Technische Universität München - Fakultät für Maschinenwesen

Test Coverage Assessment for Semi-Automatic System Testing and Regression Testing Support in Production Automation

Sebastian Ulewicz

Vollständiger Abdruck der von der Fakultät für Maschinenwesen
der Technischen Universität München zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs
genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Markus Lienkamp

Prüfende/-r der Dissertation:

1. Prof. Dr.-Ing. Birgit Vogel-Heuser

2. Prof. Dr.-Ing. Stefan Kowalewski

3. Prof. Dr. Julien Provost

Die Dissertation wurde am 02.03.2018 bei der Technischen Universität München
eingereicht und durch die Fakultät für Maschinenwesen am 04.10.2018 angenommen.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Test Coverage Assessment for Semi-Automatic System Testing and Regression Testing Support in Production Automation

Autor:
Sebastian Ulewicz

ISBN 13: 978-3-96548-011-7
1. Auflage 2018

Copyright
© sierke VERLAG
Hermann-Föge-Weg 15
37073 Göttingen
Tel.: +49 (0)551 5036647

Coverdesign: sierke MEDIA

Contents

1 Introduction

In factory automation, so-called *automated production systems* (aPS(s)) have high requirements regarding availability and reliability (Vogel-Heuser *et al.*, 2015), as these systems typically run over long periods of time (decades) and system failures or incorrect behavior can dramatically increase costs regarding maintenance or the produced products. The volume and complexity of aPSs' software have risen substantially over the last decade (Vyatkin, 2013), exacerbating the problem of ensuring reasonable system quality. Even though model-driven engineering methods (Alvarez *et al.*, 2016; Estévez *et al.*, 2017), component architectures (Hametner, Zoitl and Semo, 2010) and approaches for distributed systems (Basile, Chiacchio and Gerbasio, 2013) have been proposed in research to manage the program complexity and reduce testing efforts, most industrial aPSs are still directly programmed in the standard IEC 61131-3 (IEC, 2003). The software's quality of aPSs is typically investigated and assured by testing. Apart from unit tests performed on single software modules in an early design phase, system tests of the integrated functionality of software and hardware are defined and performed in late phases of development, often as late as during on-site plant commissioning. In addition, aPSs are often subject to changes after the start of production, e.g. because of changed requirements by the customer, newly found bugs in the control software or wear on hardware components. During the implementation of changes, new faults can unintentionally be implemented. Thus, besides testing the changes themselves, so-called *regressions* of the system are subject to investigation. For this, regression testing is performed by executing previously successfully performed test cases again, to identify possible unwanted side effects of changes.

In several research projects and discussions with industry partners performed by the author of this thesis, it was found that test plans for system testing exist in most of the cooperating companies in the field of aPS engineering, yet the definition of the individual test cases is abstract and generic. On the one hand, this means that large parts of these test plans can be reused between projects. On the other hand, the individual test cases leave significant room for interpretation during the testing process. Additionally, tests are performed manually, as many functions are not related to the software alone, but to the integrated system comprised of mechanical and electrical hardware as well as software. Thus, many actions performed during these tests, such as placing intermediate products into the machine and visually verifying the correct product quality, cannot be performed fully automatically: Sensors and actuators that would enable automated testing are not available due to their cost. Instead, the test operator is required to perform these actions manually.

The personnel manually performing system tests in late phases of development are often subject to high time pressure, an uncomfortable on-site environment and the mentioned vague specifications. This results in three main problems.

Problem 1 - Lack of documentation and repeatability: Which test cases were performed and their findings are often documented in a very rudimentary way, making the testing process intransparent and hardly reproducible.

Problem 2 - Uncertainty of test adequacy: The adequacy of the performed tests to ensure the abstractly defined required functionality is often based on the experience and intuition of the test operator. Subsequently, the possibility of not testing critical behavior and thus overlooking critical faults in the system represents a realistic problem. In addition, the quality of the performed test cases themselves remains uncertain.

Problem 3 - Inefficient or inadequate regression testing: The testing process, especially after changes, is to be kept as short as possible, while assuring sufficient system quality. The difficulty and problem for the involved personnel are to identify and perform only relevant test cases for the implemented change under the difficult situation of high time pressure without any support by automated systems. This problem is exacerbated by the necessity to restart the regression testing process after fixing a regression.

In this thesis, an approach aiming at tackling these problems is proposed to increase the overall quality and efficiency of system testing for aPS in the domain of production automation.

1.1 Objective and Contributions

The objective of this thesis is to provide support for system testing for aPS in production automation. The main contribution of the approach presented in this thesis is subdivided into three sub-concepts:

1. *A guided, semi-automatic system testing approach for aPS* to structure the testing process, reduce deviations in testing quality and to provide a foundation for an improved documentation process and the following approaches of test coverage assessment and prioritization support for regression testing.
2. *A test coverage assessment approach for system testing of aPS* to provide support for the evaluation of test adequacy in the testing process. As the behavior of integrated aPS is largely dependent on the software, this approach aims at identifying uncovered (untested) code using coverage tracing. Thus, unintended omissions of testing system behavior can be revealed and evaluated by the tester.
3. *A test prioritization approach for system testing of aPS* to support testing personnel in identifying relevant test cases based on changes to the system and information about previously performed test cases. This part of the approach aims at increasing system regression test efficiency by performing test cases that have a high probability of unveiling introduced regressions into the system first.

Thus, for the first time, the contribution presented in this thesis is to provide valuable support in quantitatively assessing and increasing testing quality in fully integrated industrial aPS in industrial quality assurance scenarios.

1.2 Thesis Structure

The thesis is structured as follows. Chapter 2 (p. 5) provides an overview of the field of investigation and basic definitions. In Chapter 3 (p. 19), the domain requirements are presented, as derived from multiple workshops with industrial partners. Based on these requirements, related work in the field of production automation and adjacent domains are investigated and rated for their applicability in Chapter 4 (p. 27). In this literature review, a research gap for the regarded domain was identified. In Chapter 5 (p. 41), the approach developed in this thesis for the identified requirements and research gap will be presented. The approach consists of three main sub-concepts: a) guided semi-automatic system testing (chapter 5.2, p. 45), b) test coverage assessment for system testing (Chapter 5.3, p. 54) and c) regression test prioritization support (Chapter 5.4, p. 67). The implementation of the approach is presented in Chapter 6 (p. 77). In Chapter 7 (p. 81), the evaluation of the approach is presented, which was performed using an industrial case study and expert workshops. In that chapter, the designed experiments are regarded in detail, including measurement results, which were subsequently discussed with experts. The relation of the approach to the imposed requirements is subject to investigation in the last part of the chapter. Following the evaluation, the most limiting property of the approach – its runtime overhead – was optimized and its scalability was estimated. These findings are presented in Chapter 8 (p. 99). The thesis concludes with Chapter 9 (p. 103), in which a summary of the achieved results and an outlook on future research is given.

2 Field of Investigation

The presented approach was designed for industrial application for automated production systems (aPS) in the domain of production automation. To allow for a better understanding of the domain requirements, required theoretical background for the field of investigation, i.e. aPS, testing and static analysis, will be presented in detail.

2.1 Automated Production Systems in Production Automation

Automated systems can be described as autonomously working technical systems, e.g. a ticket vending machine. *Process automation systems* are a specialization of automated systems, dealing with automating the control of arbitrary technical processes (Lauber and Göhner, 1999, pp. 5–6). *Automated Production Systems* (aPS) are process automation systems, which are controlling production processes.

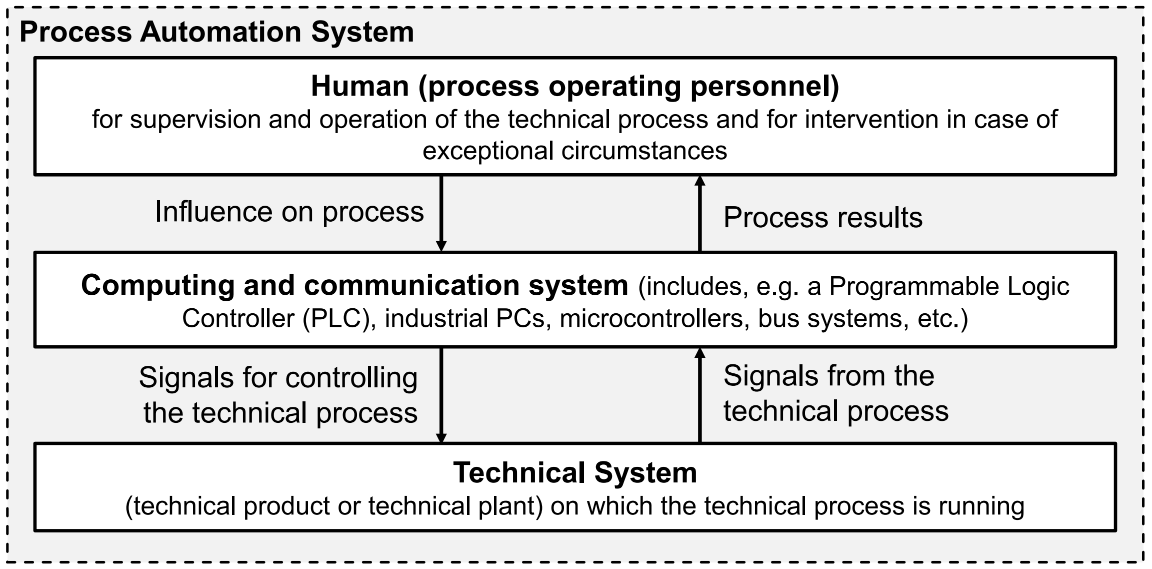

Process Automation System
Human (process operating personnel)
for supervision and operation of the technical process and for intervention in case of exceptional circumstances
Influence on process
Process results
Computing and communication system (includes, e.g. a Programmable Logic Controller (PLC), industrial PCs, microcontrollers, bus systems, etc.)
Signals for controlling the technical process
Signals from the technical process
Technical System
(technical product or technical plant) on which the technical process is running

Figure 1: The schematic structure of a process automation system (translated from (Lauber and Göhner, 1999, p. 7))

As shown in Figure 1, a process automation system consists of a controlled *technical system* (bottom), a *computing and communication system* (middle) and *human operators* (top). These parts communicate in a bi-directional manner by exchanging signals: sensor and actuator values are used to observe (sensors) and control (actuators) the technical system by the computing and communication system. The human operator is usually able to influence and observe the process via a *human machine interface* (HMI), such as a touch-sensitive display. Ideally, the operator will only interfere with the system in case of unexpected or exceptional circumstances during operation, yet during commissioning and maintenance processes of an aPS, the HMI is generally heavily used for testing. In addition, the personnel experienced with programming will also often connect to the computing system with an engineering PC, which will allow for manipulating and changing the control program.

In aPS, the computing and communication system is usually represented by a *Programmable Logic Controller* (PLC) or *embedded PC* (also called *industrial PC*) and bus systems, connecting sensors and actuators to this computing system. The majority of PLCs and embedded PCs is programmed with the programming standard IEC 61131-3 (IEC, 2003). While the third edition of the standard has been introduced in 2013 (IEC, 2013), allowing for new object-oriented elements, this new version has not been adopted by all PLC manufacturers yet and is mostly used for software libraries. Therefore, the approach presented in this thesis was developed for the common parts of the second and third edition, not focusing on the object-orientation.

In the following, the special properties and differences of this programming standard in comparison to regular desktop software will be presented, followed by a more detailed definition of the technical process.

2.1.1 Control Programs developed with the IEC 61131-3 Standard

In regular desktop software, programs are usually designed to have a beginning and an end. As control programs written in the IEC 61131-3 always need to work with recent input values, these types of programs are executed in a specific loop, the so-called PLC scan cycle. As shown in Figure 2, this cycle consists of reading the current process inputs (e.g. sensor values read from the bus system), executing the control program and writing the newly calculated process outputs (e.g. actuator values written to the bus system). This cycle usually begins as soon as the controller is started and is usually only stopped if the controller is turned off. Depending on the inputs and internally stored values, different parts of the control program can be executed.

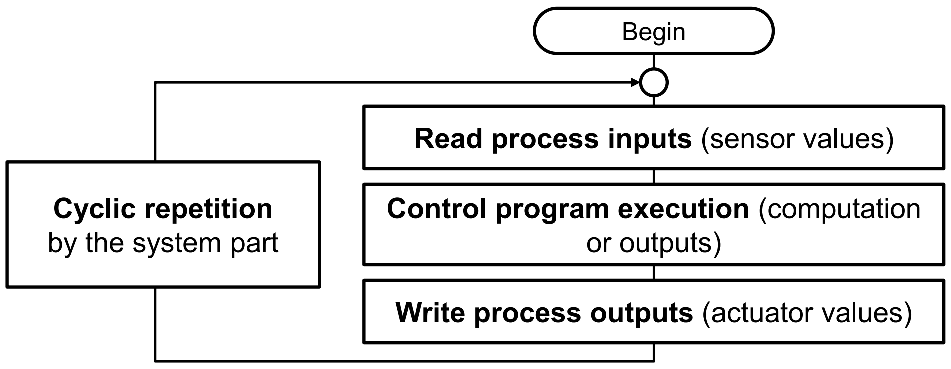

Begin
Read process inputs (sensor values)
Cyclic repetition by the system part
Control program execution (computation or outputs)
Write process outputs (actuator values)

Figure 2: Schematic of the standard PLC scan cycle: Inputs values are read, computation is performed, output values are written (based on (Lauber and Göhner, 1999, p. 281))

For application in aPSs, it is important for this scan cycle to fulfill real-time requirements. This means that each repetition shall not last longer than a specified maximum PLC scan cycle time (typically in the range of 1-100 milliseconds) to be able to control the technical process in a satisfactory manner.

The IEC 61131-3 standard only defines the syntactic composition a correct program should have. As it does not cover all areas of PLC programming, such as the Integrated Development Environments (IDEs) or the interfaces of editors and compilers, exchange of PLC programs for different PLC vendors can be non-trivial (Neumann *et al.*, 2000, p. 26). For this reason, the PLCopen (PLCopen, 2017) defines a uniform exchange format for different vendors. In this thesis, the uniform structure of the PLCopen is used as a basis for describing the IEC 61131-3. It describes *common elements* and *programming languages* (PLCopen, 2013). Common elements include *data typing*, *variables*, *configuration*, *resources*, *tasks*, *Program Organization Units* (POUs) and *Sequential Function Charts* (SFC). The programming languages describe the two textual programming languages *Instruction List* (IL) and *Structured Text* (ST) and the two graphical programming languages *Ladder Diagram* (LD) and *Function Block Diagram* (FBD). The following paragraphs contain a short description of these elements, derived from the definitions in (PLCopen, 2013).

Data Typing: The standard specifies common data types, such as Booleans (`BOOL`), Integers (`INT`) and floating point numbers (`REAL`), but also allows for users to define own data types. The latter are called *derived data types* and include *enumerations* (sets of named values) and *structures* (equivalent to the basic data structure *record*).

Variables: Variables (storage locations with an associated name) can be defined in configurations, resources, and POUs. Their scope is usually confined to the organization unit unless declared public (using the keyword `VAR_GLOBAL`). They can be assigned an initial value upon start-up of the program.

Configuration, resources, and tasks: A *configuration* is used to formulate the connection of an entire control software to a specific type of control system, including the arrangement of hardware (assignment of variables to I/O channels). It can contain several *resources* – one for each computing unit (e.g. when using a multicore processor). Each resource, in turn, can contain several *tasks*, which control the execution of a set of *Programs* (see POUs). The programs usually include calls and sub-calls of other POUs. Tasks can be configured to be executed periodically or upon certain events, such as variable value changes. A conventional PLC will most commonly contain one configuration, one resource, and a few tasks, configured to be executed periodically.

Program Organization Units (POUs): *Functions* (FUN or FC), *Function Blocks* (FB) and *Programs* (PRG) are POUs. POUs comprise of an interface and an implementation body, which contains executable code defined in any of the programming languages of the IEC 61131-3. FBs and PRGs can also be structured using SFC (see next paragraph). The greatest differences between the POUs are that, firstly, both FBs and PRGs can retain values over scan cycles, i.e.

they can "remember" values, while FUNs cannot. Secondly, PRGs and FUNs only exist once in a program and can be accessed globally. In contrast to this, FBs can be instantiated multiple times and only be accessed according to the scope of the variable of their instance.

Sequential Function Chart (SFC): SFCs are derived from Petri nets and IEC 60848 Grafcet, which graphically describe a sequential behavior. While defined as a programming language in the IEC 61131-3, SFC is described by the PLCopen as a way to structure the internal organization of FBs and PRGs rather than a programming language. It consists of steps, which are linked to actions (programmed in the programming languages and SFC), and transitions, which are associated with a condition. Starting from an initial step, a step is activated if a preceding step connected to this step is active and the transition connecting both evaluates to true. SFCs allows diverging, converging and parallel sequences.

Programming languages: *Instruction List* (IL), *Ladder Diagram* (LD) and *Function Block Diagram* (FBD) are less powerful programming languages that were designed to resemble known programming structures such as LD for electrical wiring, IL for assembler code and FBD for Boolean logic diagrams. *Structured Text* (ST) is a more powerful language with roots in Pascal and contains most capabilities of a modern programming language, including branching (e.g. `IF-THEN-ELSE`) and loops (e.g. `FOR`, `WHILE`). It is, therefore, suitable for complex functions, albeit not as easily understandable for users less experienced with programming.

2.1.2 The Technical Process in Production Automation

According to (Lauber and Göhner, 1999, pp. 43–47), *technical processes* consist of different *procedures*. These procedures can be classified in *continuous procedures* and *event-discrete procedures*. *Continuous procedures* are procedures containing time-related, continuous process values, e.g. deformation procedures in hydraulic presses. Suitable mathematical models for descriptions can be linear differential equations. *Event-discrete procedures* are characterized by sequentially occurring, stepwise, distinguishable (discrete) process states and often individually distinguishable objects. The transition between the process states can be described as binary events, defining the occurrence of certain states. Examples of these kinds of procedures are the sequential states during movement of an elevator or discrete manufacturing steps during production using machining tools. Suitable models for describing these procedures can be finite state machine models, such as Petri nets or flow charts.

Technical processes typically do not consist of only one type of procedure (Lauber and Göhner, 1999, p. 46), yet in production automation, *event-discrete procedures dominate the technical process*. The technical process is therefore dominated by binary input and output values and controlled in a sequential manner.

2.2 Testing Basics and Definitions

Many definitions for quality assurance and testing for aPSs stem from the domain of software engineering. While there are certain differences between the domains of software engineering and aPS engineering, most of the definitions also apply for the investigated domain of aPS engineering.

Testing is an analytical method for quality assurance (Hoffmann, 2013, p. 20). In comparison to quality assurance methods dealing with process quality, it relates directly to product quality (Hoffmann, 2013, p. 20). In contrast to static analytical methods (see 2.3), testing is performed during system execution, while static methods focus on syntactic and semantic properties of the source code (Hoffmann, 2013, p. 23). Testing is *"an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component."* (IEEE, 2008, p. 11) It has *" [...] the intent of (i) revealing defects, and (ii) evaluating quality."* (Burnstein, 2003, p. 27) Yet, it is not suitable for proving the correctness of the system as *"program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."* (Dijkstra, 1972)

2.2.1 Errors, Faults and Failures

As testing has the intent of revealing *defects* (Burnstein, 2003, p. 27), it is important to define this expression as well as other expressions used in relation to this definition, such as *errors*, *faults*, *bugs,* and *failures*.

