Using Model-Based Testing for Manufacturing and Integration-Testing of Embedded Control Systems

Tobias Rauter, Andrea Höller, Johannes Iber, and Christian Kreiner Institute for Technical Informatics, Graz University for Technology
{tobias.rauter, andrea.hoeller, johannes.iber, christian.kreiner}@tugraz.at

***Abstract*— Implementing integration tests into to the manufacturing process of embedded devices is a crucial development for dealing with component deviations and production flaws. Especially control devices that interact with the physical world demand on a functional verification since malfunctions have a potentially enormous impact. In this domain, devices are often configured based on the customer needs during the production process. Different sub-components of the same product family are thus assembled into one single device. The high number of possible product configurations requires complex manufacturing processes. In this work, we use Model-Based Test (MBT) concepts to implement a manufacturing and test system that generates executable assembly- and test- procedures from an abstract test procedure model and a model of the actual manufactured device. We demonstrate how our approach helps in handling the complexity of the manufacturing process with an actual implementation in a productive manufacturing system for embedded control devices.**

I. Introduction

Functional tests and integration tests on component or system level are very important steps during the manufacturing process of embedded devices. Deviations of component properties or production flaws may lead to faults in the manufactured product. Modern devices rely on complex interaction between hardware and software, as well as on communication with the physical world[1]. Therefore, different components or sub-systems have to be integrated. Deviations that are acceptable for a single component may interfere constructively and lead to a broken system. To tackle these potential problems, integration tests have to be implemented throughout the whole production process.

Devices in the domain of distributed control systems are composed of a variety of sub-components. Since these devices are used to control the physical world, a verification of a safe function is even more important. For example, a control device may comprise a main computing board, as well as some digital I/O extensions. All of these sub-components share a similar basic structure (e.g., CPU, software images), but differ in specific details and extensions. They are thus part of the same product family. Moreover, the compound of these sub-modules (i.e., the control device) is configured based on the customer needs.

Both, product families and customization leads to a high number of possible configurations, each of these requiring a different integration test. This leads to complex manufacturing processes, especially when different component revisions come into play. Similar problems are tackled in the domain of software product lines [2]. However, these approaches cannot be applied directly to production systems, since they do not take into account hardware-harnesses such as additional test beds or automated test environments.

In this work, we show how Model-Based Testing (MBT) techniques can be used throughout the production process to generate assembly- and test-steps for different product configurations in an automated manner. Essentially, we use a generic production- or test-procedure model and combine these with test case templates, together with a model of the actual device or component. Based on these artifacts, we create an executable production- and test-procedure. Additionally, we built a tool that implements our method and demonstrate how it is used in a real productive manufacturing process for embedded control devices. With this implementation, we show how our approach is able to reduce the complexity of the configuration of the manufacturing process for both, product families and customization. Moreover, we show how our models apply to UML Testing Profile (UTP) 1.2 artifacts, in order to enable the integration of UTP tools in our future work.

The paper is organized as follows. Section II discusses related work and Section III describes the proposed system. Section IV and V discuss the feasibility of the approach based on an actual productive system implementation. In Section VI, the benefits and the drawbacks of the system, as well as future directions are summed up.

II. Background and Related Work

A. Model-Based Testing and UTP

Basically, MBT methodologies aim to generate tests at different levels (e.g., unit or system tests) from a model of the System Under Test (SUT) instead of implementing them manually. In the literature, the process of MBT is divided into five steps [3]: Generate a model (1) of the SUT and/or its environment; Generate abstract tests (2) from the model and concretize them to make them executable (3); Execute the tests on the SUT (4) and analyze (5) the results. Various methodologies and tools have been proposed that implement a subset or all of these steps in order tackle common test challenges in specific domains (e.g., [4], [5], [6]).

With the introduction of the UTP [7], MBT concepts are added to UML. In order to simplify different aspects of black-box testing, UTP defines some common artifacts in four concept groups: *Test architecture*, *test behavior*, *test data* and

978-1-5090-2817-7/16 $31.00 © 2016 IEEE
DOI 10.1109/DSD.2016.13

test time. The entire set of stereotypes is documented in [7]. Here we will focus on artifacts, that will be used in our work.

A *«SUT»* represents the actual tested system. It is modeled as a black-box with a public interface. *«TestComponents»* are parts of the test environment and are able to communicate with other test components or with the SUT. A *«TestCase»* refers to a behavioral description of a specific test. They are grouped within a *«TestContext»*, that may also include a *TestControl*, which schedules the execution order of the test cases. As a composite structure, a *«TestContext»* is also referred to as a *TestConfiguration*. This artifact defines the test components and their interconnections for the current test environment.

In this work, we generate the *«TestContext»* with its executable *«TestCases»* by specifying a generic *«TestControl»*, the *«SUT»* and artifacts that implement test case behavior, as also interface components to instrument the SUT.

