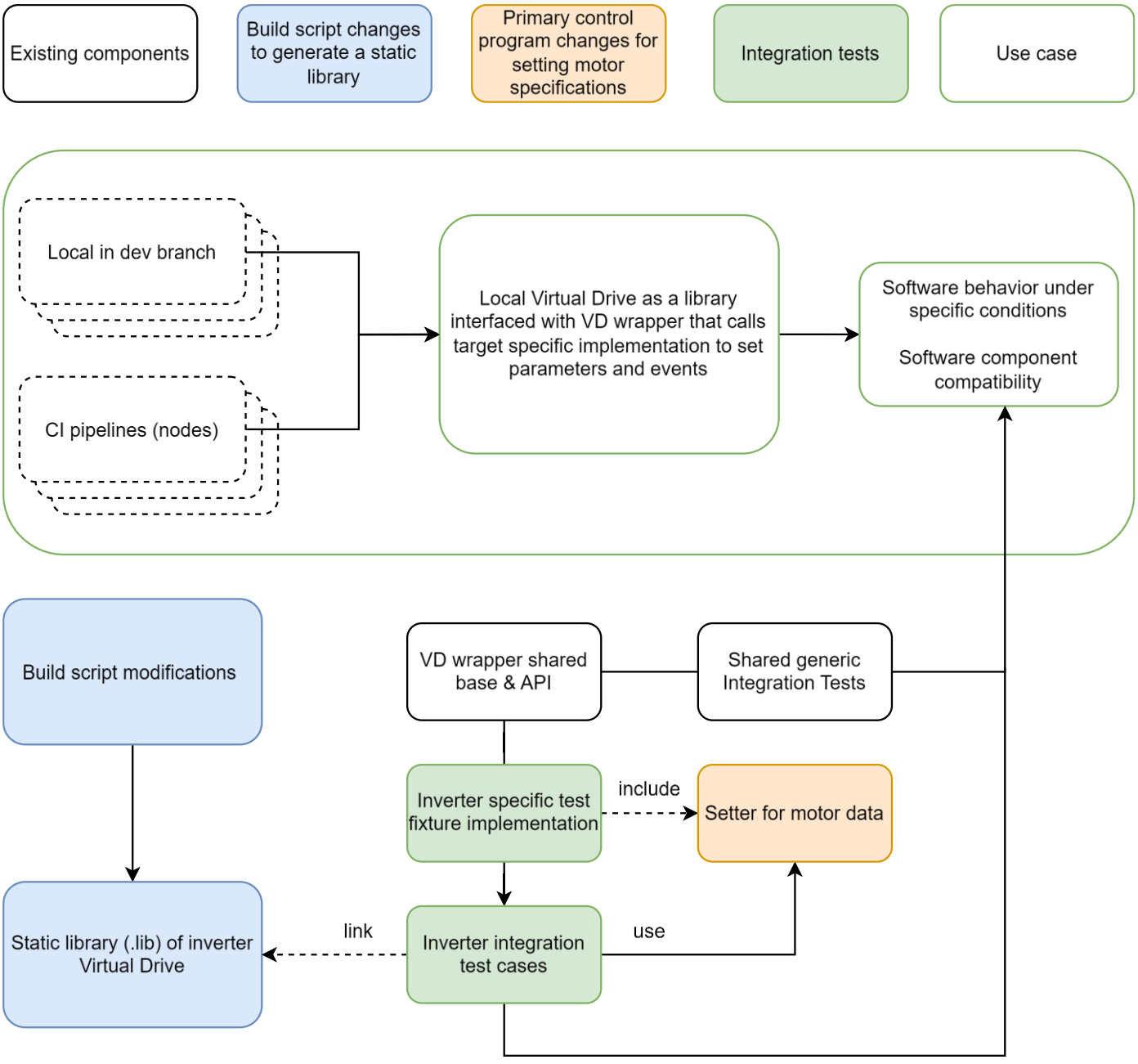
Vardanega, T., & Wellings, A. 2008. On the Value of Certifiable Real-Time Kernels for the Next Generation of Safety-Critical Systems. ACM SIGBED Review, 5(3), 8-14. New York City, New York: Association for Computing Machinery.