An *error* is *"1. a human action that produces an incorrect result, such as software containing a fault. 2. an incorrect step, process, or data definition. 3. an incorrect result. 4. the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition."* (ISO/IEC/IEEE, 2010, p. 128) In this thesis, the term *error* refers to a human action leading up to a *fault*.

A *fault* is *"a manifestation of an error in software."* or *"a defect in a hardware device or component."* (ISO/IEC/IEEE, 2010, p. 140) If referring to software, a *fault* is used equivalently to the term *bug*. If a fault is encountered, it may cause a *failure* (ISO/IEC/IEEE, 2010, p. 140).

The term *defect* is often not clearly defined as it is *"a generic term that can refer to either a fault (cause) or a failure (effect)"* (ISO/IEC/IEEE, 2010, p. 96) In this thesis and in some literature such as (Burnstein, 2003), the term *defect* is used equivalently to *fault* or *bug* (when referring to software defects).

A *failure* is the *"termination of the ability of a product to perform a required function or its inability to perform within previously specified limits."* (ISO/IEC/IEEE, 2010, p. 139) *"A failure may be produced when a fault is encountered."* (ISO/IEC/IEEE, 2010, p. 139) In the domain of aPS, the most common scenarios for failures are shutdowns of an aPS or the inability to detect products of unsatisfactory quality.

As an example in aPS control program engineering, a human *error* could be a misconception about the behavior of a piece of hardware, which leads to the programmer implementing a *faulty* control sequence. This fault could then cause the aPS to unexpectedly halt during operation, causing a *failure* of the machine to fulfill its purpose, the production of a product.

2.2.2 Levels of Testing

The system being tested, also termed *System Under Test* (SUT), *"may consist of hardware, system software, data communication features or application software or a combination of them"* (ISO/IEC/IEEE, 2010, p. 361). Thus, depending on the investigated SUT, only software components or a fully integrated system, including hardware, may be subject to testing and relevant for quality evaluation. Systems are commonly tested on different levels ranging from tests of individual components to the system as a whole.

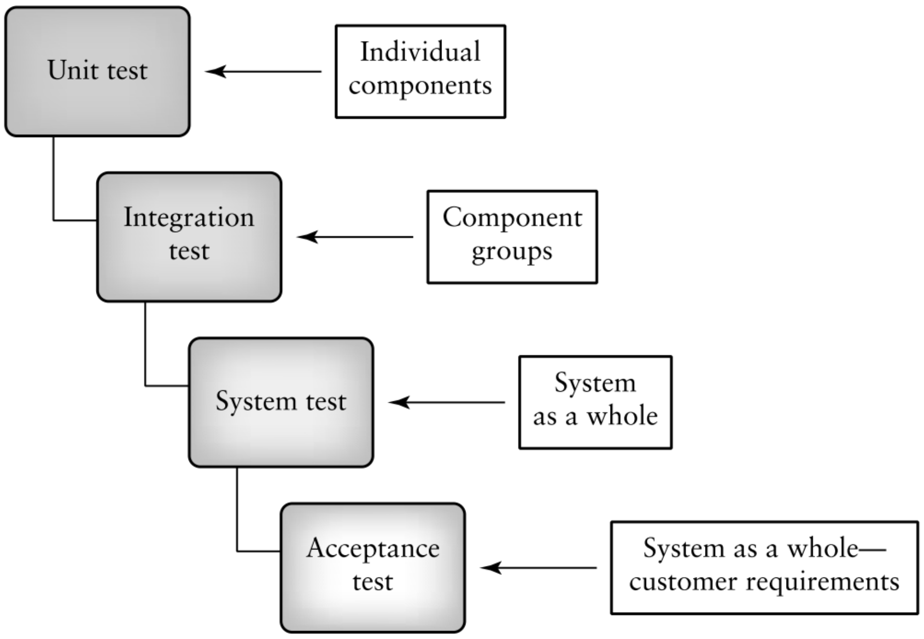


Figure 3: Levels of testing (Burnstein, 2003, p. 134)

Figure 3. shows the different levels of testing: the basic test levels are often divided into unit testing, integration testing, system testing and acceptance testing.

Unit testing, also called component testing, is *"testing of individual hardware or software components."* (IEEE, 2008, p. 8) It is often conducted on individual, independent software components and is aimed at ensuring that each individual software unit is functioning according to specification (Burnstein, 2003, p. 138). Translated to the domain of aPS, this type of test deals with testing individual program organization units (POUs) or hardware components independent from control software. The tests regarding the control software can be fully automated, as only software and no complex hardware behavior is involved.

Integration testing is *"testing in which software components, hardware components, or both are combined and tested to evaluate the interaction among them."* (IEEE, 2008, p. 9) In the domain of aPS, a scenario for this could be integrating a hardware component with a driver POU to evaluate whether they properly function together. The aim is not to test the individual components in depth, but rather focus on their interaction and integrated functionality. These tests are often not restricted to one engineering domain, but often involve hardware and software.

System testing is *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements."* (IEEE, 2008, p. 10) For aPS, this means that the fully integrated aPS, usually consisting of control software, execution hardware, bus systems, pneumatic piping, electrical wiring and hardware components are tested in integration with the technical process. The aim is to find deviations from the system to its specified requirements. In contrast to acceptance testing, it is mostly done without the customer and can, therefore, be performed more thoroughly.

Acceptance testing is *"(A) Testing conducted to establish whether a system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system. (B) Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component."* (IEEE, 2008, p. 8) Therefore, acceptance testing is similar to system testing with a different aim: establishing the notion that the system works according to specification towards the customer rather than internally within the engineering company.

2.2.3 Test Artefacts

During the course of the testing process, different artifacts are generated. The relevant artifacts for this thesis are *test plans*, *test cases*, *test suites*, *test drivers (test bed)* and *test reports*.

Before performing tests, a *test plan* can help structuring and controlling the process. A test plan identifies what items are to be tested, which tasks are to be performed, responsibilities, testing schedules, and (expected) required resources for the testing activity (ISO/IEC/IEEE, 2010, p. 370).

A *test case* specifies *"a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement."* (IEEE, 2008, p. 11) It can also serve as a documentation of which tasks were performed (IEEE, 2008, p. 11). It is, therefore, a very detailed plan for testing specific aspects of an SUT, including acceptance criteria. If the acceptance criteria are met, the test case is considered “passed”, if they are not met it is considered “failed”. If a test case could not or could not completely be executed, the result of the test case is “inconclusive”.

A set of test cases is usually called a *test suite*. A test suite can also include a sequence in which the individual test cases are to be executed.

Test cases can be manually performed, but are often automatically executed. In these cases, a *test driver*, which is *“a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results”* (ISO/IEC/IEEE, 2010, p. 369) is commonly used. In this thesis, it is synonymously used to the expression *test bed*.

Each test case should be documented. The results of the test case executions are commonly compiled in a *test report*, which is *“a document that describes the conduct and results of the testing carried out for a system or component.”* (ISO/IEC/IEEE, 2010, p. 371) Usually, it contains which test cases were performed, what their result was and other information, such as time, date and the name of the tester.

2.2.4 Test Case Design Strategies

When designing tests, two basic strategies can be distinguished: *black box testing* and *white box testing* (also known as glass box testing) (Burnstein, 2003, pp. 63–65).

Black box testing is *“pertaining to an approach that treats a system or component whose inputs, outputs, and general function are known but whose contents or implementation are unknown or irrelevant.”* (ISO/IEC/IEEE, 2010, p. 35) Test inputs and expected outputs are derived from the system specification or requirements. It is therefore mostly used synonymously to *functional testing*, which is *“1. testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions 2. testing conducted to evaluate the compliance of a system or component with specified functional requirements.”* (ISO/IEC/IEEE, 2010, p. 154)

White box testing, also known as *glass box testing*, is testing of *“a system or component whose internal contents or implementation are known.”* (ISO/IEC/IEEE, 2010, p. 157) As tests are derived directly from the implementation, it is often also referred to as *structural testing*, which is *“testing that takes into account the internal mechanism of a system or component.”* (ISO/IEC/IEEE, 2010, p. 349)

Both test case design strategies have their advantages and disadvantages. Black box testing is more suited to evaluating the compliance of an SUT to its specification, while possibly missing faults in the control flow that might not be defined in detail in the specification. White box testing is more suitable for unit testing to find these recently mentioned types of faults, yet might miss important parts of the specification that might not be implemented in the code. In addition, structural testing often requires a vast amount of test cases with growing complexity of the SUT, thus preventing its economic use in system testing.

2.2.5 Regression Testing

Regression testing is *"selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements."* (ISO/IEC/IEEE, 2010, p. 295) It is mostly performed by reusing existing test cases to investigate whether scenarios within these tests are still performed according to specification. It is therefore not aimed at testing the modification itself, which might require additional test cases, but the identification of newly introduced faults through previously successfully performed test cases. Regression testing can be performed on all testing levels and is mostly done with functional tests.

Re-executing all test cases can be very costly. For that reason, many approaches for reducing this effort were the focus of research for many years in computer science. The approaches mostly fall into the categories *regression test selection* (Rothermel and Harrold, 1997) and *regression test prioritization* (Rothermel *et al.*, 2001). *Test selection* techniques aim at reducing cost by selecting an appropriate subset of existing test cases (Rothermel *et al.*, 2001). If this selection is performed *safely*, the reduced set of test cases has the same ability to reveal faults while requiring less time to be executed. Not all test selection techniques are safe and those which are, assume *controlled regression testing*. This means that when testing will be performed on a modified system, nothing but the software changed in relation to the old version. All other factors, such as the execution hardware, the controlled hardware, and technical process would have to react exactly the same as before. This assumption is very hard to be held in system testing in the domain of aPS engineering, as every physical hardware movement or process step is generally at least stochastically distributed regarding timing. *Test prioritization* techniques use scheduling of a set of test cases to meet testing goals earlier, e.g. test coverage (Rothermel *et al.*, 2001). As prioritization techniques do not omit test cases, the possible drawback of unsafe selection cannot occur. For short test sets prioritization might not be cost effective, but for long test suites or test suites that fail, the remaining time can be spent more effectively (Rothermel *et al.*, 2001). For aPS, system test suites generally require manual interaction and are therefore costly. In addition, failing test cases require finding and fixing the problem and reiterating the regression testing process. It is therefore very beneficial to reveal possible regressions earlier.

2.2.6 Test Automation and Model-based Testing

A defined test case can be executed manually by stimulating the SUT as specified, and observing and comparing the SUT's outputs to the expected results. Particularly in unit testing, this very repetitive task is often automated. For this, a test driver (see 2.2.3) can be implemented, if the test cases directly refer to the SUT's interface and are defined in a machine-readable format.

"Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of a SUT and/or the behaviour of its environment. Test cases are generated from one of these models or their combination, and then executed on the SUT." (Utting, Pretschner and Legeard, 2012). Model-based testing is generally based on the idea that instead of defining each test case by itself, a model of the SUT can be used to derive test cases in a structured way. Figure 4 gives an overview of the process steps involved in MBT, which are:

(1) A test model is manually generated from informally specified requirements. An example of this model could be an abstract definition of SUT behavior according to the requirements.
(2) Test selection criteria are specified, which are often informal methods or guidelines, e.g. relating to coverage criteria of the test model, such as "every state in the test model has to be included in some test".
(3) The test case specification is the operational form (formalized) of the test selection criteria.
(4) Using the test case specification and the test model, test cases can be automatically generated.
(5) The test cases are subsequently executed. This can be done manually or automatically, depending on the situation and employment of a test script (see test driver, 2.2.3). The result of the test case execution is the verdict of the test cases.

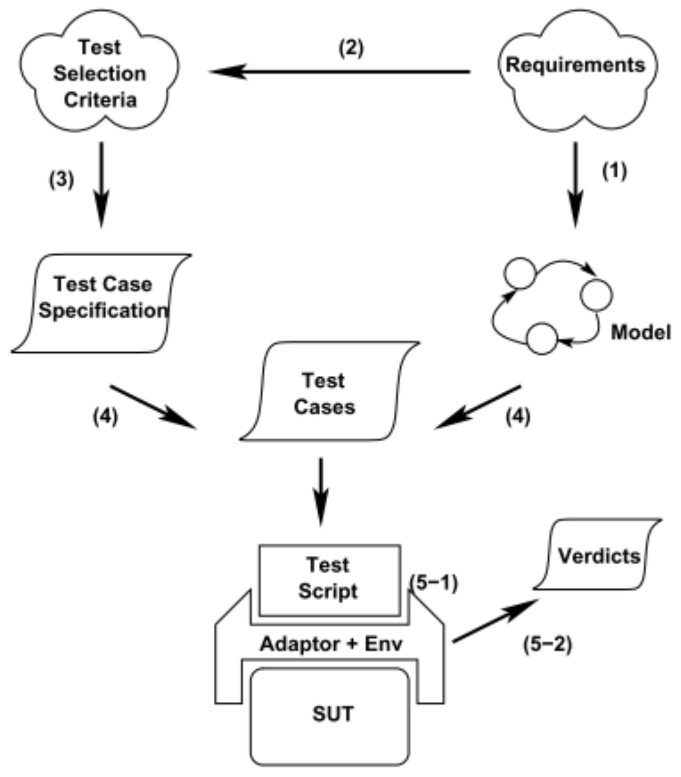


Figure 4: The process of model-based testing according to (Utting, Pretschner and Legeard, 2012)

While these process steps are present in many MBT approaches, not all approaches include every step. Some approaches directly define test cases as models, thus a test model only refers to exactly one test case (e.g. (Kormann, Tikhonov and Vogel-Heuser, 2012)).

2.2.7 Testing with Simulations

In many cases, especially in system testing, test cases do not solely relate to software interfaces. Instead, test actions referring to the integrated system are given. An example for this would be a test case specifying that a product that does not meet quality requirements should be correctly identified and sorted into a waste bin. The test case can therefore not be easily automated, but needs to be executed manually using a real machine or using a hardware or process simulation. Simulations can be used for testing in cases where a real machine is not (yet) available, the test might damage the aPS or other reasons that economically justify creating a simulation.

Integrating an SUT with a process simulation is often distinguished into *Software-in-the-Loop* (SIL) and *Hardware-in-the-Loop* (HIL) simulations. *SIL*, also called System Simulation, implies that both the control software as well as the simulated hardware and technical process are available in a virtualized form, both running on a PC for example. In contrast to this, *HIL* uses the execution hardware that is later used in productive use (e.g. a PLC) to execute the control software and can be connected to the simulation, e.g. via a bus system. While SIL is more flexible, as no additional execution hardware is needed, HIL can also be used to investigate effects caused by the real execution hardware (e.g. real-time constraints) and the simulation can be partially replaced with real sensors or actuators. (Barth and Fay, 2013)

For both techniques, it is necessary to create an adequate simulation. As the simulation itself does not include test cases or test automation, this setup has to be further extended by the test driver stimulating both the simulation and the SUT.

2.3 Static Analysis Basics and Definitions

In contrast to testing (see 2.2), static analysis aims at finding bugs in earlier development stages related to syntactic and semantic properties (Hoffmann, 2013, p. 23). Its focus is the identification of common error patterns (Louridas, 2006) or code optimization (Aho, Sethi and Ullman, 1986, p. 385). It is usually performed after successful compilation and before testing. Most tools for static analysis, such as CODESYS STATIC ANALYSIS (3S - Smart Software Solutions GmbH, 2016b) or ITRIS AUTOMATION PLC CHECKER (Itris Automation, 2017), as well as research approaches (Prähofer *et al.*, 2016) work in a similar way: Source code is transformed into an abstract representation and analyzed for common patterns relating to errors (Louridas, 2006). The first abstract representation generated from the code is often an *Abstract Syntax Tree* (AST) (Aho, Sethi and Ullman, 1986, p. 6; Parr, 2007, p. 162). As simple AST is shown in Figure 5.

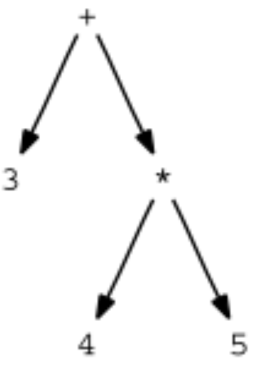


*Figure 5: An Abstract Syntax Tree (AST) for the expressions "3+4*5"* (Parr, 2007)

Based on the AST, specialized models for investigating data flow or control flow properties are common. In particular, the *Control Flow Graph* (CFG), which represents the flow of control through the code (see Figure 6, code (top) and its representation as a CFG (bottom)), can be used for various test coverage metrics (Burnstein, 2003).

```
/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer
array a. Input parameters are num_of_entries, an integer, and a, an array of integers with
num_of_entries elements. The output parameter is the integer sume */

1.  pos_sum(a, num_of_entries, sum)
2.      sum = 0
3.      inti = 1
4.      while (i <= num_of_entries)
5.          if a[i] >0
6.                  sum = sum + a[i]
            endif
7.          i = i + 1
        end while
8.  end pos_sum
```

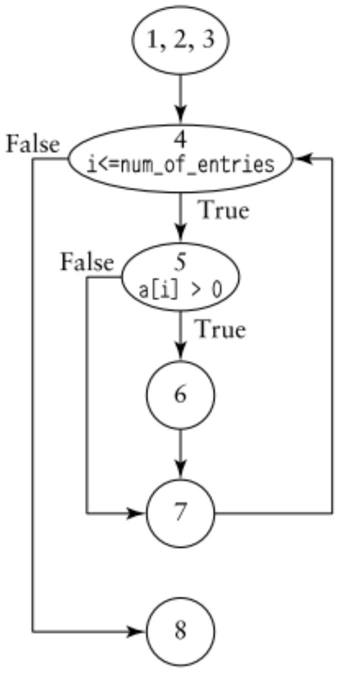


Figure 6: Code sample (top) and resulting control flow graph (bottom) (Burnstein, 2003)

The CFG is a directed graph, consisting of nodes and edges. Nodes represent sequential statements, also called *Basic Blocks* (BB). More specifically, *"[a] basic blocks is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end"* (Aho, Sethi and Ullman, 1986, p. 528). Edges represent control loops or branches in the control flow, as for example caused by loop statements (`FOR`, `WHILE`, `DO`) or conditional statements (`IF-THEN-ELSE`, `CASE`).

Static analysis methods are not aimed at verifying functional specifications and can therefore not be directly related to system testing. Yet, they prove to be very useful for instrumentation, coverage calculation, and change impact analysis.

3 Analysis of System Testing in Production Automation and Derivation of Requirements on the Approach

The presented approach was developed in close cooperation with industrial experts. In several workshops with up to seven experts from three different internationally renowned companies related to or active in the field of factory automation, the current state of practice and resulting requirements were derived. In addition to these workshops, the current situation in the aPS engineering domain was discussed with several experts from other reputable companies active in this field. While the presented situation might not apply to all companies, the described boundary conditions and problems apply to a major percentage of companies in this domain from the author's point of view. This was supported by further discussions with other companies about the subject.