B. Manufacturing Tests and Variability Management

Manufacturing tests gained a big focus in integrated circuits [8]. In this context, however, they usually target single products in mass-production. In [9], aspect-oriented programming is used to improve maintenance and re-use in the context of testing product-families. However, the authors state that especially in the embedded domain, tool support is crucial. While this is not necessarily the case for aspect-orientation, model-based technologies are widely used here. Concerning testing, variants in product families relate to variants in software product lines. Here, MBT technologies are widely used [2]. In this context, different technologies such as feature models [10], decision models [11] or orthogonal variability management [12] are used to model variability. In our work, we use a feature model as a specific view on the SUT model to identify components that are required to execute a test case.

III. Manufacturing and Test Environment

This section describes our approach to use model-based methodologies for system test and production. Section IV describes how the approach is implemented in an actual manufacturing system to present the application. A generic manufacturing process is refined to enable both, product families and customization. The overall manufacturing process consists of manual and automated assembly steps (e.g., installation of a software module or composition of different hardware parts), as well as functional test steps (e.g., functional test of the composed system). For simplicity, we denote both types of production steps as 'test steps' in this paper. This process is implemented in our tool, called Manufacturing and Test Environment (MaTE). The user provides a model that is able to describe all components of the SUT. For every single component, a set of test case templates can be defined. Based on a structural model of the SUT, the test case templates and a generic model of the test procedure, MaTE then generates and performs the actual executable test.

A. Production Process

As shown in Fig.1, MaTE builds upon a generic production process [13]: An operation is basically performed on a set of (sub-)components. The output of one production step is a new component. In our process, the output is a 'new' component C' even if the input only consists of a single component C. The operation may have changed the component's configuration or, at least, retrieved some information from it (e.g., the component C has passed all functional tests). The resulting component C' may be a completely manufactured device, as also an input for a following production step.

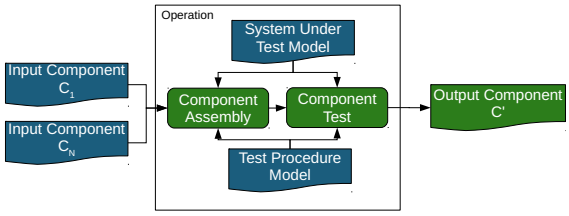

Fig. 1. In a basic manufacturing process, an operation procedure (e.g., assembly or test) is performed on one or multiple components. The result is a (new) component that may be the input for the next production step.

Since MaTE focuses on the production-test, the compound of the input components $C_1..C_n$ are termed SUT at the time the operations are executed. An input component may be

- a basic module, such as a single Printed Curcuit Board (PCB) or a software module that must be installed on a device,
- a composition of components,
- a complete system or device.

Different manufacturing steps must be carried out for all types of input components. An exemplary step that applies to (almost) all devices, is the 'mount on testbed'-step. Prior to any further test- or production-step, the device must be connected with MaTE. Basically, this is a manual step (i.e., the operator has to act), which must be acknowledged. This acknowledgement can be interpreted as the 'result' of the production step and MaTE is only allowed to continue the procedure if the result is positive. The production steps can be thus interpreted as test steps with operator interaction. Therefore, MaTE does not need to distinguish between production and test steps in MaTE.

Different types of tests (component test, integration tests and system-level tests) have to be executed based on the type of the input components. In order to handle variability, MaTE uses a test procedure model and a structural model of the SUT to configure the procedure to fit the instances of the processed components.

B. Test Environment Architecture

Fig.2 illustrates the architectural structure of MaTE. The overall test environment consists of the system under test (upper part) and the test framework (lower part). Basically, we have three stakeholders in this setup. The framework (MaTE) generates and executes the tests supervised by the operator. This is the person who actually performs the production steps. In order to use the framework for a specific set of produced systems, the user (i.e., the Original Equipment Manufacturer (OEM)) must provide a minimum configuration for MaTE.

This configuration consists of the test procedure and SUT models, as well as libraries that can be loaded to execute specific tests and interface with the actual devices.

As discussed below, the SUT is described as a hierarchical component model. Each *Component Under Test* holds a set of properties and refers to a *«SUT»* in the UTP context. Moreover, a *Component Under Test* may provide an interface (i.e., a serial connection) and contain other components. Since the *Components Under Test* always need some kind of instrumentation, the top-level component is always a *Testbed* that contains the actual tested device, as well as optional pure *Interface components* or *Test Data Generators* (e.g., a configurable signal generator). These components are used to communicate with devices that do not provide a communication interface by themselves (e.g., to flash a bootloader onto a new device).