Verma A., Khatana, A., Chaudhary, S., 2017. A Comparative Study of Black Box Testing and White Box Testing. December 2017 International Journal of Computer Sciences and Engineering (IJCSE) https://doi.org/10.26438

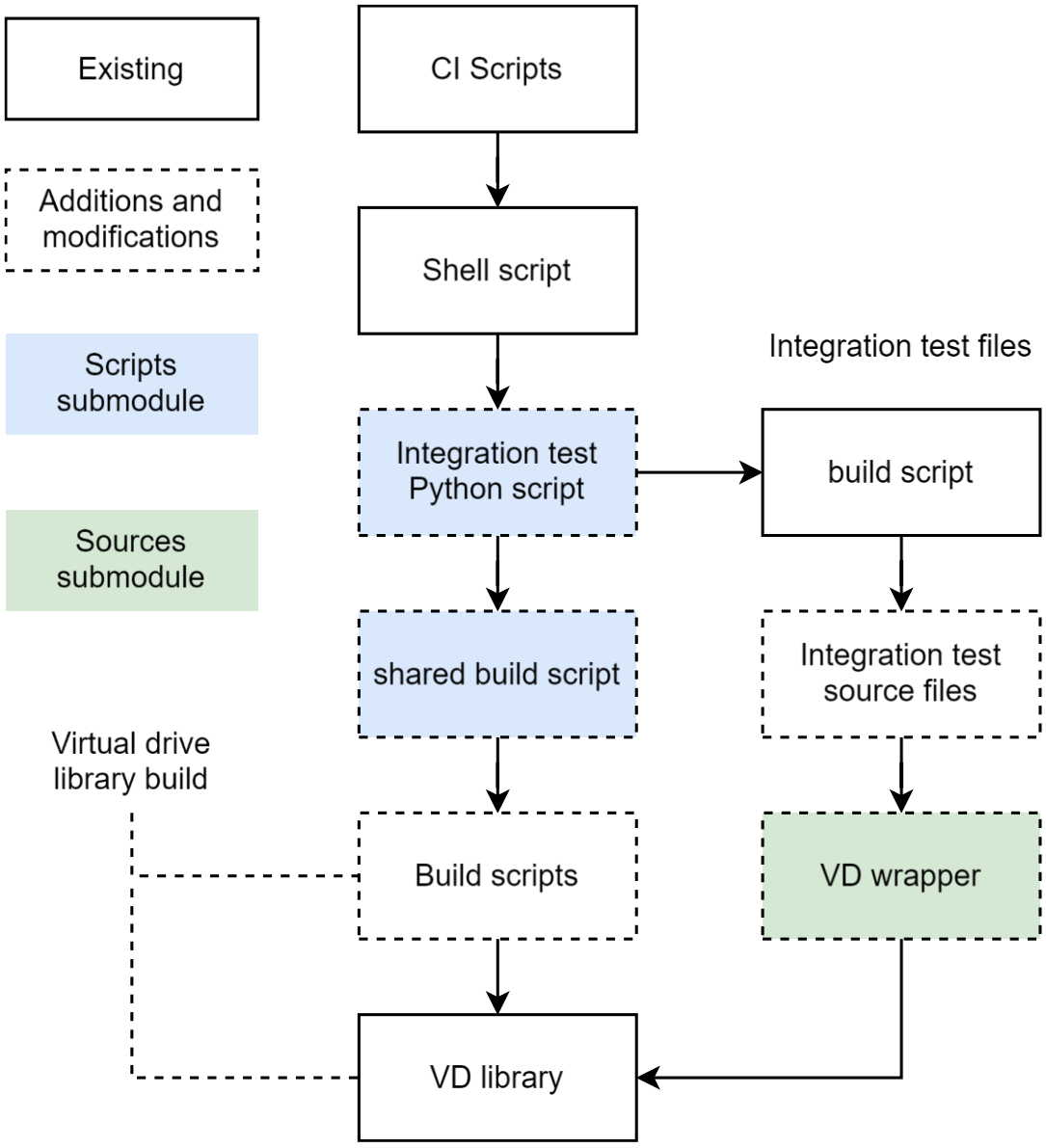
Williams, L., 2006. White-Box Testing. Educational material. Provo, Utah: Brigham Young University (BYU)

Zander, J., Schieferdecker, I., Mosterman, P. J., 2011, Model-Based Testing for Embedded Systems. Boca Raton, Florida: CRC Press. https://doi.org/10.1201/b11321

Appendix 1: Final State and Usage Diagram



Appendix 2: Shared Components



Appendix 3: Single Assertion of the Event System