Some of the requirements are derived from the situation of aPS engineering in production automation and need to be addressed to achieve industrial applicability of a new approach. Some others are relating to problems with the current industrial approach of system testing, thus need to be fulfilled in order to improve the current situation.

These requirements represent the foundation for the development of the approach presented in this thesis and the rating of related work. At the same time, the requirements were used for the evaluation, to design experiments and for the expert discussion to gain an understanding whether the proposed approach fulfills the initially imposed requirements presented in this section.

3.1 Requirements Regarding the Applicability of the Approach

To allow for a quick adoption of the approach in industry, it was important to compile requirements regarding the boundary conditions of the industrial partners. In general, the experts noted that the approach to be developed was to *be applicable to industrial aPS software*, *allow for testing including valid hardware and technical process behavior* and *not interfere with real-time capabilities or memory restrictions* of the execution hardware. All these requirements were to be fulfilled, while *no additional resources for formalized behavior models (simulations)* are available.

In the following sections, requirement definitions are presented for each of these boundary conditions.

3.1.1 Support of Industrial aPS Software

Most aPS in production automation are programmed in the IEC 61131-3 standard (IEC, 2003). While other standards were developed such as the IEC 61499 (IEC, 2012) aiming at improving distributed automation systems, the IEC 61131-3 is still the dominant programming method of PLCs in production automation. The third edition of the IEC 61131-3 (IEC, 2013), which includes object-orientation, is still hesitantly adopted and mostly used in software libraries.

The IEC 61131-3 defines several programming languages in which an aPS software can be programmed in (see section 2.1.1, p. 66). The companies participating in the workshops mainly develop their programs in *Structured Text* (ST) and *Sequential Function Chart* (SFC). To enable an evaluation of the applicability of the approach and a rating of related approaches in the field, the following requirement was derived:

Requirement R_{IEC} – Support of IEC 61131-3: At least one of the languages specified in the IEC 61131-3 needs to be supported by the approach.

Furthermore, typical industrial aPS software consists of multiple *Program Organization Units* (POUs) and many lines of code. To allow for a quantitative measure, the following requirement including definite numbers was defined:

Requirement R_{SW} – Support of industrial code complexity: APS software with multiple, interacting POUs and more than 5000 lines of code need to be supported.

3.1.2 Inclusion of Valid Hardware and Process Behavior

The approach needs to be applicable to real industrial testing use cases, as defined by the currently performed system test cases in the company. System tests, as described in this approach, are defined as black box tests (test derived from a specification rather than the code itself) of a fully integrated system comprised of software and controlled hardware in interaction with the technical process. The tests include manual manipulations of the hardware or technical process that cannot be performed by the software. As an example, manually opening and closing doors, or putting intermediate products in the machine, can be typical operations during system testing. To include the possibility of interacting with more than just the software of the system, the following requirement was defined:

Requirement R_{Int} – Support of interaction with the integrated system: Manipulation and observation of hardware and technical system must be supported.

Testing a system integrates all parts of the system, meaning software and hardware in combination with the controlled process. As each test is defined in a black-box manner, test cases only describe the stimulation and expected behavior of the fully integrated system. The interaction between software, hardware, and technical process are therefore not defined in detail. Thus, to be able to perform system tests that only describe input-output-behavior of the fully integrated system, the internals of the system need to be connected. For this it is also important that the hardware behavior is valid, i.e. behaving exactly like in reality. Substituting this part of the system with a simulation is an abstraction of the final system. Thus, finding faults (partially) related to hardware behavior that is lost during abstraction cannot be found.

Requirement R_{HWB} – Inclusion of valid hardware behavior: The software is required to interact with the hardware and technical process during testing.

To clarify the difference between both requirements (R_{Int} and R_{HWB}), Figure 7 shows the connection between them and the SUT. R_{Int} relates to enabling all required external interaction with the system, while R_{HWB} concerns the valid behavior and internal interaction of the software, hardware and technical process. Thus, substituting parts of the system, e.g. with simulations, the performed tests cannot be seen as systems tests relating to the final system, as behavior is modified due to this substitution.

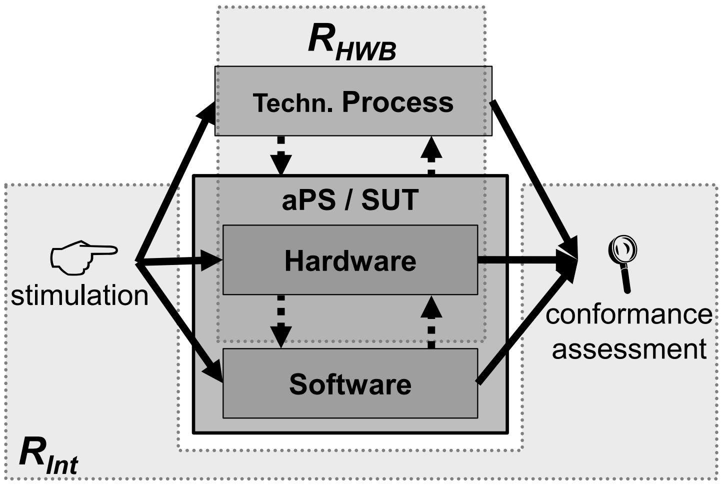


Figure 7: Relation of the requirements R_{Int} and R_{HWB} to the SUT

3.1.3 Independence from Formalized Behavior Models

Simulations are used in early phases of aPS development, mostly to gain a better understanding of achievable production cycle times (time to produce one product), which is often an integral part of the customer's specification. Yet, these simulations are always a simplification of real hardware behavior and often do no reach a level of detail or include unwanted situations and are therefore not usable for testing purposes. Developing a suitable simulation for testing purposes would need substantial resources. As the regarded aPSs are produced in very small lot sizes, this effort would not be profitable, as the cost for one simulation would have to be divided among only a very small number of machines for which this simulation could be applied. This problem especially applies to medium and smaller sized companies, where an approach which is independent of simulations is required, as these are often no option for system testing in production automation for economic reasons. At the same time, first versions of simulations often miss important effects in the machines, which are iteratively implemented upon identification in the real system. Yet, with small lot sizes, performing these iterations is limited, leaving the validity of simulations questionable. This further decreases the payoff of creating such simulations. Most companies, therefore, currently choose to skip simulations for testing and perform system tests directly on the machine instead, which results in the following requirement.

Requirement R_{Sim} – Independence from behavior simulations: No formalized models or simulations of hardware or technical process behavior is required for the approach.

3.1.4 Real-time and Memory Size Restrictions

The approach should not influence the real-time properties of the tested system in a way that would not permit needed real-time capabilities of the system to hold. The needed real-time capabilities are seen as unaffected if a possible increase in execution time of modified code does not lead to the PLC scan cycle time to be exceeded. As described in section 2.1.1, the PLC scan cycle includes reading all inputs, executing the PLC program and writing all outputs. The requirement was defined as follows.

Requirement R_{RT} – Insignificant influence on real-time properties: Maximum PLC scan cycle time is not to be exceeded due to the approach.

In addition, possibly increased size of compiled control code software should not lead to exceeded memory on the execution hardware (PLC).

Requirement R_{Mem} – Insignificant influence on memory size: Available memory size on the execution hardware is not to be exceeded due to the approach.

Both requirements depend on the application use case. The rating of the fulfillment of the approaches is thus related to a specific use case (e.g. application example). Yet, this information can be extrapolated to gain qualitative estimation of the fulfillment of this requirement.

3.2 Requirements Regarding the Improvement of the Current Situation in System Testing of aPS

Besides the boundary conditions as present in the aPS industry, several requirements were derived from properties of the testing process that were seen as unsatisfactory for the involved companies with the currently employed approach of manual testing.

The environment in which the system tests are currently performed is often very uncomfortable: the tests are often performed on-site, in the customer's premises, often in a loud and dirty environment. In addition, the test and modifications to the aPS are often performed under substantial time pressure. During commissioning, deadlines for the start of operation are often coupled with penalties upon exceeding. During maintenance or fixing of bugs, every minute of the aPS not being in operation is costing substantial amounts of money. For these reasons, the testing personnel, engineers or technicians, are under substantial stress, which can lead to several problems, such as *varying testing quality*, *lack of documentation* or *missing of critical test cases*. In addition, complex interdependencies within the software and the lack of documentation make it hard to decide which tests to use for regression testing upon implementing changes.

In the next sections, these problems were used to derive requirements on the approach for improving this current situation.

3.2.1 Improvement of Repeatability and Transparency of the System Testing Process

Due to the strong relation of the individual capability of each tester to the testing quality and the strong influence of stress during the testing situation, unwanted deviations in test quality are likely to occur. In addition, personnel expenses are high, especially when relying on experienced testers, and repeatability of the tests can suffer, in particular if the documentation of the performed tests is kept to a minimum. To tackle this problem, the following requirement was defined.

Requirement R_{Rep} – Improved repeatability: The approach is to enable and improve the repetition of the testing process at a later time and by different personnel.

During the current system testing setting, documentation of the performed actions are seen as cumbersome and are often performed only in a minimal manner. This fact forbids reproducing the behavior of the system during testing or assessing the current state of testing. At the same time, increasing the resources available for documentation is not possible. For this reason, the following requirement was defined.

Requirement R_{Doc} – Improved documentation: Detailed documentation of the performed test cases and their outcome has to be supported.

3.2.2 Support of the Assessment of Test Adequacy

Test adequacy is a matter of judgment by the tester. Whether everything was tested can roughly be compared to the test plan or generic test specifications (if available), yet a support for this estimation is not available in aPS engineering. While different coverage metrics (see section 4.1.4) allow for a quantified estimation of test adequacy, e.g. a percentage value of full coverage, this number was seen as questionable by the experts for usage in system tests. As resources for completely testing a system are not available (testing all behavior in every detail in an aPS is not feasible), fully covering a complete aPS was seen as an unachievable test adequacy criterion. Any specific number below "100% coverage" was seen to have little meaning, as metrics greatly vary in thoroughness (for further details, see section 4.1.4). Therefore, rather than assessing how complete the system behavior was tested, the requirement was set to finding untested behavior and assessing its need for specifying tests in agreement with the experts.

Requirement R_{TA} – Support for test adequacy assessment: The approach is to support the assessment of test adequacy through identification of untested behavior.

3.2.3 Increase in Efficiency During the Testing Process of Changes to a Previously Tested Control Software

Similar to the assessment of test adequacy, the efficiency during regression testing relies heavily on the ability of the tester. Based on experience and intuition, this person has to select, prioritize and repeat test cases in an efficient way. The set and sequence of test cases should ideally be able to find as many faults as quickly as possible. Yet, so far, this is done completely without tool support. Thus, the approach provides automated support to improve the efficiency during the testing process required after the implementation of software changes to a previously tested system. In particular, a support for choosing or prioritizing test cases for regression testing has to be given.

Requirement R_{Reg} – Support for regression testing: The approach is to increase the efficiency during regression testing.

3.3 Scope Limitation

Based on the field of investigation and the requirements compiled in the industrial workshops, the problems that this thesis is aiming to solve are within the field of system testing of aPS in production automation regarding event-discrete technical processes and specially engineered aPS (small lot size). Certain problems within this domain are out of the scope of this work. Firstly, this thesis deals with the problems in system testing of fully integrated aPS rather than single mechatronic or software components. Therefore, the scope is limited to system testing rather than testing processes that can be fully automated, such as software unit testing. Secondly, only discrete-event technical processes are regarded, excluding continuous processes as dominant in the chemical industry. Thirdly, the approach aims at improving the testing process of aPS that are produced in small quantities (special purpose machines with lot sizes less than ten, down to one). In contrast to other fields, such as mass production, all engineering documents, test specifications or simulations have to be developed and produced for these small series. Thus, reusability of these artifacts is very low, resulting in tighter resource restrictions.

4 State of the Art

Given the requirements compiled in the previous chapter, current works of research were reviewed and rated for their applicability on the domain and problem. In addition, adjacent domains such as computer science and automotive engineering were investigated. In the following sections, these works will be presented. In each section, the reviewed approaches are compiled into a table, rating each approach regarding the requirements derived in Chapter 1. The resulting research gap is defined in this process and formulated in Section 4.2. The approach developed in this thesis aims at filling this research gap.

4.1 Analysis of Existing Approaches in Production Automation and Adjacent Domains

A multitude of approaches was developed for improving the quality assurance of software in general and control software of aPS is particular. (Hoffmann, 2013) differentiates three categories for analytic quality assurance of software: *static analysis*, *formal software verification,* and *software testing*. Each of these fields will be regarded in more detail. In addition, a closer investigation of the fields of *test coverage assessment* and *regression testing* will be given, as these are central concepts of the approach developed in this thesis. The rating scheme is given in Table 1, stating for each requirement what properties were required for it to be fulfilled (+), partially fulfilled (○) or not fulfilled (-).

Table 1: Rating scheme for the evaluation of existing, related approaches

R_{IEC} – Support of IEC 61131-3
+: At least one IEC 61131-3 language supported or independent from implementation language ○: No support of IEC 61131-3 languages, but application in the field of embedded systems -: No support of IEC 61131-3 languages and application only in the field of computer science or higher programming languages
R_{SW} – Support of industrial code
+: APS software with multiple, interacting POUs and more than 5000 lines of code ○: Lab-sized experiments -: Approach only focused/applied on individual functions or POUs or loosely connected, delimited modules
R_{Int} – Support of interaction with the integrated system
+: Manual or simulated interaction with hardware and technical process behavior supported -: No support for interaction with hardware or technical process behavior
R_{HWB} – Inclusion of valid hardware behavior
+: Inclusion of hardware behavior using real hardware ○: Inclusion of hardware behavior using simulations or other hardware behavior models -: No inclusion of hardware behavior
R_{Sim} – Independence from behavior simulations
+: Simulation or hardware behavior model not required for execution of the approach ○: Simulation or behavior model required for the approach; can be (partially) generated from existing engineering documents -: Simulation or behavior model required for the approach

R_{RT} – Insignificant influence on real-time properties
+: Approach does not alter or embed program (e.g. HIL) or significance of influence on real-time properties investigated and found to be negligible ○: Approach embeds unmodified program, but influence on real-time properties not investigated -: Approach modifies or embeds original program and influence was not investigated in relation to aPS or embedded systems
R_{Mem} – Insignificant influence on memory size
+: Approach does not alter program (e.g. HIL) or significance of influence on memory overhead investigated and found to be negligible ○: Approach embeds unmodified program, but influence on memory usage not investigated -: Approach modifies original program and influence was not investigated related to aPS or embedded systems
R_{Rep} – Improved Repeatability
+: Repeatability is a focus of the approach and was investigated or the approach employs fully automatic testing/execution ○: Repeatability was not investigated; improvements very likely -: Repeatability was not investigated; improvements very unlikely
R_{Doc} – Improved Documentation
+: Automatic recording of detailed execution information during testing ○: Improvement of documentation not investigated; automatic recording likely possible -: Improvement of documentation not investigated; automatic recording unlikely possible
R_{TA} – Support for Test Adequacy Assessment
+: Approach enables assessment of test adequacy ○: No method for test adequacy assessment implemented; tests are generated using coverage metrics, thus, coverage can be inferred, if all test are executed -: No method for test adequacy assessment implemented
R_{Reg} – Support for Regression Testing
+: Selection or prioritization of existing test cases supported ○: No methods directly aimed at regression testing; re-testing has very low demand on resources -: No methods directly aimed at regression testing; re-testing has high demand on resources

4.1.1 Static Code Analysis

Static code analysis (see Chapter 2.3) aims at an early detection of common error patterns. An advantage of these approaches is the relatively small cost of execution: Most static analysis tools finish analysis in a matter of seconds. Most available commercial tools, such as CODESYS STATIC ANALYSIS (3S - Smart Software Solutions GmbH, 2016b) or Itris PLC CHECKER (Itris Automation, 2017) are aimed at checking (configurable) pre-defined rules regarding naming conventions or potential problems in variable access. In research, this has been extended by the ability for user-defined rules and more advanced analysis of the control flow (Prähofer *et al.*, 2012, 2016; Angerer *et al.*, 2013; Stattelmann *et al.*, 2014; Nair *et al.*, 2015; Biallas, 2016). One approach offers an additional explorative analysis of programs, using graphical representations of the dependencies within the code (Feldmann, Ulewicz, *et al.*, 2016; Ulewicz, Feldmann, *et al.*, 2016). Another focus of recent research was the adherence to coding guidelines in FBD programs for nuclear power plants (Jung, Yoo and Lee, 2017).

As summarized in Table 2, the approaches all possess several quite similar positive and negative aspects. On the positive side, they are optimized for aPS software (R_{IEC}), do not struggle with industrially sized programs (R_{SW}) and do not alter the program, so no influence on real-time behavior or memory size exists (R_{RT} and R_{Mem}). In addition, no specially designed formal hardware models (R_{Sim}) are required and the performed analyses are deterministically repeatable (R_{Rep}). Automatic documentation is technically possible, yet often not explicitly mentioned in the presentation of the approaches (R_{Doc}). Support for the identifying regressions in the system are not directly part of the approaches, yet, static analysis is relatively low in expense and thus could be fully executed for every evolution of a system (R_{Reg}).

Despite all of these positive aspects, several problems regarding the approaches and their applicability to the initially defined requirements exist. First, static analysis techniques tend to over-approximate the possible negative impact of error patterns. This means that the results of these methods often contain many *false positives*, i.e. results that are presented as critical but do not cause any problems in real use. This is partially due to not taking the behavior of the software's environment (hardware and technical process) into account (R_{Int} and R_{HWB}). Second, the analysis does not relate to a particular system, but to common error patterns. Thus, static code analysis does not allow for an investigation of a system's functional conformance to its specification. This also does not allow for any adequacy assessment of the SUT (R_{TA}).

Table 2: Evaluation of related approaches in the field of static code analysis

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
(3S - Smart Software Solutions GmbH, 2016b; Itris Automation, 2017)	+	+	-	-	+	+	+	+	+	-	○
(Prähofer *et al.*, 2012, 2016; Angerer *et al.*, 2013)	+	+	-	-	+	+	+	+	○	-	○
(Nair *et al.*, 2015) (Stattelmann *et al.*, 2014)	+	+	-	-	+	+	+	+	○	-	○
(Biallas, 2016)	+	+	-	-	+	+	+	+	○	-	○
(Feldmann, Ulewicz, *et al.*, 2016) (Ulewicz, Feldmann, *et al.*, 2016)	+	+	-	-	+	+	+	+	○	-	○
(Jung, Yoo and Lee, 2017)	+	+	-	-	+	+	+	+	○	-	○

4.1.2 Formal Verification

Formal verification can be used to mathematically prove compliance of a model to a specification. This is commonly done using model checking, where the compliance of the model to so-called proof obligations (rules the model has to fulfill) is verified. Several tools for model checking have been developed that offer a backend for formal analysis (Uppsala University (UPP) and Aalborg University (AAL), 2010; *nuXmv*, 2014, *NuSMV*, 2015) and even a frontend graphical editor for developing behavior models (Akesson *et al.*, 2006; Uppsala University (UPP) and Aalborg University (AAL), 2010). With these tools, the models have to be manually generated or imported from other sources. In some works, these model checkers were shown to be applicable for verification of small problems in aPS, given that all models and proof obligations are specified manually (Buzhinsky and Vyatkin, 2017). Other research groups developed their own specialized tools for the domain of aPS, such as ARCADE.PLC (Biallas, Brauer and Kowalewski, 2012; Biallas, 2016), which generates a model directly from aPS control software. Other works use similar approaches, such as for programs written in LD (Kottler *et al.*, 2017) or SFC (Bauer *et al.*, 2004). A toolchain for verifying industrially-sized programs is introduced in (Gourcuff, de Smet and Faure, 2008; Fernández Adiego *et al.*, 2015). These approaches require only the specification of proof obligations rather than a creation of a system behavior model. In (Ljungkrantz *et al.*, 2010), an approach is proposed, in which reusable components including not only the implementation but also its specification for verification purposes. The approach is mainly aimed at individual POUs and simple integrations thereof. An approach for a more abstract verification of the interoperability of technical process and an aPS system (hardware and control software) is presented in (Hackenberg *et al.*, 2014). The approach requires modeling of material flow behavior, steps in the technical process, system behavior and software behavior and thus requires high efforts, even if the system verification is performed on a rather abstract level.