The test framework contains a set of testing- and utility components. A *Test Result Database* is used to store test results, while the *Test Data Database* and the *Test procedure Database* store information that is used to generate the actual test for the target platform. Each test case in the behavioral test model (i.e., the test procedure) may refer to a specific *Test Case Implementation*. The *Test Generator* and the *Test Case Executor* are in charge to configure and execute the *Test Case Implementations* based on the current test procedure and SUT. In order to communicate with the SUT, a *Test Case Implementation* uses *Interface Components*. Currently, these components may form compositions to provide different types of communication channels: A command line interface of the SUT can be opened via a serial connection or a Secure Shell (SSH) on top of a network connection. On the other hand, a network connection may provide a basic TCP socket, as well as an XML-RPC interface on application-level. In a UTP-perspective, *Interface Components* and *Test Case Implementations* are *«Test Components»*.

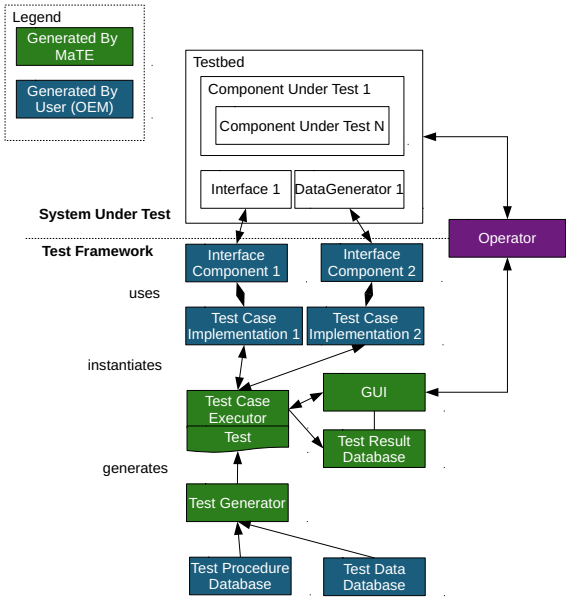

Fig. 2. The building blocks for the test environment used in MaTE.

C. System Under Test Model

As mentioned above, the SUT is represented by a hierarchical component model. Similar to other approaches [13], we identified a tree structure as a suitable system representation during the manufacturing process. The overall device consists of different sub-systems, which comprise different types of modules down to atomic components. These components represent completely different types of objects. A basic hardware part like a flash memory must be tested with the same infrastructure as a software module stored on this hardware, or a complete device. Therefore, the SUT-model must be able to serve this diversity. However, in our approach, many details of the components and their thorough interconnections are negligible: The framework itself only needs to know what type of component it is communicating with and what features and interfaces this components offers. Based on the component type and type specific features, the proper test case can be loaded and executed. Specific information about the tested component is hidden in the implementation of this test-case. Moreover, information about the communication internals are encapsulated by the feature-specific interface component. As a consequence, the SUT model does not need to provide in-depth information about the components and a list of provided features and properties is sufficient.

In our use case, we either create this model manually for each product or generate it with the help of reflection mechanisms of the devices. However, it would also be possible to automatically generate it from system models used in the development phase.

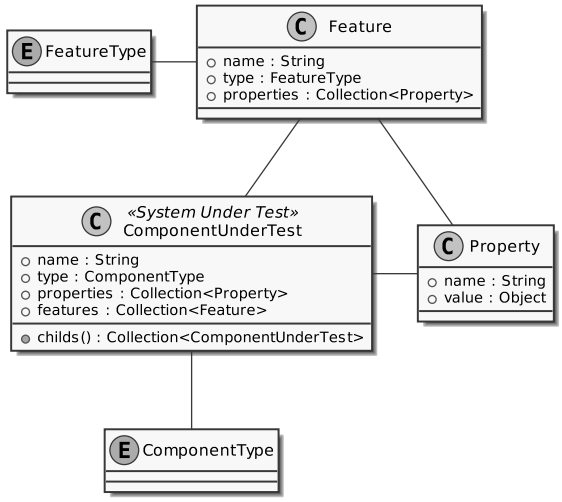

Fig. 3. The artifacts used to model the system under test. Basically, the SUT is described as hierarchical composition of different components.

Fig.3 shows the basic blocks used to build the SUT model. Each *ComponentUnderTest* has a *ComponentType* and a name. Based on the type, a component may have different properties (e.g., *start address* and *size* of a software image). These properties are typed name-value pairs that may have additional constraints (e.g., a property is mandatory or should be within a fixed range). Each *ComponentType* thus represents spe-

cialization of *ComponentUnderTest*. Components can define parent-child relationship with other components. In order to represent globally usable functions or interfaces, a component can provide a list of *Features*. Again, a *Feature* comprises a *FeatureType*, a name and type-specific properties. *Features* are used by test case implementations or interface components to communicate with the SUT or to instrument specific events on the device. Examples for features are a SSH-server, a digital I/O interface of a control device or a signal generator of the test-bed.

In order to use MaTE, the first step is to define the types of components and features, as well as the corresponding properties for the current product-family. This is done by the OEM that wishes to adapt the framework for the specific products. Basically, *ComponentTypes* and *FeatureTypes*, as well as their properties and associations have to be defined. Currently, MaTE supports a JSON-representation for this configuration. In Section IV, we will show this configuration for an exemplary and simplified system. With this configuration, the framework user (i.e., the OEM) is able to instantiate the SUT model. Again, this is done with a JSON-representation. Moreover, in the event that the SUT has some kind of reflection mechanism. MaTE can create the SUT model instance at runtime. This is useful when a lot of different configurations are possible. In this case the operator assembles the components and the SUT provides information about its configuration (i.e., which components used to are form the SUT) to MaTE. With this feature, MaTE is able to test systems without a priori knowledge of their configuration.

D. Test Procedure Model

The artifacts used for the test procedure model are shown in Fig.4. A *TestProcedureTemplate* represents the test control in UTP context. It contains a sequence of *TestSteps* that should be executed in order to complete the test procedure. The *TestProcedureTemplate* and *TestSteps* are very high-level concepts and do not have any connection to the tested device. An exemplary test procedure is the sequence of *Test Steps*, which are referred to as '*Assemble Device*', '*Initialize Device*' and '*Integration Test of all Modules*'. The procedure can be thus used for many types of devices, while the actual device-specific information is added by *TestCases*.

A *TestStep* consists of a name, a *TestStepType* and constraints. While the *TestStepType* is used to find proper *TestCases*, constraints are used to verify the generation of the test procedure. A trivial constraint, for example, is that there has to be a *TestCase* that executes the *TestStep*. The overall test should thus fail when the system is not able to find components that are suitable for the *TestStep*.

With this architecture, it is also possible to generate multiple *TestCases* for one *TestStep*. One *TestStep* may be executed for different components or a set of *TestCases* may be required to perform one *TestStep* for a specific component.

A *TestCase* represents a unit test case for one or more specific components. In order to enable the connection of *TestSteps* and components, a *TestCase* contains a list of supported component types, as well as the *TestStepType*, the test case is implementing. To execute a test case, normally a reference to a *TestCaseImplementation* is needed. Different test cases may use the same implementation, but with different configurations. Within a *TestCase*, it is possible to set some of the properties of the implementation. However, another possibility is to set a template value that will be replaced by a property of the component under test at test-generation- or execution-time. *TestCaseImplementations*, as well as other test components, contain a list of required features. For example, a serial connection to a specific software component is needed to execute the test. Again, the system has to check and satisfy these features before executing the test.

Similar to the SUT model, the OEM has to provide configurations and implementations for the test procedure model. He has to define the test step types and *TestCaseImplementations*. The *TestCaseImplementations* have to implement a plugin-interface, that is resolved by MaTE at runtime. Based on this configuration, the OEM is able to define actual *TestSteps* and *TestCases* for specific components. Again, this information is provided in a JSON-representation.

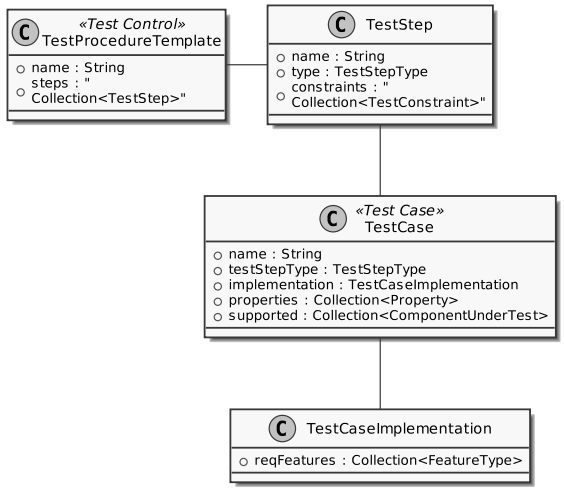

Fig. 4. The artifacts used to model the test procedure. An abstract test procedure with generic test steps is refined with component-specific test cases.

E. Test Generation

Based on the SUT model and the test procedure model, MaTE generates the executable test. To achieve this, the following algorithm has to be executed:

- Create empty test
- Enumerate features
- For all test steps:
  - Find test cases for components
  - Check test step constraints
- For all test cases:
  - Check feature requirements
  - Satisfy feature requirements (configure feature connections)
  - Configure property-templates (from SUT and feature model)

– Add to test

After the initialization of an empty executable test structure, the SUT model is loaded and all components are iterated and their features are extracted. This preparation step is done to enable a fast look-up later on. The features are arranged hierarchically by their type. The feature types *SSH-Server* and *Telnet-Server*, for example, share the same base-type *Command-Line Interface*. Therefore, both features would satisfy a test case implementation that requires the base feature. In addition to these *OR* relationships, MaTE supports 'requires' relations between features. *Feature A* may require the presence of *Feature B* in order to be usable.