Regarding the applicability of these approaches on aPS, they are independent of the aPS implementation or allow for the generation of models from at least one of the IEC programming languages (R_{IEC}). None of the approaches influences real-time behavior (R_{RT} and R_{Mem}) as the code itself is not executed and investigated, but a model thereof. Once the modeling is complete, the verification process can be performed fully automatically, not hindering repeatability or documentation (R_{Rep} and R_{Doc}), even if this has not been investigated in detail by most works. This ability to automate the formal verification process is also beneficial for regression testing, as the resources for repetition are rather low (not manual interaction needed), yet most formal approaches tend to require increasingly long times (hours - days) to solve complex aPS problems.

While a system's compliance with the specification can be exhaustively proven, the approaches using formal verification require extensive resources for the specification of formalized requirements and system models, extensive knowledge of formal methods and often fail to deal with the complexity (R_{SW}) of fully integrated systems ("State Space Explosion"). Some approaches try to mitigate the complexity problem by abstraction (Gourcuff, de Smet and Faure, 2008; Fernández Adiego *et al.*, 2015) or by focusing on performed changes (Ulewicz, Ulbrich, *et al.*, 2016). Yet, the appropriate relation to and interaction with valid hardware behavior remains (R_{HWB} and R_{Int}).

Table 3: Evaluation of related approaches in the field of formal verification

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
(Buzhinsky and Vyatkin, 2017)	+	○	-	○	-	+	+	+	○	-	○
(Biallas, Brauer and Kowalewski, 2012; Biallas, 2016)	+	○	-	○	-	+	+	+	○	-	○
(Bauer *et al.*, 2004)	+	○	-	○	-	+	+	+	○	-	○
(Kottler *et al.*, 2017)	+	-	-	-	+	+	+	+	○	-	○
(Gourcuff, de Smet and Faure, 2008)	+	+	-	-	+	+	+	+	○	-	○
(Fernández Adiego *et al.*, 2015)	+	+	-	-	+	+	+	+	○	-	○
(Ljungkrantz *et al.*, 2010)	+	-	-	-	+	+	+	+	○	-	○
(Hackenberg *et al.*, 2014)	+	○	+	○	-	+	+	+	○	-	○
(Ulewicz, Ulbrich, *et al.*, 2016)	+	○	-	○	-	+	+	+	○	-	+

4.1.3 Testing

Using model-based test generation and test automation techniques (see (Rösch *et al.*, 2015) for a literature review), the effort of specifying and performing test cases can be reduced. Formalized functional specifications can be used to describe intended system behavior (test models) or fault handling functionality of the system. Models can also be used to describe hardware or process behavior and can, therefore, be used for simulation.

Tools for describing test models and generating test cases from them are readily available, e.g. Sepp.med MBTsuite (Sepp.med GmbH, 2017) or ALL4TEC MaTeLo (ALL4TEC, 2017). While the generation of individual test cases is reduced, the definition of such test models requires substantial effort. In addition, behavior models are required for system testing (R_{Sim}, R_{HWB} and R_{Int}). Nonetheless, several approaches in the aPS domain pursue this direction. Some use environmental models of the system (Kumar *et al.*, 2013), special requirement ontologies (Sinha *et al.*, 2016) or synchronized depth first search in automata (Pinkal and Niggemann, 2017) to generate executable test cases. Others convert specialized Grafcet specifications to automata for conformance testing, generating input and expected output sequences (Provost, Roussel and Faure, 2011, 2014). All of the aforementioned approaches require complex models to be specified and validated. As the models often also yield a very large set of test cases, some works are aiming at reducing these test sets, e.g. by including plant model features (Ma and Provost, 2017a, 2017b). Some works use modified UML sequence diagrams to specify individual test cases (Hametner *et al.*, 2011; Vogel-Heuser *et al.*, 2013). More recently, an approach using timing sequence diagrams to generate test cases covering all possible signal mutations for different classes of mutations was proposed (Rösch and Vogel-Heuser, 2017).

Most of the mentioned MBT approaches are fully automated, thus requiring behavior models of hardware and technical process outside the control software, to be applicable to system testing. For this, *virtual commissioning* techniques have proven to be valuable for testing of aPS produced in greater lot sizes. Here, detailed simulations are specified, enabling an automatic execution of a multitude of test scenarios (Liu *et al.*, 2014; Süß *et al.*, 2016; Thonnessen *et al.*, 2017). Software tools for creating and performing simulations for aPS are readily available, e.g. TRYSIM (Cephalos GmbH, 2017), WINMOD (Mewes & Partner GmbH, 2017) and PLCLOGIX (Logic Design Inc., 2017). Unfortunately, the creation of simulations requires extensive effort to enable the representation of valid system behavior regarding the hardware and technical process. This problem can be mitigated by designing simulations of different abstraction levels related to the tested problem (Puntel-Schmidt *et al.*, 2014; Puntel-Schmidt and Fay, 2015) or by using existing engineering artifacts for an automatic generation of simulation models (Barth and Fay, 2013). In many cases, the required documents and resources for the creation of the simulations are not available in the industry, especially for individually engineered machines and plants (lot size 1).

Table 4: Evaluation of related approaches in the field of (model-based) testing

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
Test generation tools (ALL4TEC, 2017; Sepp.med GmbH, 2017)	+	+	+	○	-	○	○	○	○	○	-
(Kumar *et al.*, 2013)	+	○	○	+	-	+	+	+	○	○	-
(Sinha *et al.*, 2016)	+	-	○	+	-	+	+	+	○	○	-
(Pinkal and Niggemann, 2017)	+	○	-	○	-	+	+	+	○	○	-
(Provost, Roussel and Faure, 2011, 2014)	+	○	-	○	-	+	+	+	○	○	-
(Ma and Provost, 2017a, 2017b)	+	+	○	+	-	+	+	+	○	○	-
(Hametner *et al.*, 2011)	+	○	+	○	-	○	○	+	○	-	-
(Vogel-Heuser *et al.*, 2013)	+	○	+	○	-	○	○	+	○	-	-
(Rösch and Vogel-Heuser, 2017)	+	+	+	+	+	+	+	○	○	○	-
Simulation Tools (Cephalos GmbH, 2017; Logic Design Inc., 2017; Mewes & Partner GmbH, 2017)	+	+	+	○	-	+	+	○	○	-	-
(Süß *et al.*, 2016)	+	+	+	○	-	+	+	○	○	-	○
(Liu *et al.*, 2014)	+	-	+	○	-	+	+	○	○	-	○
(Thonnessen *et al.*, 2017)	+	+	+	○	○	+	+	○	○	-	○
(Puntel-Schmidt and Fay, 2015) (Puntel-Schmidt *et al.*, 2014)	+	+	+	○	○	+	+	○	○	-	○
(Barth and Fay, 2013)	+	+	+	○	○	+	+	○	○	-	○

4.1.4 Test Coverage Assessment

Coverage metrics in the field of computer science have been an active research topic for many years. They can be used for test case generation (Anand *et al.*, 2013), change impact analysis (Bohner and Arnold, 1996; De Lucia, Fasano and Oliveto, 2008), regression test selection and prioritization (Engström, Runeson and Skoglund, 2010; Yoo and Harman, 2012) or for assessing test suite adequacy (Yang and Chao, 1995; Zhu, Hall and May, 1997; Gligoric *et* *al.*, 2013). While some approaches have already been incorporated into the production automation domain, coverage metrics have rarely been used for assessing test suite adequacy in this field. In the following, a closer look into work related to the presented approach will be taken.

4.1.4.1 Requirements based Coverage Assessment

Requirements based coverage metrics are based on the relation of requirements and test cases in which test cases check whether the system under test fulfills a set of requirements. In reverse, if an approach uses functional requirements or specifications for test generation, it is assumed that the generated test case is adequate for these requirements.

A basic realization of this approach is available in multiple requirements management tools, such as RATIONAL DOORS (IBM, 2016) or POLARION (Siemens, 2016): informally specified requirements can be linked to informally specified test cases. If a requirement does not have a related test case, it is assumed that a test case is missing. In case one or more test cases are linked to a requirement, the requirement is seen as fulfilled if all test cases were completed successfully. This implies that the creator of the test cases specified all relevant test scenarios, which is relying heavily on the ability of the individual. If test cases for a requirement are missing, this would not become apparent if all other test cases were executed successfully as no quantitative measure beyond the connection of tests and requirements is given.

Using formalized requirements, the expected behavior of the system can be specified in more detail and a subsequent relation between content of the requirement and the test cases can be performed, rather than solely evaluating the connection of requirement and test case. This was implemented in different research works in specific applications for embedded systems (Siegl and Caliebe, 2011), computer science (Whalen *et al.*, 2006) and testing of safety field busses (Krause, 2012). Here, test cases can be generated from and related to a system model and thus allow for an assessment of test coverage. Another work presents an approach, where tests are symbolically executed and compared to a specification using a model checker (Ammann and Black, 1999). The approach claims independence from implementation language, yet the type of test cases is not taking cyclical execution into account, preventing application on aPS. Several works in the domain of aPS testing also use specifications models for test generation using coverage metrics (Kumar *et al.*, 2013; Sinha *et al.*, 2016) and therefore allow for an assessment of test adequacy related to the test models prior to test execution. Table 5 gives an overview of these approaches, including a rating regarding the initially derived requirements in aPS system testing.

An approach which bases test coverage on formalized or traceable requirements exhibits multiple positive aspects: 1) it is a direct measure of how well a test suite addresses a set of requirements, 2) it is implementation independent and 3) it does not require execution or instrumentation of the system under test (Whalen *et al.*, 2006). At the same time, this type of metric has several negative aspects:

1. Detailed requirements specification as well as a model of the relation between these requirements and test cases need to be available. This requires additional effort.

2. A quantitative evaluation of this metric requires formalized requirements specifications or relies solely on the assumption that a given set of test cases can fully strengthen the notion that all requirements are fulfilled, even if each requirement only relates to a single test case.

3. Missing or incomplete requirements will go unnoticed in this metric. If the set of requirements is not maintained correctly or inadequately defined from the beginning, the metric cannot yield satisfying results.

4. Unrequired parts of the code cannot be identified as test cases are related to requirements only. If unneeded code was implemented, this metric is unable to identify these unnecessary parts of the code resulting in additional maintenance effort in later stages of system maintenance.

Table 5: Evaluation of related approaches in the field of requirements-based coverage assessment

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
Commercial requirements engineering tools (IBM, 2016; Siemens, 2016)	+	+	+	+	+	+	+	-	-	-	-
(Siegl and Caliebe, 2011)	○	○	-	-	+	+	+	○	○	+	-
(Whalen *et al.*, 2006)	○	○	-	-	+	+	+	○	○	+	-
(Krause, 2012)	○	-	-	-	+	+	+	○	○	○	-
(Ammann and Black, 1999)	-	○	-	-	+	+	+	○	○	+	-
(Kumar *et al.*, 2013)	+	○	○	+	-	+	+	+	○	○	-
(Sinha *et al.*, 2016)	+	-	○	+	-	+	+	+	○	○	-

4.1.4.2 Code Structure Based Coverage Assessment

Structural code coverage metrics have been a common method for assessing software test adequacy in safety critical systems: For safety-critical avionics systems, the DO-178b standard proposes the use of structural metrics for assessing the adequacy of a suite of tests for a test subject (RTCA, 1992). Depending on the criticality, the standard proposes more or less detailed metrics, such as statement coverage (*"every statement in the program has been executed at least once"* (RTCA, 1992)) or condition/decision coverage (*"every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at last once"* (RTCA, 1992)) is proposed. While the presented approach does not aim to fulfil coverage criteria as in software testing for safety critical systems for economic reasons, certain properties about the different metrics still apply to the field of aPS. The different metrics differ substantially regarding the number of test cases to fulfill each criterion, but also in their ability to detect faults. In most cases, full statement coverage needs fewer test cases but can fail to detect faults in complex decisions within the control flow. The practical ability to detect faults in desktop software was evaluated for unit testing (Zhu, Hall and May, 1997) and complete test suites (Gligoric *et al.*, 2013; Gopinath, Jensen and Groce, 2014). From these evaluations, even the simple statement coverage metric turns out to be a very valuable metric, especially for finding out if a test suite is inadequate (missing test cases). The DO-178b (RTCA, 1992) also proposes statement coverage as the minimal requirement for safety critical systems.

There are many tools available from computer science for structurally based coverage analysis. An overview of available tools is given by (Yang, Li and Weiss, 2009), yet not all tools have remained in development. Still, tools such as CLOVER (Atlassian, 2016), BULLSEYECOVERAGE (Bullseye, 2016) and PURIFYPLUS (Unicom, 2016) are readily available and offer coverage analysis using multiple coverage criteria for higher object-oriented programming languages (e.g. Java, C++, C#). Even if their application on the programming languages and execution hardware of PLCs could be achieved (R_{IEC}), these tools were not developed in respect to the industrial scenarios required in the aPS industry: all tests are executed fully automatically and do not require human operators (R_{Int}). In addition, the influence of the tracing algorithms used by the tools is unsure regarding a port into the PLC field (R_{RT} and R_{Mem}).

In comparison to computer science and critical embedded systems, production automation engineering has seen few approaches of structural code coverage metrics for test adequacy assessment. A tool for test coverage measurement for Function Block Diagrams (FBD) is presented by (Jee *et al.*, 2010) for use in safety critical programs of nuclear reactors. The test case coverage is externally checked by analyzing the data flow paths of the FBD and comparing them to the test inputs. This approach seems to be hardly applicable to aPS in production automation: Here, the technical process is dominated by event-discrete process steps (see 2.1.2) which differ substantially from the complex logical data flows of nuclear power plants.

A reason for the hesitant use of coverage criteria in production automation might be the strict real-time requirements and the overhead created by measuring coverage (R_{RT} and R_{Mem}). Only a few works present efficient tracing algorithms for embedded systems (Wu *et al.*, 2007) and automation software (Prähofer *et al.*, 2011). As tracing algorithms of the former approach are designed for embedded systems, the scenarios and test environment prohibit an easy adaption to automated production systems (no interaction to hardware without behavior models). The latter approach for automation software uses a very sophisticated tracing approach aimed at debugging automated production systems, not taking coverage assessment into account. In particular, this approach does not include structured system tests or a connection of tests to the recorded data and focuses mainly on reproducing variable values at certain points in time for debugging purposes.

Some works in the production automation field take a different approach stemming from computer science: testing input sequence generation from the code itself to achieve full coverage related to a certain criterion. Some are generating test relating to the data flow of FBD (Jee, Yoo and Cha, 2005; Jee *et al.*, 2009; Doganay, Bohlin and Sellin, 2013; Enoiu *et al.*, 2013;

Enoiu, Sundmark and Pettersson, 2013; Maruchi, Shin and Sakai, 2014). Other works use model-checkers and the control flow to do the same (Simon *et al.*, 2015; Bohlender *et al.*, 2016). Some approaches extend this idea for software product lines, offering efficient test input sequence generation techniques for individual configurations (Lochau *et al.*, 2014). These approaches possess two main problems for system testing in production automation: 1) the generation technique generates test input sequences directly from the code but does not include an expected behavior ("test oracle"). The test cases themselves are very focused on software rather than system operation scenarios (R_{Int}, R_{HWB}), thus realistic system testing scenarios are hardly achievable. 2) As the generated test suite is usually very large, it is difficult to execute all test cases, especially concerning the complete system. Test coverage is completely unknown if certain test cases are omitted (R_{TA}).

An overview of the mentioned approaches, including their rating, are shown in Table 6.

Table 6: Evaluation of related approaches in the field of structure-based coverage assessment

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
Commercial test coverage assessment tools (Atlassian, 2016; Bullseye, 2016; Unicom, 2016)	-	+	-	-	+	-	-	○	○	+	-
(Jee *et al.*, 2010)	+	○	-	-	+	○	○	+	○	+	-
(Wu *et al.*, 2007)	○	○	-	-	+	+	+	+	○	+	-
(Prähofer *et al.*, 2011)	+	+	+	+	+	+	+	-	+	-	-
(Jee, Yoo and Cha, 2005; Jee *et al.*, 2009)	+	-	-	-	+	○	○	+	○	○	-
(Maruchi, Shin and Sakai, 2014)	+	-	-	-	+	○	○	+	○	○	-
(Doganay, Bohlin and Sellin, 2013) (Enoiu *et al.*, 2013; Enoiu, Sundmark and Pettersson, 2013)	+	+	-	-	+	○	+	+	○	○	-
(Simon *et al.*, 2015) (Bohlender *et al.*, 2016)	+	-	-	-	+	○	○	+	○	○	-
(Lochau *et al.*, 2014)	+	-	-	-	+	○	○	+	○	○	-

4.1.5 Regression Testing

Besides the general approach to make testing of aPS more efficient and increase testing quality as presented in the previous paragraphs, several approaches focus on the selection and prioritization of existing test cases. In the domain of computer science in particular, this has been an active field of research for many years (Yoo and Harman, 2012). Two main classes of approaches can be identified in this field: static and dynamic techniques. *Static techniques* focus on the connection of engineering artifacts to the test cases (traceability) to gain information about what test case should be selected and prioritized due to a change. *Dynamic techniques* leverage data acquired during the execution of test cases to allow for a relation of the test cases to the tested code. After changes are performed on the system, this allows for an assessment of test cases having a higher probability to yield different results, e.g. by failing, exposing newly introduced unwanted behavior. While static analysis generally is better at finding all relevant test cases (soundness), dynamic approaches in computer science often choose less unnecessary test cases (Ernst, 2003) and are thus more precise.