The actual test generation starts with the iteration of all test steps in the test procedure model. In a first stage, all test cases that execute the defined test steps for any of the components in the SUT model are collected. Subsequently, all test step constraints are checked. Currently, two combinable types of constraints are supported:

- Min/Max N test cases of a specific test step must exist for each component ($N \in \mathbb{N}$). The trivial, hard-coded form of this requirement is that at least one *TestCase* that executes a given *TestStep* must exist.
- Test cases of a specific test step should exist for min/max M components ($0 \leq M \leq \#componentsintheSUT$). This constraint is used for tests, where the existence of *M* components of a specific type has to be ensured.

The generator is now able to iterate the test cases and instantiate all test case implementations. First, the required features are checked. The feature tree is traversed until a feature that satisfies the requirement is found. Here, some additional configuration steps take place: Based on the component linked to the feature (i.e., the component that implements the feature in the SUT model), some test data may need to be adjusted or additional test components (e.g., interface components that connect to a remote service) have to be loaded and configured. When all feature-requirements are satisfied, the configuration of property-templates is performed. As mentioned above, some properties may be set to template values in the test case specification. Here, this templates are resolved to actual values of the tested component. After the configuration of all test cases, the overall test is ready for execution.

F. Test Execution

The test executor sequentially executes all test cases according to the test procedure. At this stage, the greater part of the test cases are already completely configured. However, some test cases may require additional data depending on the current environment or specific features of the SUT. Therefore, the test executor is able to refine the test oracle of the given test cases. Technically, this is done by an additional iteration of the test-generator. Based on the result (test verdict) of each test case, the executor decides whether the overall test can be continued or should be aborted.

IV. Implementation and Use Cases

We implemented the system described in Section III for a distributed manufacturing system for Programmable Logic Controller (PLC) devices in order to examine the feasibility of our approach. These devices consist of different submodules that are in charge of providing I/O connections or specific computations. In order to simplify the use case description, we focus on the following sub-systems here:

- A central board (*Controller*) that is in charge for communication and control calculations,
- a digital I/O board (*DIO*) that is used to read and set digital channels,
- a analogous I/O board (*AIO*) that is able to measure electric currents and provide voltage in a continuous range,
- as also a base plate that contains a number of sockets to hold the previously introduced modules.

Every module has some similar components. For example, each module has some kind of CPU that runs a software image. However, the actual CPU and images, as also peripherals differ from device to device. Moreover, for each device, there exist different revisions with minor changes. Additionally, the configuration of the complete device (i.e., the presence and number of specific boards and their sockets) varies from device to device.

Therefore, this use case requires support for both, product families and customized devices. For a simplified use case, we show how MaTE handles these challenges and how model based testing techniques simplify the definition and adaption of production processes in the Cyber Physical Systems (CPS) domain.

A. Product Family

The first use case shows how MaTE is used to assemble and test product families using the example of the controller board. As mentioned before, all modules contain a basic set of components like the bootloader, the system image and peripheral hardware such as flash modules.

TABLE I
THE MODEL CONFIGURATION FOR THE GIVEN TEST SETUP. IN ORDER TO INSTANTIATE THE SUT MODEL, FIVE *ComponentUnderTest*-TYPES ARE NEEDED. FOR THE TEST PROCEDURE GENERATION, THE *TestCases* AND *TestSteps* NEED TO BE DEFINED.

Element Type	Element Name	Properties
ComponentUnderTest	TestBed	
ComponentUnderTest	Controller	
ComponentUnderTest	Bootloader	imgName, startAddr
ComponentUnderTest	SystemImage	imgName, startAddr
ComponentUnderTest	SpiFlash	addr, size
TestCase	MountBoard	text
TestCase	FlashImage	imgName
TestCase	FlashTest	address
TestStep	MountDevice	
TestStep	Install	
TestStep	MemTest	

1) User-Specific Model Configuration: In order to enable modeling of these components and devices, all component types, features and corresponding properties must be provided by the OEM. Moreover, the *TestSteps* and *TestCases*, as well as the *TestCaseImplementations* have to be defined and

implemented. Table I provides an overview of the element types which are required to define the tests for the current use-case.

2) System Under Test and Test Procedure Model: Based on this configuration, the user is able to instantiate the model for the test cases and the SUT. An example for this configuration is the definition of the component type '*Bootloader*', which is used in the SUT model in Fig.5. The type contains two properties: The *startAddress* of the bootloader on the device, as well as the file path (*imgName*) of the image file. Moreover, components of this type provide a command line interface that enables the installation of system images. This is represented by a feature *blTelnetServer* which is a sub-type of the feature-type *InstallInterface*. In our current implementation, all of this configuration work, as also the instantiation of the models is handled via a JSON-description. However, in our future work we want to build the transformations needed to enable the use of common UML or UTP tools for these tasks. Fig.5 shows this instance for a reduced system setup process. The SUT model comprises the test bed with a J-TAG-Interface, as well as the controller board. Moreover, the user defined some test cases and provided their implementation. For system integration, we use a simple test procedure, illustrated in the upper part of Fig.5, that asks the operator to mount the system, installs the software images and executes tests for external memory modules.

3) Test Generation: Technically, the user provided a JSON representation of the configuration, the SUT model and the test procedure model. Based on this information, MaTE generates the actual executable test shown in Table II. For each *TestStep* in the test procedure model, the *TestCases* are loaded and configured according to the SUT model. This results in five executable test steps. Note that the *Install* and *Memtest* steps in the test procedure model result in two tests steps in the resulting executed test since each of them can be applied to two *ComponentUnderTests*. In order to actually execute the test steps, MaTE has to ensure that the interfaces required for communication exist. This is done with the resolution of the feature requirements. To apply the same test procedure to another SUT that consists of the same type of components one has to update the SUT model, but there is no need to change anything else.

B. Customization

The second important use case in our system is customization. A completely manufactured device consists of a base plate, one controller board and an arbitrary number of digital or analogous I/O boards. Based on the customer needs, the devices are configured on demand.

1) User-Specific Model Configuration: In this use case, we use a similar SUT configuration since all boards consist of a controller, a bootloader and a firmware-image. However, they all have different firmware images and additional hardware components.

2) System Under Test and Test Procedure Model: Since the detailed configuration is not known before the actual test execution, the SUT model only contains the main controller component that is able to run the reflection service. The test procedure for this use case, consists of three test steps:

- Assembly: Similar to the first use case, this test step asks the operator to assemble the components. At this time, the SUT model consists only of the controller board (and the test bed).
- Read Configuration: This test step triggers a special test case that reads the component configuration (i.e., the connected components such as *DIO* or *AIO* boards) of the connected device. The SUT model is updated with the received information. All additional peripheral components are added on the fly.
- Integration Test: The test case that is in charge for integration testing is loaded for each component connected on the base plate. Since the SUT model changed in the last step, the calculation of the executable test is re-triggered and the integration test cases for all supported components are added.

3) Test Generation: Similar to the first use-case, the initially generated tests consists of an assembly step and the 'Read Configuration' step. For the 'Integration Test' step, no *TestCase* is loaded, because at this moment no supported component exists in the SUT model. During the execution of the 'Read Configuration' step, however, the model is updated and all existing components are added dynamically. Based on this updated SUT model, the test-generation is re-triggered and the integration tests for all components are loaded and executed. With a test-procedure that consists of three steps, MaTE is thus able to generate integration tests for all possible product configurations.

V. Discussion and Evaluation

In our real production system, MaTE handles 28 different components (including the controller board and I/O boards mentioned in the use case) and a variety of customized control devices based on these components (the base plate consists of 21 sockets). For each component, there exist up to four different tests (setup, test, calibration and integration).

A. Framework Overview

In the configuration of our industrial partner, the framework itself (including storage, GUI, test generator and test executor) is relatively small (23% of the overall codes base, which is about 19000 Line of Code (LOC)). 60% of the software are test components for different interfaces and automated test devices that are used to calibrate the different components, as also for test case implementations. The former 17% are used for SUT, test procedure and test case models which are currently stored in JSON-format. Although the storage format for the models is very inefficient in terms of LOC, we see that the majority (60%) of the produced software consists of test components and test case implementations.

B. Dynamic Test Generation

In order to evaluate the feasibility of the dynamic test generation approach, we analyzed the system after a first