4.1.5.1 Static Regression Test Selection and Prioritization

Regarding static traceability methods, established tools for requirements engineering can be used to infer connections between requirements and test cases (IBM, 2016; Siemens, 2016) similarly to test coverage. In the case of changed requirements, related test cases can quickly be identified. Yet in practice, changes are often directly performed on the system, often without a change of the requirements documents. In these cases, a detailed relation of test cases to the performed change is impossible to achieve, hindering a prioritization of available test cases (R_{Reg}). Several approaches try to improve this by including additional information, such as system models of the SUT (Caliebe, Herpel and German, 2012) or the connection between test cases, requirements and product cost of a software product line (Baller *et al.*, 2014). An approach from computer science performed prioritization upon four factors: requirements volatility, customer priority, implementation complexity and fault proneness (Srikanth, Williams and Osborne, 2005). This prioritization method requires manual effort, yet structured prioritization of test cases with a reduced level of subjectivity can be achieved. Other approaches use historic test execution data, e.g. information about test runs, such as the fault detection rate of individual test cases (Kim and Porter, 2002) or a fault symptom database with connected test cases (Abele and Weyrich, 2017). A combination of the system's software structure and historic process data in decentralized production systems (Zeller and Weyrich, 2015) or historic test execution data in combination with a clustering of test cases (Alagöz, Herpel and German, 2017) for test selection are more propositions to improve the regression testing process. To cope with the amount of data for prioritization using historic data, agent-based prioritization techniques have been proposed (Malz and Göhner, 2011; Malz, Jazdi and Göhner, 2012). For variant-rich industrial IT systems with well-delimited modules, an approach for prioritizing sub-system test cases that are directly related to modules via a system model was presented (Abele and Weyrich, 2016; Abele *et al.*, 2017). Yet, this approach would only be applicable to systems that are far more modularized and loosely connected as those regarded in this work.

While these approaches seem beneficial in specific situations, all of these approaches require additional artifacts and their connection to the system, which are often not available in the industry. In addition, the approaches are often focused on automatically executable test cases that do not take hardware behavior into account (R_{Int} and R_{HWB}). One approach for test selection is independent of additional models, solely basing the choice of test cases on changes to the software (Ulewicz, Schütz and Vogel-Heuser, 2014), yet the approach is limited to unit and integration testing, and is not applicable to system testing of aPS. Table 7 summarizes these findings.

Table 7: Evaluation of related approaches in the field of static regression test selection and prioritization

	$\mathbf{R_{IEC}}$	$\mathbf{R_{SW}}$	$\mathbf{R_{Int}}$	$\mathbf{R_{HWB}}$	$\mathbf{R_{Sim}}$	$\mathbf{R_{RT}}$	$\mathbf{R_{Mem}}$	$\mathbf{R_{Rep}}$	$\mathbf{R_{Doc}}$	$\mathbf{R_{TA}}$	$\mathbf{R_{Reg}}$
Commercial requirements engineering tools (IBM, 2016; Siemens, 2016)	+	+	+	+	+	+	+	-	-	-	-
(Caliebe, Herpel and German, 2012)	○	+	-	-	+	+	+	○	○	-	+
(Baller *et al.*, 2014)	+	+	-	-	+	○	○	○	○	-	+
(Kim and Porter, 2002)	○	+	-	-	+	+	+	+	○	-	+
(Srikanth, Williams and Osborne, 2005)	-	○	-	-	+	-	-	+	○	-	+
(Zeller and Weyrich, 2015)	○	○	○	+	-	○	○	○	○	-	+
(Alagöz, Herpel and German, 2017)	○	+	○	+	-	+	+	○	○	-	+
(Malz and Göhner, 2011; Malz, Jazdi and Göhner, 2012)	+	○	-	-	+	○	○	○	○	-	+
(Abele and Weyrich, 2017)	○	+	+	+	+	○	○	○	+	-	+
(Abele and Weyrich, 2016; Abele *et al.*, 2017)	○	-	+	+	+	○	○	○	○	-	+
(Ulewicz, Schütz and Vogel-Heuser, 2014)	+	○	-	-	+	+	+	-	-	-	+

4.1.5.2 Dynamic Regression Test Selection and Prioritization

When using dynamic regression test selection and prioritization methods, execution traces are recorded during the execution of test cases to allow for a relation between executed code and its related test case. By identifying changes in a system, this information can be used to prioritize available test cases (Jones and Harrold, 2001; Rothermel *et al.*, 2001; Orso, Apiwattanapong and Harrold, 2003). Originally, these works stem from computer science, thus important requirements for aPS are not regarded. In particular, the test cases used in the approaches are performed completely automatically, with no possibility for manual interaction or inclusion of valid hardware behavior (R_{Int} and R_{HWB}). In addition, their runtime overhead is not investigated (R_{RT}). Only one approach for tracing seems applicable for aPS, yet the approach focuses on reproducing variable values for manual debugging and does not consider test automation or regression testing (Prähofer *et al.*, 2011). In work preliminary to the approach presented in this thesis, a regression test prioritization approach was presented, trying to overcome these obstacles. Yet, the approach was only applied on a lab-sized study (R_{SW}) and did not include a test adequacy investigation (R_{TA}). An overview of the approaches and their rating is given in Table 8.

Table 8: Evaluation of related approaches in the field of dynamic regression test selection and prioritization

	R_{IEC}	R_{SW}	R_{Int}	R_{HWB}	R_{Sim}	R_{RT}	R_{Mem}	R_{Rep}	R_{Doc}	R_{TA}	R_{Reg}
(Orso, Apiwattanapong and Harrold, 2003)	-	+	-	-	+	-	-	+	○	-	+
(Rothermel *et al.*, 2001)	-	+	-	-	+	-	-	+	○	-	+
(Jones and Harrold, 2001)	-	+	-	-	+	-	-	+	○	-	+
(Prähofer *et al.*, 2011)	+	+	+	+	+	+	+	-	+	-	-
(Ulewicz and Vogel-Heuser, 2016b)	+	○	+	+	+	+	+	○	+	-	+

4.2 Discussion of the Research Gap

As the discussion of the related work in the previous sections shows, none of the analyzed approaches succeeds in fulfilling the imposed requirements identified in the workshops with the industry partners (Chapter 1). This is mostly due to relying on additional detailed formalized requirements or to not taking valid hardware behavior into account sufficiently. In addition, the disregard of important real-time requirements and industrially relevant test scenarios including manual interaction with the system prevent all approaches from computer science to be directly applicable. Thus, the identified research gap is defined as follows.

Research Gap: There is no approach for test adequacy and regression test prioritization for system testing of aPS in production automation, which is (directly) applicable in the aPS industry. The industrial applicability is mainly defined by the support of IEC 61131-3 languages and control programs of industrial complexity, no significant influence on real-time properties and the inclusion and possibility to manipulate hardware behavior. Furthermore, the approach is not to depend on formalized behavior simulations.

The approach developed in this thesis aims at filling this research gap.

5 A Concept for Efficient System Testing of Automated Production Systems

This section describes the developed concepts that aim at improving the system testing process in production automation. To illustrate their interrelation, a brief overview of the complete approach will be given, followed by a detailed description of the individual concepts.

As described in section 3.1.2, system testing in aPS engineering is currently performed mostly fully manually. This causes several problems: the repeatability of test cases is low, the documentation of the test cases is minimal and the quality of executed system tests is unknown or roughly estimated by personal experience of the test engineer or technician. Furthermore, selection, prioritization, and execution of these tests after software changes suffer from a similar problem: change impacts are largely unknown due to the complexity of the system and a lack of tool support. This results in unneeded re-execution of unaffected tests or – even worse – omission of affected tests and the subsequent disregarding of unwanted behavior caused by changes.

The goals of the presented approach are to increase code quality (a) by reducing deviations in system test execution and increasing transparency through detailed automatic documentation (R_{Rep} and R_{Doc}), (b) by identifying untested parts of the implementation (R_{TA}) and (c) by increasing testing efficiency by prioritizing tests that should be re-executed due to changes to the program (R_{Reg}). To ensure industrial applicability, requirements regarding the application domain are to be fulfilled. This means that industrial aPS software properties need to be factored in (R_{IEC} and R_{SW}) while ensuring real-time capabilities of the system (R_{RT}). As the type of considered tests is on the system test level, i.e. functional tests regarding the integrated behavior of a system's software and hardware (R_{HWB}), human interaction during execution is required (R_{Int}). Additionally, the approach may not rely on behavior models or simulations (R_{Sim}), as these are often not available.

The presented approach aims at reaching these goals by using a novel *guided semi-automatic system testing* approach, combined with a *tracing and coverage assessment* concept. These concepts are combined with a *change impact analysis and test selection and prioritization* concept to further improve the testing process of aPS with focus on regression testing of software changes.

5.1 Concept Overview

Regarding the requirement demanding more testing quality and efficiency, several problems are present when using the current approach of manual system testing. The testing quality is impaired due to a lack of *transparency*, i.e. very few if any detailed knowledge about what tests were already performed is documented, *repeatability*, i.e. the possibility of exactly repeating a test, and *measurability* to assess the testing adequacy in general. In addition, manual testing lacks in testing efficiency regarding regression testing as *selection and prioritization* of tests after implementing modifications in the system are mostly performed by a matter of feeling by the test technician or engineer, resulting in varying quality of testing efficiency.

Many of these problems cannot be overcome solely relying on improvements in manual testing, as most of the decisions and processes during manual testing heavily rely on individual experience and ability. In addition, manually performing tests requires significant manual effort for documentation, as no computer-aided support is given. Therefore, a guided semi-automatic system testing concept (see Chapter 5.2) was developed, allowing for the required type of system tests for real industrial scenarios. At the same time, it aims at structuring the testing process (increased repeatability) and enabling measurements during execution (increased transparency and measurability). It also represents the foundation for the other presented concepts offering the possibility to relate the testing artifacts (e.g. test suite information, test cases, test traces) amongst each other and to the control software itself and previous revisions of it.

Under the assumption that all parts of the control software were programmed to achieve required functionality, untested parts of the code imply untested functionality and thus missing test cases. As absent needed test cases represent a lack of test quality, a *test coverage assessment concept* (see Chapter 5.3) was developed that enables the identification of untested code. For this identification, a measurement on which parts of the code are executed during testing is performed (statement coverage). To acquire significant and complete information about executed statements, this measurement was to be developed to be real-time capable, e.g. not missing any program executions due to a recording algorithm lagging behind. As neither industrial control programs nor available development environments include the possibility to measure the needed information, the test coverage measurement concept includes the instrumentation of the code, implementing tracepoints that enable recording whether certain statements have been executed. Each tracepoint consist of the instrumented part of the code (a function call) and the information about its location and relation to the trace record. To keep this information connected for later use in the coverage analysis, a tracepoint database is created, storing this information.

To be in line with the requirements of applicability in the production automation domain, the concepts must not create any significant deviation from the behavior of the system (R_{RT}). Thus, only minimal modification of the code is permitted, resulting in the possibility to use the instrumented code as the final control program. For this reason, a detailed *control flow analysis* of the complete control program (see Chapter 5.3) is performed, optimizing the instrumentation of the code by identifying suitable locations of tracepoints within the code while still enabling the recording of all needed information. To allow for an efficient analysis, the executable code is converted into a *dependency model*, including the program control flow, which is based on directed graphs. The dependency model is additionally used as a base for visualizing the calculated test coverage.

Figure 8 represents a graphical overview of the connections of the described concepts: An original program is to be tested with a test suite (top left). A subsequent test coverage is to be performed to identify potentially untested code. The test suite is comprised of multiple tests, which were manually derived from the system requirements and specified in the previously described semi-automatic system test format. From this test suite, an executable test suite (executable IEC 61131-3 code) is automatically generated. To allow for a calculation of the coverage, the program is instrumented by the inclusion of tracepoints. The location of these tracepoints is identified through a dependency model, which is directly generated from the code of the original program. The instrumented program is then merged with the executable test suite and executed, during which test traces are recorded using tracepoints. The subsequent coverage calculation is performed using the traces, the dependency model and tracepoint information stored in a tracepoint database, which was created during code instrumentation. After the calculation of the coverage, the result can be visualized using a graphical representation of the dependency model, including information about coverage, particularly emphasizing unexecuted parts of the model. Using this information, the identification of untested code and a subsequent assessment of further needed test efforts are facilitated. While this assessment is difficult to automate completely, a manual assessment by the test engineer or technician is supported.

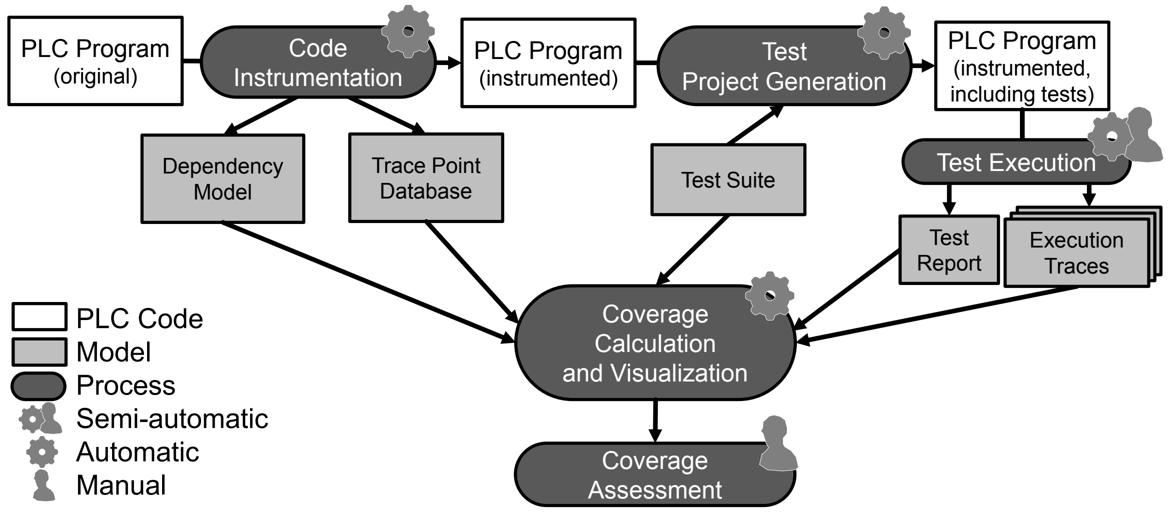


Figure 8: Coverage Assessment Overview

When regarding program modifications, regression testing plays an important role by possibly identifying newly introduced faults through re-execution of available tests. While the goal of identifying faults is important, re-executing test cases is also expensive resource and time wise, especially regarding system tests that involve personnel. To minimize these efforts, ideally only needed re-testing efforts would be performed (test selection) and arranged in a sequence optimized for efficiency (test prioritization). As a safe selection of test cases, i.e. avoidance of omission of test cases that could find faults, is hard to achieve for system testing of aPS in production automation (see Chapter 2.2.5), the presented approach focuses on the *prioritization of test cases* (see Chapter 5.4).

For this, an approach was developed that gives the personnel involved in the system regression testing process automated support in prioritizing test cases based on the implemented change of the system to enable quick identification of newly introduced unwanted behavior. As shown in Figure 9, so-called execution traces are recorded using the semi-automatic system testing approach (see Chapter 5.2) to allow for a relation between each system test and the related parts of the control software of the aPS. After changes to the system, this information is combined with a change identification and change impact analysis to allow for a prioritization of test cases with a high probability to identify newly introduced unwanted side effects. For this, test cases related to changed and possibly influenced parts of the code are prioritized before the rest of the test cases. A refined prioritization is performed either by prioritizing test cases covering as many modifications as fast as possible or by covering modifications as intensely as possible.

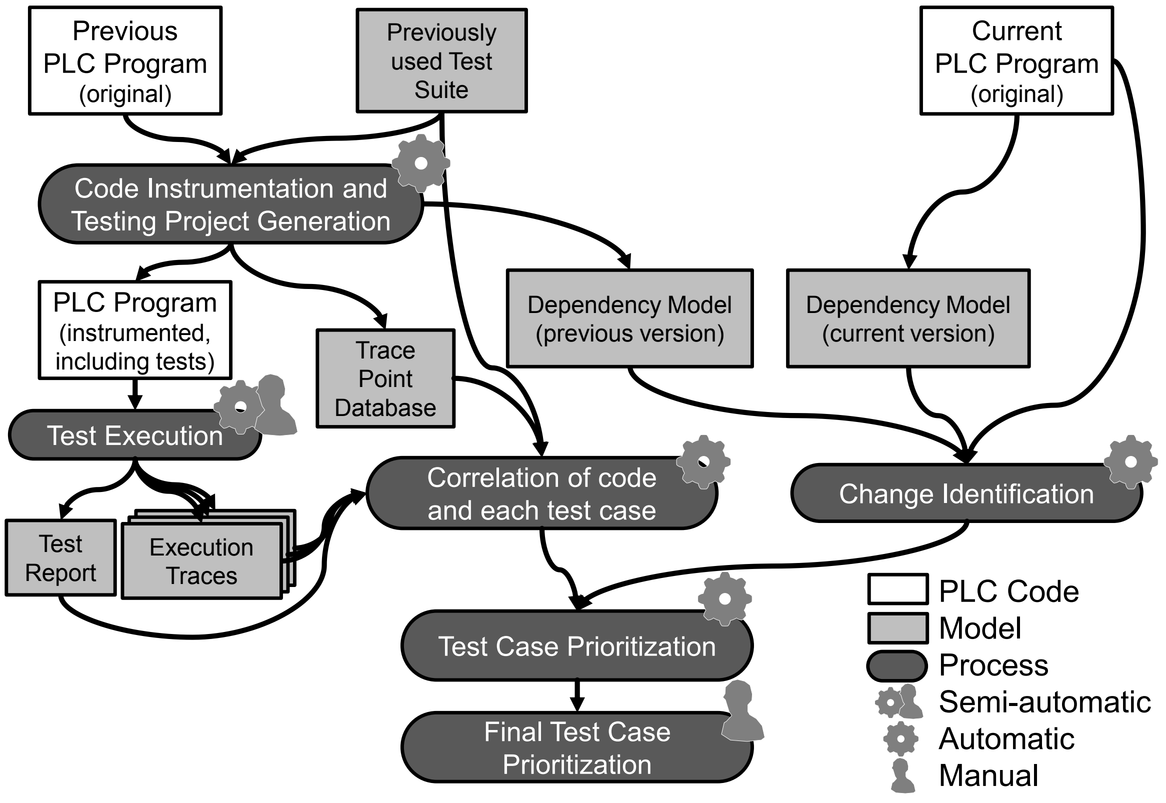

Previous PLC Program (original)
Previously used Test Suite
Current PLC Program (original)
Code Instrumentation and Testing Project Generation
PLC Program (instrumented, including tests)
Trace Point Database
Dependency Model (previous version)
Dependency Model (current version)
Test Execution
Correlation of code and each test case
Change Identification
Test Report
Execution Traces
Test Case Prioritization
Final Test Case Prioritization
PLC Code
Model
Process
Semi-automatic
Automatic
Manual

Figure 9: Overview of the concepts developed for prioritization of regression tests

In the following chapters, the main concepts will be described in detail.

5.2 Guided Semi-Automatic System Testing

In this section, a concept for guided, semi-automatic system testing is presented that is oriented towards the requirements of industrial use, as described in Chapter 1. It was first presented in (Ulewicz and Vogel-Heuser, 2016a). As discussed in the previous sections, many system testing processes in factory automation need the possibility of manual interaction and realistic hardware behavior (R_{Int} and R_{HWB}) while simulations or other formal behavior models are not available (R_{Sim}). As a result, in the presented concept, a human operator is included in a semi-automatic testing process as an efficient and applicable way to overcome the obstacle of stimulating and assessing hardware behavior in aPS in production automation. The system testing approach facilitates analysis of test case properties, the relation of test cases to other software and testing artifacts and usage of automated code generation algorithms. Therefore, it represents the foundation of the subsequently described approaches on coverage assessment (Chapter 5.3) and test prioritization (Chapter 5.4).

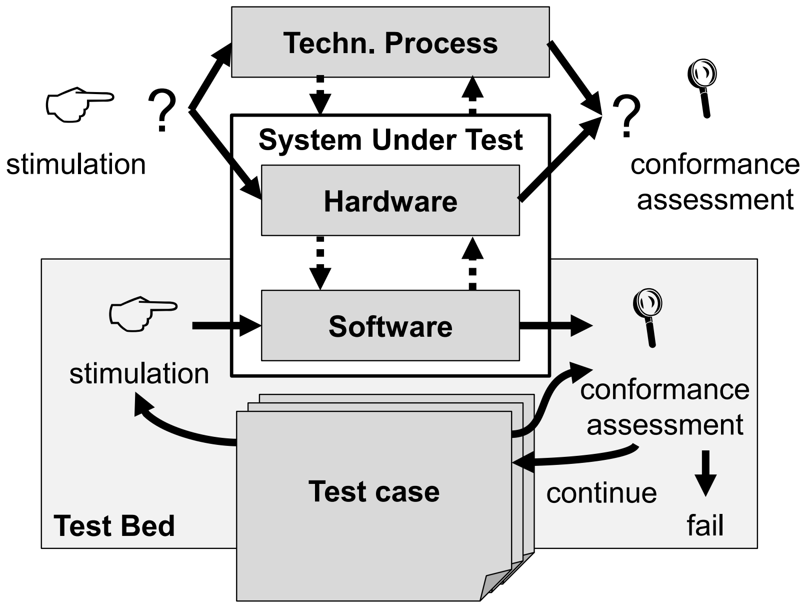


Figure 10: Inclusion of stimulation and behavior assessment into a system testing approach is currently not regarded in most approaches in research

The main gap the presented approach (Figure 10) is focusing on is the fact that none of the reviewed testing approaches considers the efficient testing of systems comprised of software and hardware in which not all testing processes can be automated. In these approaches, a test bed including multiple tests is used to stimulate a system under test via its inputs; its conformance analyzed using its outputs. While this is feasible when concentrating on software-only systems, systems including hardware can often not be fully stimulated or observed by the software. Some approaches try to avoid this problem, using simulations of the hardware behavior, yet this poses multiple problems for the domain of production automation, as simulations simplify reality, might not be fault free and require high effort in creation. In most cases, only the real hardware is adequate for the assessment of the system's behavior. As subsequently defined in requirement R_{Sim}, the presented approach is aiming at including stimulation and behavior assessment of the hardware parts of a mechatronic system in an efficient way to identify faults in the software.

In the following chapters, concepts for including human operators into the system testing process (5.2.1) and a test system integrating this concept into a semi-automatic process will be presented. The test system comprises the test cases (5.2.2), the test suite and execution history (5.2.3), and the generation of the PLC test project (5.2.4).

5.2.1 Including Human Operators into Testing Processes

In automated production systems, only required sensors or actuators for regular operation are available, mainly for cost reasons. As testing often involves only partial functionality of the system outside of regular operation, specified system states in the test cases cannot be reached automatically, as needed actuators are missing. In current practice, these states are mostly induced through manual interaction. An example for this would be a test whether safety doors are operational by manually opening and closing the doors for which no actuators are installed. Besides the missing actuators for reaching systems states, adequate sensors within the machine are missing or are inadequate for assessing system or product states. As an example, achieved product quality has to be visually assessed by test engineers or technicians. Installing and bringing adequate sensors and actuators into operation for these situations is not economically reasonable. Human interaction is flexible enough for most of these tasks and is, therefore, a prime candidate for inclusion within the concept. On the downside, humans exhibit a comparably low degree of precision during their actions and these actions are less resource efficient per repetition than highly specialized automated systems.

Automated production systems are developed to automatically perform actions, which are partially available during testing, e.g. in manual operating mode. Even though not all systems states can be reached or assessed automatically, many can. For this reason, this concept aims at combining human flexibility with automated precision and efficiency, by including a human testing engineer or technician into a testing process that is automated as much as economically reasonable.

Human interactions with the hardware during testing will be classified into *manipulative actions*, such as putting an intermediate product into the production plant or opening safety doors, and *diagnostic actions*, such as verifying visually whether an action was performed in a satisfying way. For integration of these actions with an automated system and testing functions, an interaction between the testing engineer and the automated testing processes needs to be achieved. The interaction between human operators and the automated production systems is usually achieved by using a human-machine interface (HMI). As peripheral devices, such as touch displays and other input devices are often already available in the production systems, these devices are considered for inclusion into the concept.

On the one hand, HMI devices are used to display information about the production process and machine status to a human operator during operation. In the same manner, tasks can be displayed to a testing engineer during the testing process. This can be realized in the form of textual or graphical information. This enables an inclusion of human actions by displaying information about the task to be performed, whether it is a manipulative or diagnostic action. On the other hand, the devices enable the human operator to influence the production process by changing parameters or switching operating modes. Again, this can be used for inclusion of human actions within the testing process, by using this feature for acknowledgment of manipulative actions or input of results of diagnostic actions. Input hardware such as touch displays or keyboards can be used to provide information to the automation system.

As portrayed in Figure 11, HMI devices can be used to include a human test engineer or technician into the test process. For this, the test case shall include information about manipulative as well as diagnostic actions to be performed by the tester at the appropriate times. Therefore, the HMI is bridging the gap between the fully automated test and the manual actions by the test engineer or technician. To further detail the concept, the test operations are regarded in more detail, resulting in a simple, yet powerful HMI concept.

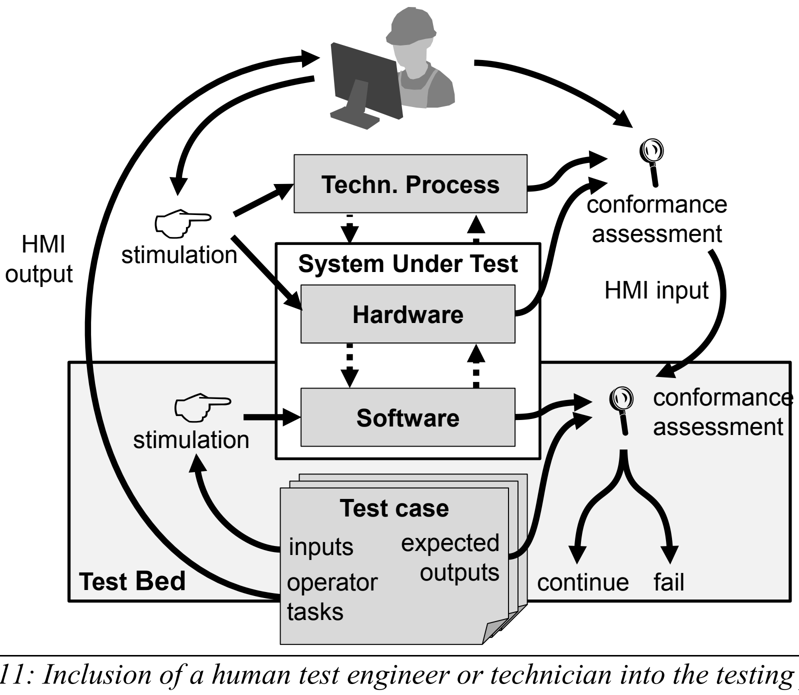


Figure 11: Inclusion of a human test engineer or technician into the testing process

As with regular test operations, in the most simple case, manipulative and diagnostic actions can either be successful (e.g. intermediate product inserted into the system) or not (e.g. product quality not satisfactory). For this reason, at least these actions need to be enabled by the HMI.

To conclude the involvement of the operator with the testing process, this results in the need for an HMI that can *display information* and *receive inputs for a successful and unsuccessful termination of the action*. This can be achieved through a simple visualization as shown in Figure 12.

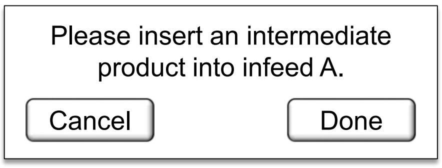

Please insert an intermediate product into infeed A.

Figure 12: HMI example for including a manipulative action into a test case

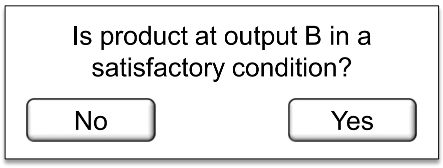

Is product at output B in a satisfactory condition?

Figure 13: HMI example for including a diagnostic action into a test case

Here, the task display is used to display information about the task in a textual manner using a text box. Two buttons are used to convey to the automated system whether this task was performed in a satisfactory manner ("Done") or not ("Cancel").

As mentioned before, besides the actions performed by the operator, certain processes can be fully or partially automated. Even though the processes are automated, there is a probability for these actions to fail. Therefore, the test engineer is involved in the testing process with a monitoring task during these processes. It is, therefore, reasonable to use the HMI to display information about the automatically performed actions and enable to abort the process through inputs. This can be achieved using the same HMI concept.

As exemplified in Figure 13, the task display is used to display information about the performed automated action to the operator and one button is used to stop the process in case it is not performed in a satisfactory way ("No"). The button formerly used to assess whether the action was satisfactory can be used to manually end an automatic process step in case there is no sensor available ("Yes").

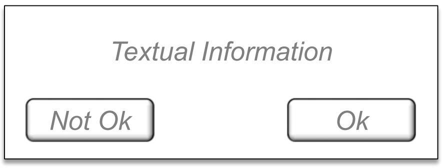


Figure 14: HMI template for manipulative and diagnostic actions using a customizable interface

In both use cases, customizing the text within the task display as well as the buttons is reasonable, resulting in the generic template visualization shown in Figure 14. This information has to be part of the test case for increased flexibility.

Using these presented concepts, a multitude of manual interactions can be included in the testing process, yet the automation of the testing process needs to be regarded in more detail to achieve a semi-automatic testing.

5.2.2 Test Case Metamodel and System Test Execution Process

The definition of the test case is based on joint concepts developed in the research project MoBaTeSt[1]. A refined variant of this test case format is used in the currently available version of the CODESYS Test Manager ((3S - Smart Software Solutions GmbH, 2016c), Version 4.1.0.x). Yet, this format does not include all properties of the test case format described in this section (especially *succeed conditions*). As an official formal specification was not published by 3S – Smart Software Solutions GmbH, the metamodel presented in Figure 15 is used for explanation. It represents a parallel refinement of the test case format as done by the author of this thesis. It has to be noted that the format in the CODESYS Test Manager includes most shown elements, yet with slightly different names. For the approach presented in this thesis, the most significant addition to the elements and the process in which the system test is executed are *succeed conditions groups* and more detailed *step conditions*. In contrast to assessments checking whether the SUT does not comply with an expected behavior, these conditions enable checking whether the SUT has reached a target state. This is particularly useful for system testing, where test steps are often designed as "remain in the current test step until X happens or the timeout is reached".

A system test relates to an SUT, defines a test interface (ManipulatedVariable and CheckedVariable) and includes a sequential set of test steps to test this SUT. In the case of the system tests defined in this thesis, the SUT is the fully integrated aPS. In the model, this is denoted by stating the POU designated as the entry point for the logic part of the program (e.g. "PLC_PRG" or "Main") as the SUT. Each test step in the set of test steps can define stimulating actions (StepAction), conditions relating to failing (FailCondition) or passing (SucceedConditionGroup) of a test step, and a step duration (StepDuration).

To get a better understanding of the purpose and usage of the elements the process of test case execution is schematically shown in Figure 16. Each test case is started at the first test step. The execution of a test step is started with checking for a timeout of the test case as defined in SystemTest.Timeout (see Figure 15, top left). If the timeout has not yet been reached, (optional) stimulating actions (StepAction) are performed, e.g. by setting variable values. Subsequently, the SUT is executed (optional, as defined in Step.ExecuteSUT, see Figure 15), after which each FailCondition is checked. StepConditions are usually checked by comparing variable values to specified expected values (CheckValue), the occurrence of rising or falling flanks (CheckFlank) or other custom definitions directly specified in ST (CheckExpression). If none

[1] „Model-based testing of PLC software variants for automated special purpose machinery" (German: "Modellbasiertes Testen von SPS-Steuerungssoftwarevarianten für den Sondermaschinenbau"). It was supported by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology within the research program "Informations- und Kommunikationstechnik in Bayern" under Grant IUK413.

of the FailConditions is violated, the SucceedConditionGroups are checked. If all conditions within a SucceedConditionGroup evaluate to true, this results in the group being evaluated as succeeded. If one or more SucceedConditionGroups are succeeded, the next step is executed.

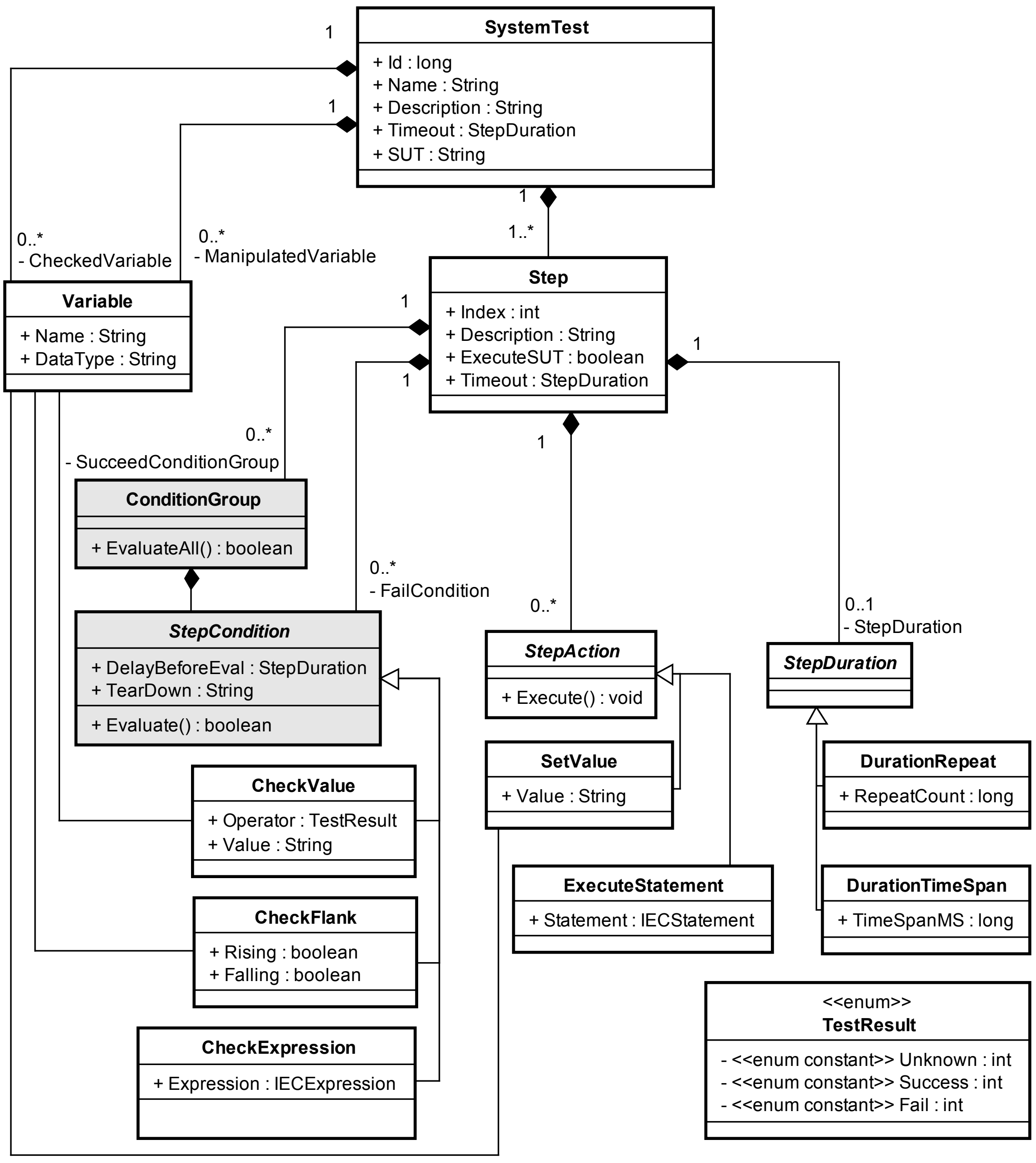


Figure 15: Metamodel for system tests based on joint concepts developed in project MOBATEST

If the system test timeout is exceeded or one or more FailConditions fail to be fulfilled, the system test is marked as *failed*. As abruptly stopping an aPS during operation can lead to problems such as an undefined state of the system, an additional tear down sequence can be defined. In case all test steps were completed successfully, the test result is set to *succeeded*. Unless the test is marked as *failed* or *succeeded*, the test case result remains *unknown* (inconclusive).

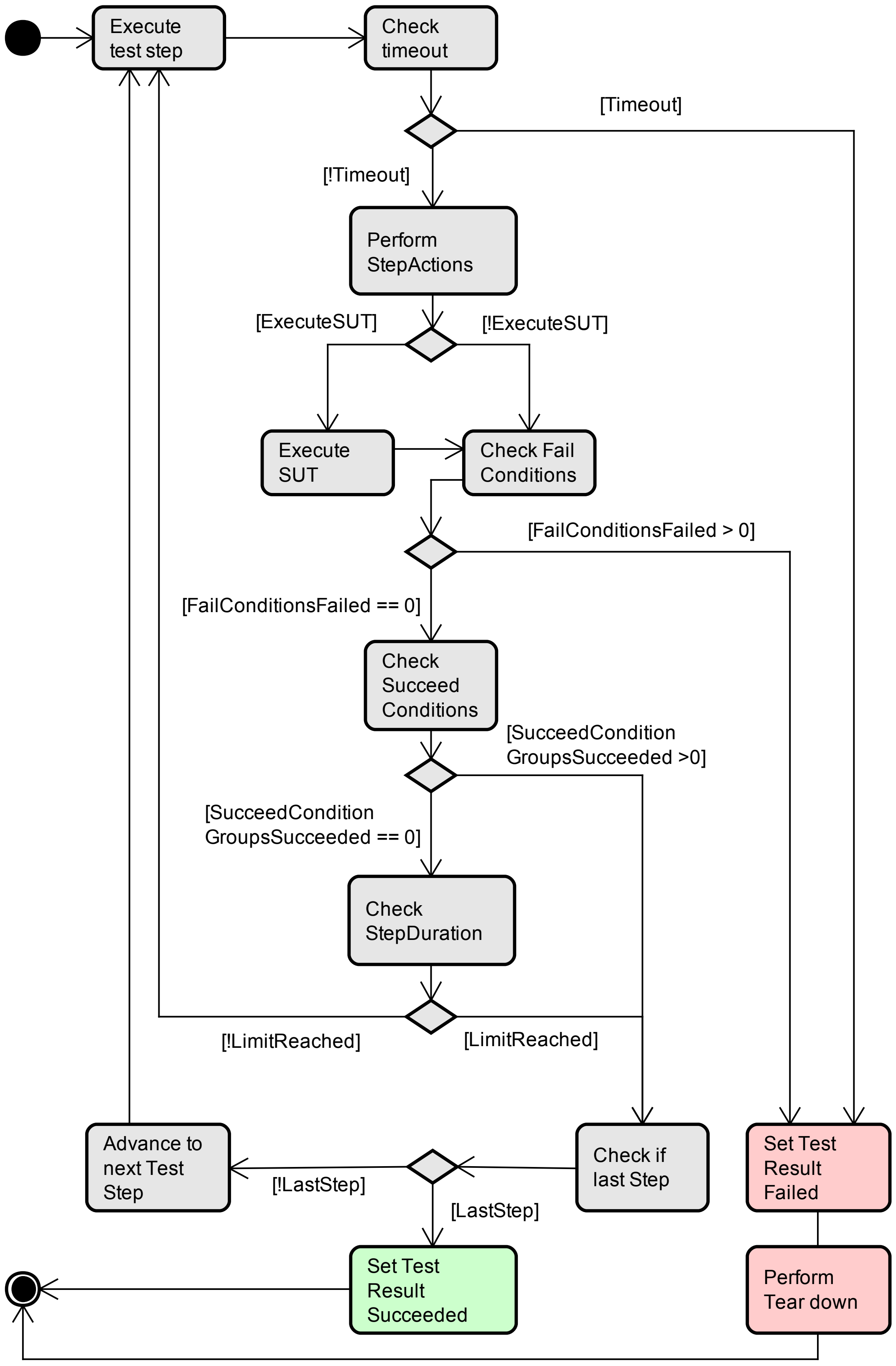


Figure 16: System Test Loop

The test case metamodel is used to define system tests. While unit tests generally directly relate to an SUT interface, system tests require the inclusion of a tester and thus a connection to an HMI. This interaction between test case and HMI is enabled by defining shared variables for the task text (`taskText: STRING`) and the buttons of the HMI (`nokButton: BOOL` and `okButton: BOOL`) as described in Chapter 5.2.1. The `taskText` is changed by the test case using a StepAction, which results in the text to be displayed on the HMI. Conversely, the button values (`true` means the button is pressed) is written by the visualization and checked by the test case using StepConditions.