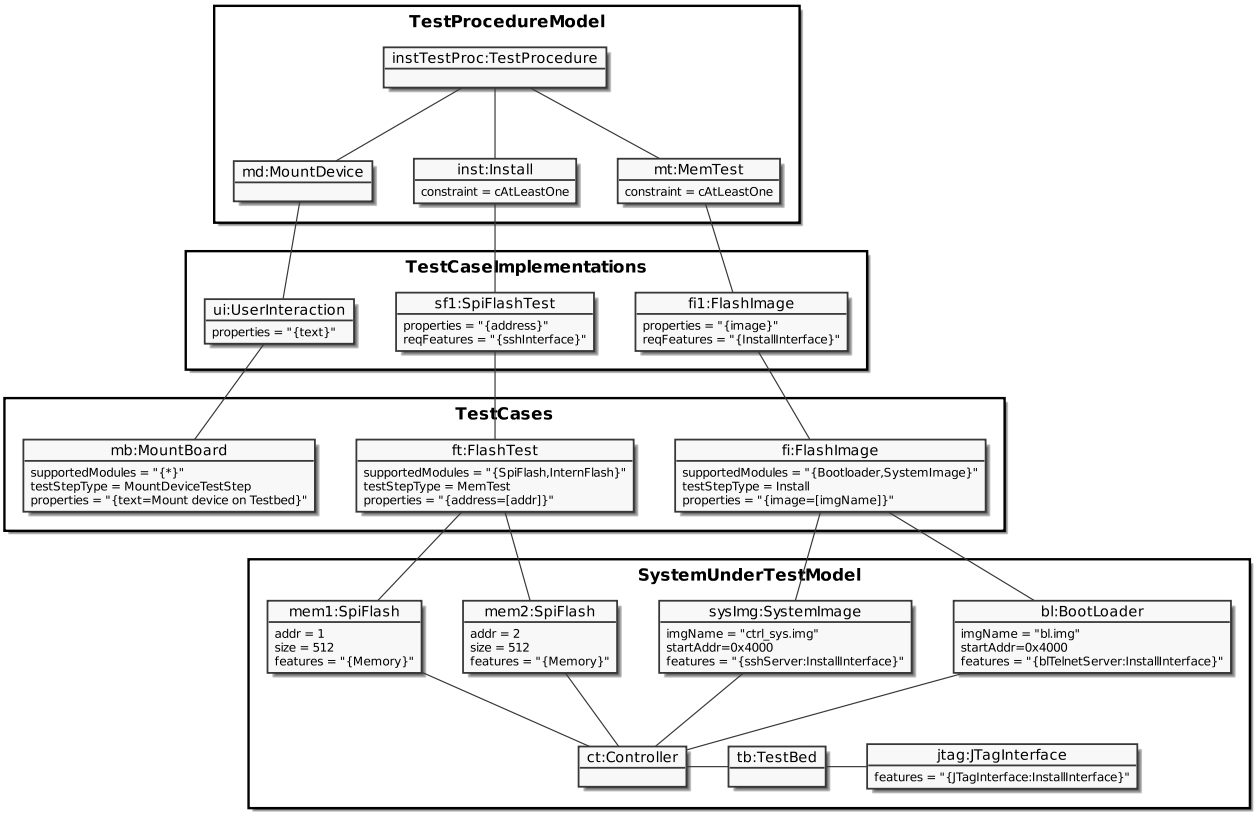

Fig. 5. An exemplar test environment for the presented use case. MaTE uses the SUT model to find proper test cases that implement the test procedure for the given device.

TABLE II
THE TEST STEPS OF THE RESULTING EXECUTABLE TEST FOR THE GIVEN SYSTEM MODEL.

Test Step	Test Case	Test Case Implementation	Feature Resolution (Components)	Description
MountDevice	MountBoard	UserInteraction		Asks the operator to mount the system
Install	FlashImage	FlashImage	InstallInterface=jtag	Install bootloader via JTag
Install	FlashImage	FlashImage	InstallInterface=bl	Install system image via bootloader
MemTest	FlashTest	SpiFlashTest	sshInterface=sysImg	Run test for SPI-flash module 1
MemTest	FlashTest	SpiFlashTest	sshInterface=sysImg	Run test for SPI-flash module 2

batch of devices has been produced. We provided the plain framework to our industrial partner and they provided the model configurations, instances and test case implementations needed for their manufacturing process. At the moment of the analysis several thousand entities of 19 different product types had been produced. We do not consider dynamic SUT models here, but only devices with a fixed configuration (e.g., the components that compromise a complete control device and fixed-configuration control devices). For each device type, one up to three different test procedures are performed.

Table III shows the implementation effort required by the OEM to configure MaTE for its needs: One SUT model is required for each device class. 11 out of 18 *TestProcedures* have been re-used at least once. Only 7 *TestProcedures* are thus completely device-specific.

However, the separation of *TestSteps*, *TestCases* and *TestCaseImplementations* needs to be re-evaluated. While there are indeed cases where this segregation is useful (e.g., calibration of different types of signals), many *TestStep-TestCase* connections turned out to be one-to-one relations. In such cases, the separation introduces overhead only. Here, more efficient solutions have to be investigated. Moreover, it turned out that the OEM only provided 14 *TestCaseImplementations* for 59 *TestCases*. On a first sight, this significant difference indicates complex *TestCaseImplementations* that perform varying tasks and thus violate the principle of single responsibility. However, only one implementation violates this principle (a component that is in charge for calibration of different types of I/O ports). All other *TestCaseImplementations* focus on one responsibility and the variance that is needed for different devices is encapsulated in the *TestCases*.

As discussed earlier, the *TestCaseImplementations* and corresponding interface components represent a majority of the code base. However, they only provided 'low level' actions

to the SUT and do not comprise any logic of the test and production process. While the processes will change over time with a high probability (e.g, more cost efficient, new devices), the requirements concerning the *TestCaseImplementations* are more stable. Additionally, the small number of *TestCaseImplementations* suggest a high re-use rate for different test procedures. Due to this diverse use of the same component in different test cases, they are reaching higher maturity levels faster compared to conventional systems.

On the other hand, the actual function (i.e., the definition of the test procedures) is very lean. Only 17% of the overall code base is used for this configuration. Especially in early deployment phases, where some components changed relatively often(e.g., new software or hardware revisions), the simple configuration of the system model supported quick adoption of the manufacturing system without changing implementation.

In summary, MaTE generated over 600 test cases based on 48 *TestSteps* and 19 SUT models. The OEM is thus able to focus on the quality of the single test steps and the overall process instead of implementing all test cases.

TABLE III
THE CONFIGURATION AND IMPLEMENTATION EFFORT FOR THE OEM. BASED ON RELATIVELEY LITTLE TEST CASE DEFINITIONS, MATE GENERATES 635 TESTS FOR THE 19 DIFFERENT DEVICES.

Type	Quantity
SUT Models	19
TestProcedures	18
TestSteps	48
TestCases	59
TestCaseImpl.	12
Generated Test Cases	635

C. Dynamic SUT Models

The support for dynamic SUT models, that change during the test-procedure enabled a simple system-level test of different customizations. The introduced feature-model enabled on-the-fly configuration of additional test-hardware (e.g., a digital I/O port is used to feed a signal generator which is used to test an analog I/O port) without a structural knowledge of the system at the test-design time. Components that provide features that are required to execute the test cases are located and configured for the given test case at run-time.

Dynamic models are not yet evaluated within the manufacturing process of actual products. However, currently these tests are done manually pre-deployment. Normally, this integration check consists of many repetitions of one very simple task (e.g., set I/O port an check output). When done manually, this task is time-consuming (thus expensive) and error-prone. In a first evaluation of a subset of these use-cases, we have seen a significant speedup of the process. Moreover, the only manual task that remains is a check on whether the reflection mechanism detected all the components. The operator thus has to compare the real device with a generated image of the device based on the model to detect, whether all sub-comonents are registered properly.

VI. CONCLUSION AND FUTURE WORK

In this work, we use MBT concepts to generate a manufacturing and test system for product families and customizable devices. Our system generates the test configuration and executable test cases from generic test procedures and a model of the actual system. Based on a product family for an embedded control device, we show that our system is able to improve maintainability and quality of the overall production process. This is achieved by a significant reduction of the pre-defined test case definitions. In our evaluated production batch, the OEM provided 19 test procedures that overall consist of 49 test steps and MaTE generates test procedures for 19 devices with more than 600 tests.

Currently, we simply configure our models with JSON-representations. Since we have already prepared MaTE for this, the next step would be a generator that enables the use of common UTP tools for test and system definition. Moreover, based on the same models, the test and calibration data gathered in the production process in a unified way can be used in the system-lifecycle to detect anomalies and defects. With the help of these common tools, we further plan to simplify the test procedure generation, especially in the context of the dynamic SUT models.

REFERENCES

[1] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, 2012.

[2] E. Engström and P. Runeson, "Software product line testing – A systematic mapping study," *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.

[3] M. Utting and B. Legeard, "Practical Model-Based Testing: A Tools Approach," nov 2006.

[4] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An Automated Model Based Testing Approach for Platform Games," in *International Conference on Model Driven Engineering Languages and Systems*, 2015.

[5] V. Panzica, L. Manna, I. Segall, and J. Greenyer, "Synthesizing Tests for Combinatorial Coverage of Modal Scenario Specifications," in *International Conference on Model Driven Engineering Languages and Systems*, 2015.

[6] P. Iyenghar, E. Pulvermueller, and C. Westerkamp, "Towards Model-Based Test automation for embedded systems using UML and UTP," in *ETFA2011*. IEEE, sep 2011, pp. 1–9.

[7] Object Management Group (OMG), "UML Testing Profile (UTP) Version 1.2," no. April, 2013.

[8] S. Mitra, S. Seshia, and N. Nicolici, "Post-Silicon Validation Opportunities, Challenges and Recent Advances," in *Design Automation Conference (DAC)*, 2010, pp. 12–17.

[9] J. Pesonen, M. Katara, and T. Mikkonen, "Production-testing of embedded systems with aspects," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3875 LNCS, pp. 90–102, 2006.

[10] K. C. Kang, S. G. Cohen, J. a. Hess, W. E. Novak, and a. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Distribution*, vol. 17, no. November, p. 161, 1990.

[11] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '11*, pp. 119–126, 2011.

[12] K. Pohl, G. Böckle, and F. J. van der Linden, "Software Product Line Engineering: Foundations, Principles and Techniques," sep 2005.

[13] K. Jørgensen and T. Petersen, "Product Family Modelling for Manufacturing Planning," *International Conference on Production Research*, 2011.