As schematically shown in Figure 17, a system test usually consists of several steps that generally include interaction with the HMI by setting the task text using a SetValue StepAction, checking whether the `nokButton` was pressed using a FailCondition and including the jump to the next test step using SucceedConditions observing the `okButton` or a defined value of another global variable. Thus, the duration of a generic step in system tests is often not exactly defined, but relating to an event (button pressed, sensor triggered, etc.). Test steps in system tests can be exclusively relating to manual actions (via HMI) or automatic actions (setting and checking software variables), but can also be a mixture of both.

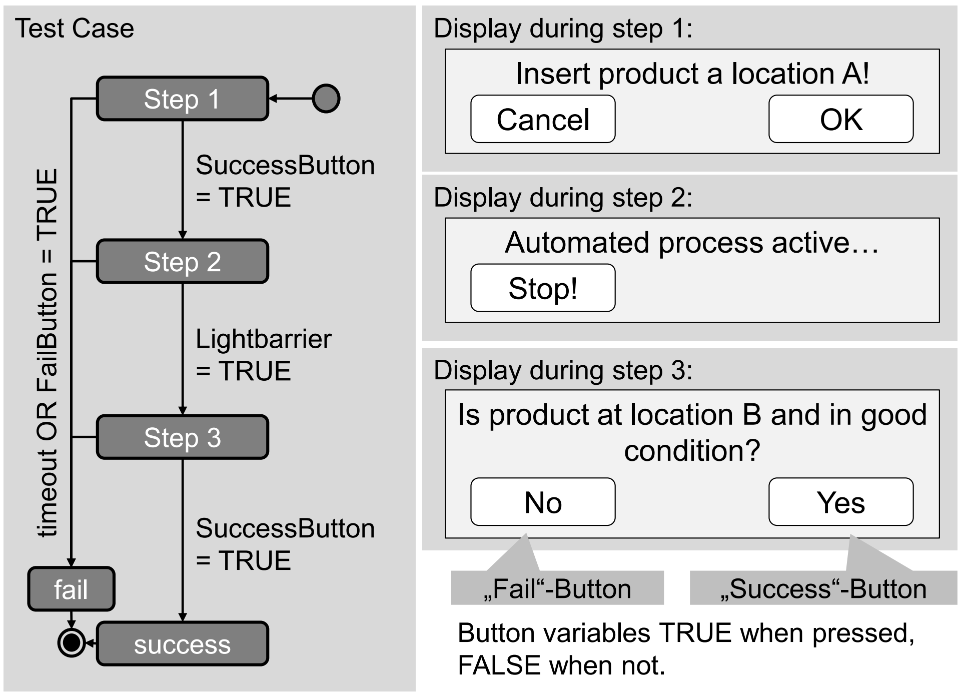


Figure 17: Basic System Test: The schematic test case (left) will change the test HMI (right) to display different tasks, which have to be acknowledged (Ulewicz and Vogel-Heuser, 2016a)

5.2.3 Test Suite and Test Execution History

In line with the requirement R_{Doc}, the approach is to enable a detailed documentation of the testing process. For this, objects for storing information about the *test suite* and the *test execution history* were defined. As schematically shown in Figure 18, each PLC software project can contain one test suite and one test history.

As defined in Chapter 2.2.3, a test suite is defined as a set of test cases in a specific order. For this reason, it aggregates all system tests within the PLC project and possesses two lists for defining a test case order and selection (TestCaseOrder and TestCaseSelection).

Whenever a test suite is executed, the resulting information about the results and durations of each test case is stored (TestSuiteExecutionInformation and TestExecutionInformation). The information about the execution of each test case can be directly related to the system test object. As each test case can be executed multiple times relating to different PLC software project revisions, multiple TestExecutionInformation-objects can be associated with one system test.

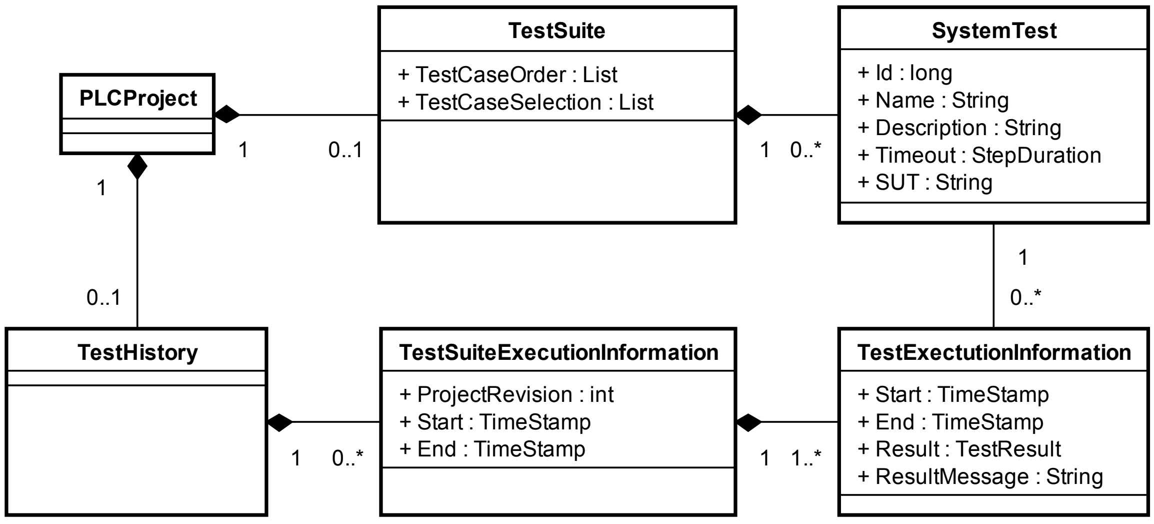


Figure 18: Metamodel for storing information about test suites and the test history of a PLC software project

Using this metamodel, detailed information about the test history can be stored and related to test cases.

5.2.4 Test Bed and PLC Software Project Generation

The starting point for generating a PLC software project that includes the possibility to execute the specified system tests is an original, unaltered PLC software project. The contents of such a PLC program running on a PLC runtime is depicted schematically in Figure 19 (left): a program is interacting with controlled hardware and an HMI via globally defined variables. In addition, global variables for system-wide storage of information are common. The program is depicted as one POU, yet it usually consists of calls and sub-calls of other POUs and merely represents the entry point into this network of calls.

Based on this original PLC software project, several additional items are generated into the project (see Figure 19, right side). Instead of directly executing the original Program cyclically, the PLC runtime is changed to call a *test bed block*. This test bed block continues calling the original program, which is now the SUT. Yet, additional function calls are implemented before and after this call: Based on the test cases which are stored in global variable arrays on the PLC runtime, *actions are performed* before the call, and *conditions are checked* after the call, by writing on specific global variables and reading from them respectively. To allow for an interaction with the tester, additional HMI resources are integrated into the PLC software project. During the testing process, tracing is performed (see 5.3.2 for more details), recording relevant information during the test case execution for later use in documentation and analysis of test coverage.

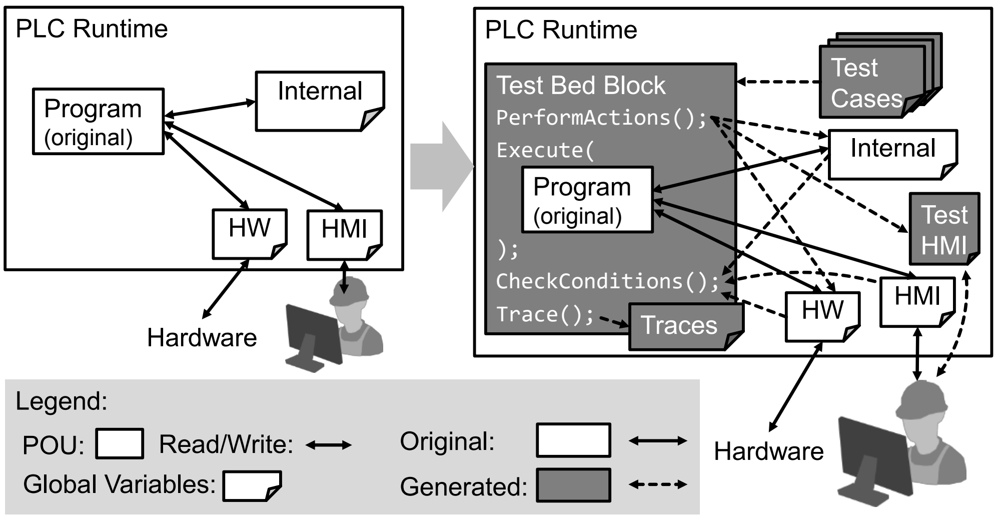


Figure 19: Test project generation (based on (Ulewicz and Vogel-Heuser, 2016a)*)*

The test cases are stored as objects within the control software project. Using the functionality provided by the CODESYS TEST MANAGER (CODESYS, 2015), these test cases can be automatically converted into executable test cases. The tool can, among other things, upload the generated project onto any execution hardware, start the testing process, generate a test report and download any specified files from the execution hardware. This functionality is used in combination with the generation of additional tracing POUs and variables and the insertion of the HMI components for displaying test case information on a display and enabling user input during test execution, respectively.

5.3 Coverage Assessment for System Tests

The guided semi-automatic system testing approach (Chapter 5.2) sets the foundation for measurements during testing and relating test cases to these measurements. This chapter describes the considerations and developments leading from the possibility to record data during testing to an approach for coverage assessment in system testing for identification of untested behavior (R_{TA}). This part of the approach was preliminarily published in (Ulewicz and Vogel-Heuser, 2018a).

Starting from identifying a suitable coverage metric for identifying untested functionality (Section 5.3.1), the basic concept for coverage assessment is presented in Section 5.3.2. The required modifications to the programs and the way this is achieved are presented in Section 5.3.3. The acquired coverage information is related to the coverage measure (Section 5.3.4) and visualized for a quick identification of untested functionality (Section 5.3.5).

5.3.1 Identifying a Suitable Coverage Metric

As described in the state of the art (Section 4.1.4.2), multiple different coverage metrics have been developed in the field of computer science. Each metric has positive and negative aspects regarding the presented problem (identifying untested behavior), which were analyzed to choose a suitable metric for the given requirements of insignificant influence on real-time behavior (R_{RT}) and providing support in assessing test adequacy by identifying untested system behavior (R_{TA}).

Requirements based metrics are generally not suitable if detailed functional specifications are not available, which is often the case in aPS engineering. A coverage cannot be calculated if the relation between detailed functional specifications cannot be made. In addition, unneeded functions, i.e. unneeded code, cannot be identified as these would not be specified even with detailed specifications. This type of coverage metric was therefore found not to be suitable for the presented approach and was excluded from further consideration.

In contrast to this, code structure based metrics can be calculated without the need for additional detailed functional specifications. In the field of computer science, different metrics were developed and checked for their suitability for assessing test suite adequacy, i.e. whether a test suit comprised of multiple test cases covers all relevant behavior in the system. *Statement coverage* was found to be very effective in detecting mutations, i.e. defects, in code (Gopinath, Jensen and Groce, 2014). For identifying non-adequate test suites, i.e. test suites missing test cases, statement coverage does not seem to have any downsides compared to more detailed criteria (Gligoric *et al.*, 2013). In addition, to record these detailed metrics, more detailed instrumentation is required: decisions need to be analyzed in more detail and more memory is needed to store the information in case more complicated decisions are present. According to requirement R_{TA} and R_{Mem}, both available execution time and memory are critical and statement coverage is expected to require less of both in comparison to more complex metrics, such as condition/decision coverage. Industrial application was also expected to yield complex coverage results to be evaluated by the tester, which would be amplified by the even more detailed results from other metrics. Therefore, for this approach, statement coverage was more promising for the requirement of minimal influence on real time properties of the system and was subsequently chosen for the presented approach.

5.3.2 Assessing Test Coverage using Statement Coverage

Based on the finding that statement coverage represents a promising coverage measure, the conception of the basic concept of assessing coverage was developed. In Section 5.3.2.1 a concept for recording statement coverage is presented, which is subsequently implemented into the process of system testing, as described in Section 5.3.2.2.

5.3.2.1 Basic Concept for Recording Statement Coverage Information

The calculation of statement coverage requires information about which statements were covered during code execution. Statements are regarded as covered if they are executed. To find out whether a particular statement was executed, another statement recording its own execution can be placed right after or in front of the statement of interest. Given, designated memory for storing this information exists, e.g. an array with one element for each statement, which can be set to TRUE if the respective statement was visited, all required information for statement coverage can be recorded in memory and real-time (see Figure 20). This technique requires an exact relation between array index and the respective statement, which has to be saved during instrumentation (insertion of the recording statement).

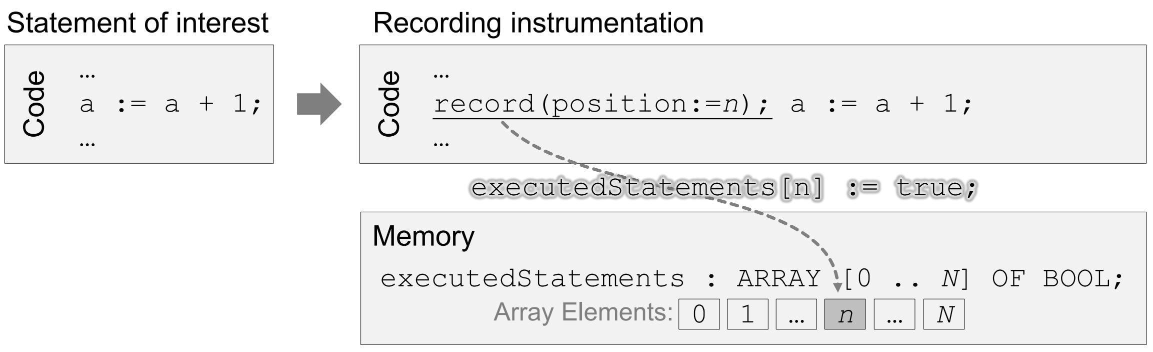

Statement of interest
Code
...
a := a + 1;
...
Recording instrumentation
Code
...
record(position:=n); a := a + 1;
...
executedStatements[n] := true;
Memory
executedStatements : ARRAY [0 .. N] OF BOOL;
Array Elements: 0 1 ... n ... N

Figure 20: A method to record statement coverage: insertion of a record function call just before the statement of interest, which changes entries in an array

Yet, using this technique in this simple form, the code required for recording whether any of the investigated statements was executed would increase the source code dramatically. This would go in hand with high increases in required execution time. Yet, when regarding the properties of a control flow (see Chapter 2.3), the number of required recording statements can be drastically reduced.

A simple control flow comprises of *jumps in the control flow* and *basic blocks*. While jumps are generated when implementing control statements (`IF-THEN-ELSE`, `WHILE`, …), basic blocks are made up of most other statements (assignments, calculations, …). As basic blocks are defined to only have one entry and one exit point in the control flow, it can be derived that all statements within this basic block are executed, if one of the statements was executed. It is thus sufficient to only record once per basic block whether it was covered, instead of recording this information for each individual statement. As control statements connect basic blocks, these statements do not have to be recorded, as it can be derived that they were executed if both the preceding as well as the subsequent basic block was executed. To conclude, at least every basic blocks execution is to be recorded in order to find uncovered statements. In fact, the number of required records could be further reduced by taking mutually exclusive paths of the code into account, such as `IF-THEN-ELSE` decisions, where either the `THEN`- or the `ELSE`-part would have to be recorded. Yet, this optimization was not implemented into the approach yet.

Based on the knowledge of what information needs to be acquired to find untested statements, the method for efficient code instrumentation was developed, which is described in detail in Section 5.3.3. Before going into detail about the instrumentation, the basic concept for recording coverage information in the context of executing a complete test suite will be described in the next section.

5.3.2.2 Basic Concept for Recording Traces and Calculating Coverage

Before traces can be recorded, the original PLC software projects are required to be instrumented to implement the basic concept for recording statement coverage as described in the previous section. The instrumentation will be described in detail in Section 5.3.3. The instrumentation achieves two things: 1) the control code is extended by functions, function calls, and designated memory (*trace array*) to allow for recording traces and 2) information about the relation of the function calls to decipher recorded traces are saved. The latter information is required after all tests have been executed to allow for the relation of each code to covered code. To enable an execution of test cases, the instrumented project is extended by generated executable test cases as described in Section 5.2.4 and uploaded onto the PLC for test execution. The test execution is similar to the “pure” guided semi-automatic system testing approach (Section 5.2), yet before each test case the volatile memory storing trace information (trace array) is cleared, filled up again during test execution, and saved to non-volatile memory after each test. This distinction between volatile and non-volatile memory reduces the impact of the recording on real-time properties, as volatile memory typically is significantly quicker to be read and written. The result is a trace file for each test case, storing the coverage information for each respective test case. After testing, the traces stored on the non-volatile memory of the PLC are transferred to the engineering system. Traces are superimposed to acquire information about the overall coverage and the uncovered parts of the code in particular. This information is subsequently visualized to enable a coverage assessment by the human testing engineer or technician (see Section 5.3.5). The relation between the previously described steps is schematically shown in Figure 21.

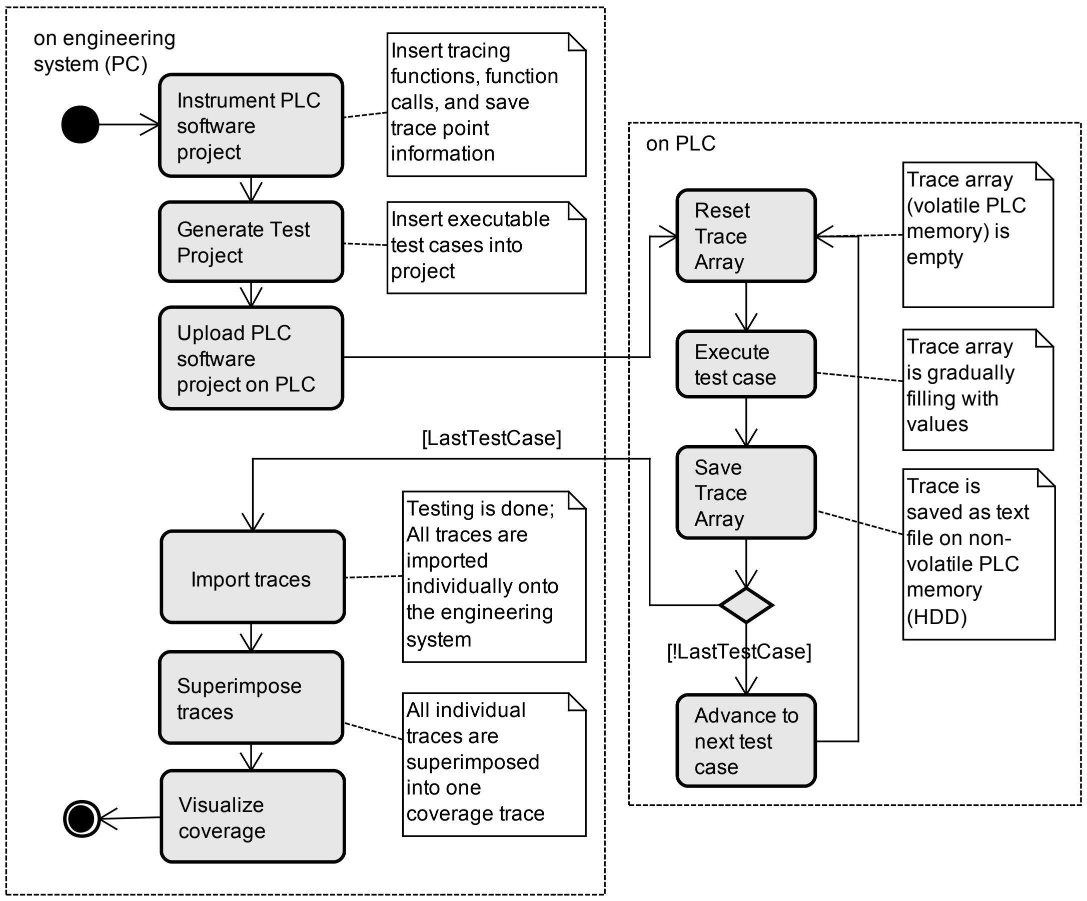

on engineering system (PC)
Instrument PLC software project
Insert tracing functions, function calls, and save trace point information
Generate Test Project
Insert executable test cases into project
Upload PLC software project on PLC
[LastTestCase]
Import traces
Testing is done; All traces are imported individually onto the engineering system
Superimpose traces
All individual traces are superimposed into one coverage trace
Visualize coverage
on PLC
Reset Trace Array
Trace array (volatile PLC memory) is empty
Execute test case
Trace array is gradually filling with values
Save Trace Array
Trace is saved as text file on non-volatile PLC memory (HDD)
[!LastTestCase]
Advance to next test case

Figure 21: A detailed overview of the coverage calculation concept: the PLC software project is instrumented on the engineering system, executed on the PLC while traces are recorded. Traces are subsequently imported into the engineering system and analyzed.

5.3.3 Preparing the PLC Software Project for Execution Tracing

An unmodified PLC software project does not allow for recording the information described in the previous sections. For this, it has to be instrumented. This means that special functions for tracing are added to the project, function calls are directly added into the code, and memory is designated for recording this trace information during runtime. The designated memory is realized in the form of an array in the PLC software program with an array entry for each basic block. To enable a direct relation between the source code and the trace array, information about the relation of array entries, basic blocks and their position within the code is stored during instrumentation. The instrumentation is based on a so-called *dependency model*, which will be described in Section 5.3.3.1. The model is used to find and add tracepoints, i.e. points at which basic block execution is recorded. This and the inclusion of the functions for tracing will be described in Section 5.3.3.2.

5.3.3.1 The Dependency Model

The dependency model used for identifying decisive points and basic blocks within the code is an extension of the dependency model definition presented in (Feldmann, Hauer, *et al.*, 2016).

The original dependency model was designed to analyze programs for modularity and other maintainability properties by analyzing the control flow as well as the data flow within control programs. As this analysis did not consider the control flow within program organization units (POUs), the metamodel, which is presented in Figure 22, was extended by suitable stereotypes to be able to represent these features in a generated model. The extended stereotypes are shown in light gray in the figure. The dependency model contains information about the control flow, data flow, and call dependencies within and between PLC software entities, POUs, and variables in particular.

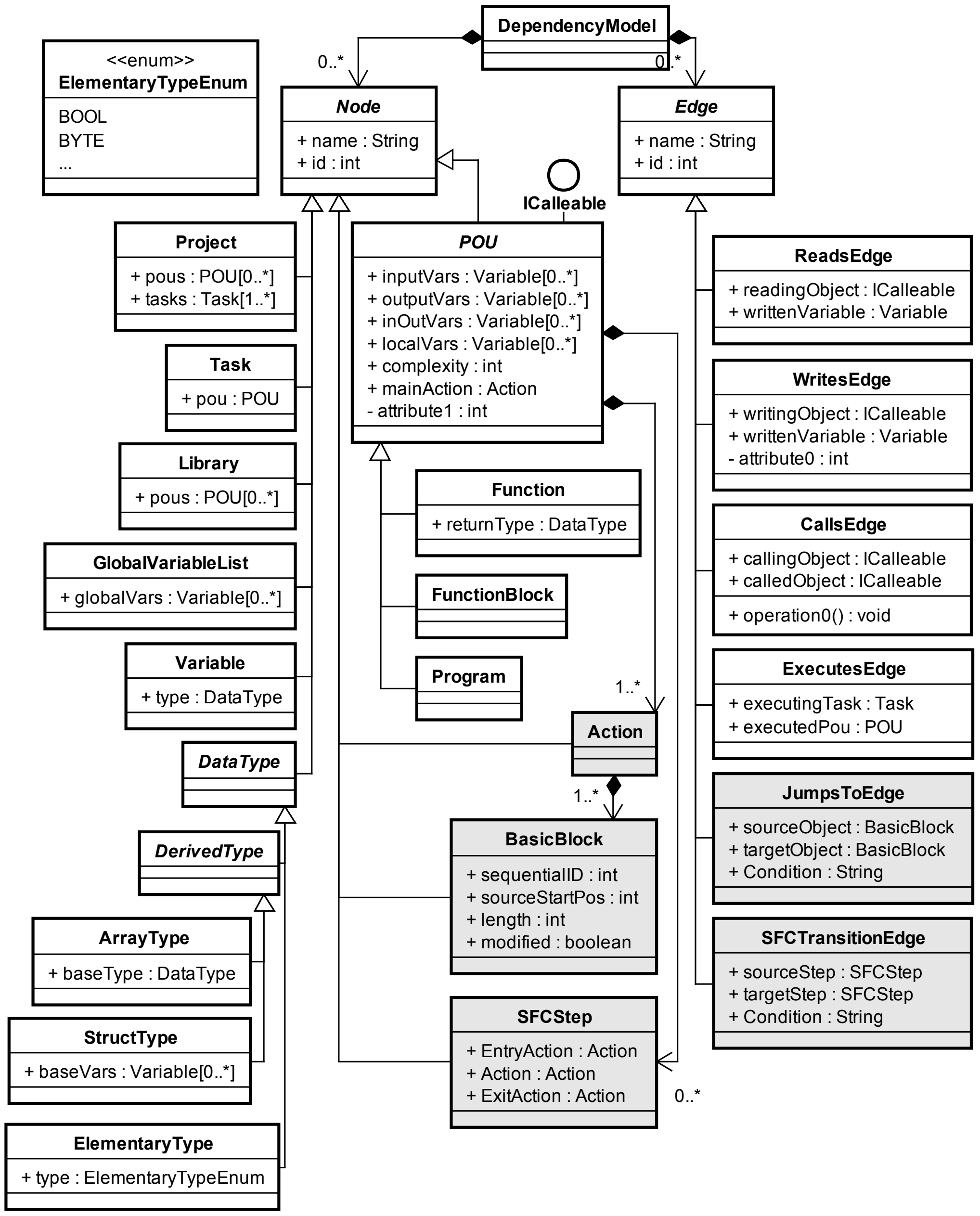
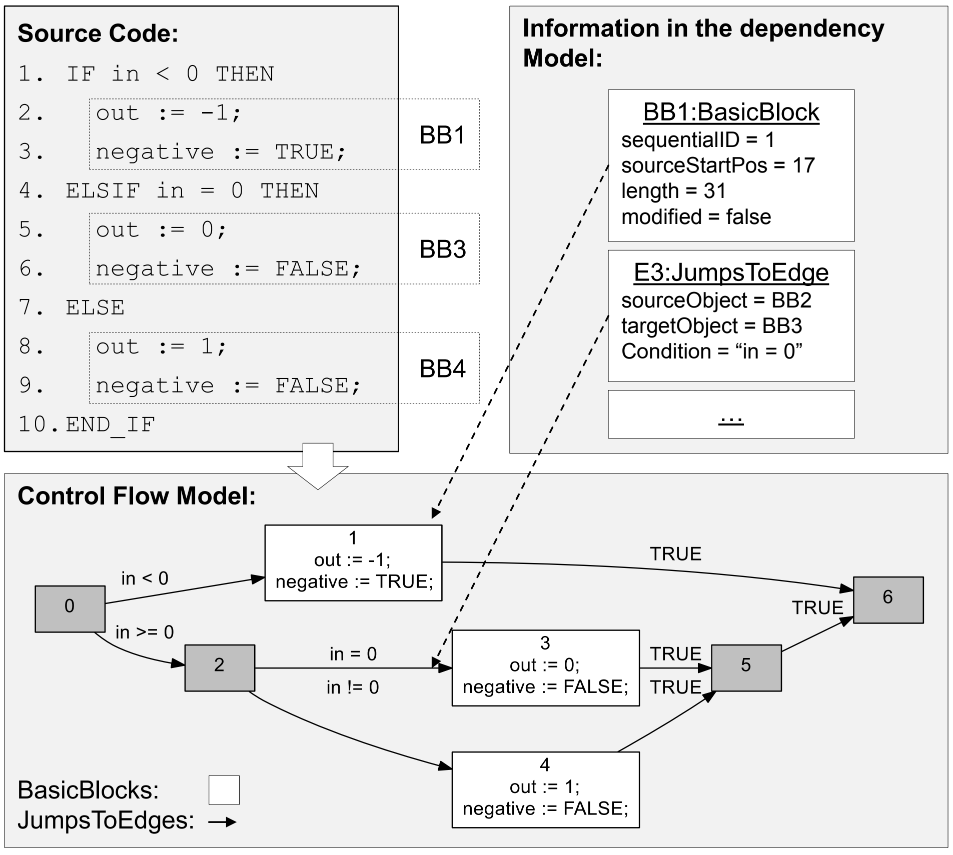


Figure 22: Dependency metamodel for code analysis and instrumentation (extension of (Feldmann, Hauer, *et al.*, 2016)*; extensions filled light gray)*

The dependency model is a directed graph consisting of nodes and edges. Nodes represent structural entities of an IEC 61131-3 project, whereas edges represent the dependencies between these entities (Feldmann, Hauer, *et al.*, 2016). An edge connects two nodes, a source node, and a target node, in one direction. The metamodel is able to contain nodes from different hierarchies in the project, starting from the project itself, the defined tasks (threads) in the project, the POUs (functions, function blocks and functions) called by the tasks and other POUs. This was extended by code elements such as actions (functions embedded in POUs) and basic blocks (code segments that do not contain decisions such as if-statements). For SFC, the step node was implemented, which can itself contain multiple actions per step. In the current version of the approach, actions with ST implementation and the type qualifiers P0 (single execution upon step deactivation), P1 (single execution upon step activation) and N (repeated execution while step being active) are supported. The edges represent dependencies between the nodes, such as calls between POUs, write operations on variables and, in the extended model, also progressions between basic blocks (JumpsToEdge) and SFC-steps (SFCTransitionEdge).


Source Code:

```
IF in < 0 THEN
   out := -1;
   negative := TRUE;
ELSIF in = 0 THEN
   out := 0;
   negative := FALSE;
ELSE
   out := 1;
   negative := FALSE;
END_IF
```

Information in the dependency Model:

BB1:BasicBlock
sequentialID = 1
sourceStartPos = 17
length = 31
modified = false

E3:JumpsToEdge
sourceObject = BB2
targetObject = BB3
Condition = “in = 0”

...

Figure 23: Example for a CFG (bottom) generated from code (top left) and an excerpt of contained information in the dependency model (top right)

The JumpsToEdge is generated from control statements, such as if-statements, and additionally stores the condition as expressed in the if-statement or implicitly expressed in the else-statement. For example, if there is an if-statement, the `then`-part of the if-statement will be represented as an edge leading from the code before the `if`- to the `then`-part with the condition specified in the if-statement. If there is an `else`-part of the statement, there will be an additional edge leading from the code before the if-statement to the `else`-part with the inverted condition of the if-statement (see Figure 23 for a practical example).

If a basic block calls another POU, there will be an edge leading from the first function block to the initial basic block within the called POU without a condition. The same applies for the SFCTransitionEdge: Transitions in SFC-charts are converted to transition edges, showing the connection between SFC steps as defined in the SFC chart.

5.3.3.2 Instrumentation of the PLC Software Project

Based on the metamodel presented in the previous section, a dependency model can be automatically generated from the source code of the PLC program. This is done by identifying all defined tasks as initial points for code exploration. The called POUs are identified from each initial point. The code specified within these POUs (its implementation body and, if available, its actions) is converted to an abstract syntax tree (AST), which is then walked through iteratively. During this walk-through, basic blocks and control statements connecting these blocks are identified and saved as nodes and edges. In case a basic block calls another POU, this POU is equally walked through until the end of the code is reached. In this process, each basic block is sequentially numbered (sequentialID, see Figure 23, top right), to allow for an easy correlation between basic blocks and trace data. The result is a call graph spanning the control software project's code, which is relevant for the control flow, omitting POUs which will never be called. These POUs will already be identified by the compiler and are thus of no further relevance. The created graph is the basis for code instrumentation, explained in more detail in the next section.

The recording of data needed to infer execution traces is achieved by instrumenting the source code of the PLC project and saving information about inserted parts within a database. The instrumentation consists of inserting function calls in the beginning of each basic block and allocating memory for temporal storage of execution trace information. The trace information is realized as an array of Boolean variables for each basic block (`tpa: ARRAY[0..MAXTP] OF BOOL`, where `MAXTP` is the total number of traced basic blocks)[2]. The array is reused for each test case by resetting each entry before each test case and saving the recorded information after each test case. For this, two functions and one function block were developed:

- Reset function `tp_reset()`: The reset function is called before each test case and is used to reset the complete trace array to ensure that all array items are set to their initial state (`false`).

[2] The trace array was later replaced by a trace structure, containing the same information, but improving the performance of the approach (see Chapter 8.1).

- Record function `tpr(INT i)`: This function, which is called at each tracepoint, is given the identification number of the tracepoint after which the related array item in the trace array is set to `true`. In its simplest form, it only consists of one line of code: `tpa[i]:=TRUE;` This dedicated call-by-value function was later replaced by inserting the tracing code directly at the respective position in the source code for performance reasons (see Chapter 8.1).

- Saving function block `tp_save(BOOL xExecute, STRING szFilename):` After each completed test case (failed or successful), this function will be called to save the information stored within the trace array into a common text file on the execution hardware. As this process might take several PLC scan cycles, the writing process needs to be completed (Output `xDone = TRUE`) before the next test case is initiated. The data saved is the ID and value of each tracepoint (e.g. "`1: true, 2: false, 3: false, …`").

These POUs are inserted into the project alongside the trace array at the instrumentation phase of the control software project. Using the information collected in the dependency model, each basic block is instrumented with a tracepoint. Function calls of the `tpr`-Function are inserted into the code using the information about the location of the basic blocks (see Figure 24).

Original code:	Instrumented code:
`IF in < 0 THEN`	`tpr(i:=42); IF in < 0 THEN`
`  out := -1;`	`  tpr(i:=43); out := -1;`
`  negative := TRUE;`	`  negative := TRUE;`
`ELSIF in = 0 THEN`	`ELSIF in = 0 THEN`
`  out := 0;`	`  tpr(i:=44); out := 0;`
`  negative := FALSE;`	`  negative := FALSE;`
`ELSE`	`ELSE`
`  out := 1;`	`  tpr(i:=45); out := 1;`
`  negative := FALSE;`	`  negative := FALSE;`
`END_IF`	`END_IF`

Figure 24: Instrumentation example: The original code (left) is extended by function calls resulting in instrumented code (right)

5.3.4 Relating Test Cases to Code

Alongside the instrumentation of the code (Section 5.3.3.2), a tracepoint database is created allowing for the relation between the instrumented code, the dependency model and the execution trace information created during test execution. The database contains information about the related basic block and thus about the tracepoint location (sourceStartPos) of each inserted part within the code. In addition to this information, an entity named "visit" will be filled with information from the execution traces after the testing process is finished: each test case will create an execution trace stating if a tracepoint was "visited" or not. Thus, each tracepoint in the tracepoint database can be "visited" by each test case. This information is combined to identify which test cases were not visited by any test case (`WasVisited()` evaluates to `false`) to identify untested parts of the source code. A practical, simplified example for this is given in Figure 25: Three traces are combined into one trace. Each trace stores information about whether each of three basic blocks was visited. Combining the three traces reveals that basic block 3 was not visited by either of the test cases.

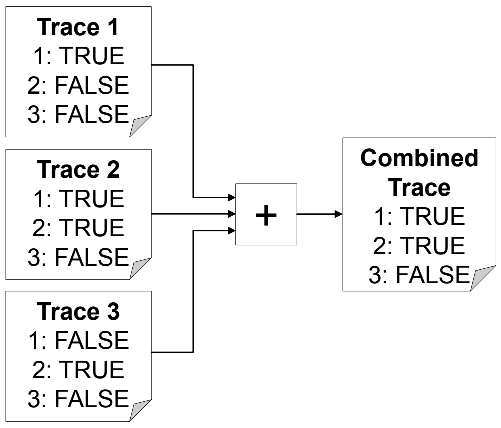


Figure 25: Generic example for combining statement coverage traces

The connection between the basic block to the individual system test is shown in Figure 26: a SystemTest (bottom right) has TestExecutionInformation, if it was previously executed, as extracted from a test report returned by the CODESYS TEST MANAGER upon completing the testing process. Information about which test case visited which tracepoint can be extracted from the imported execution traces. This information can directly be linked to a system test execution, i.e. a TestExecutionInformation.

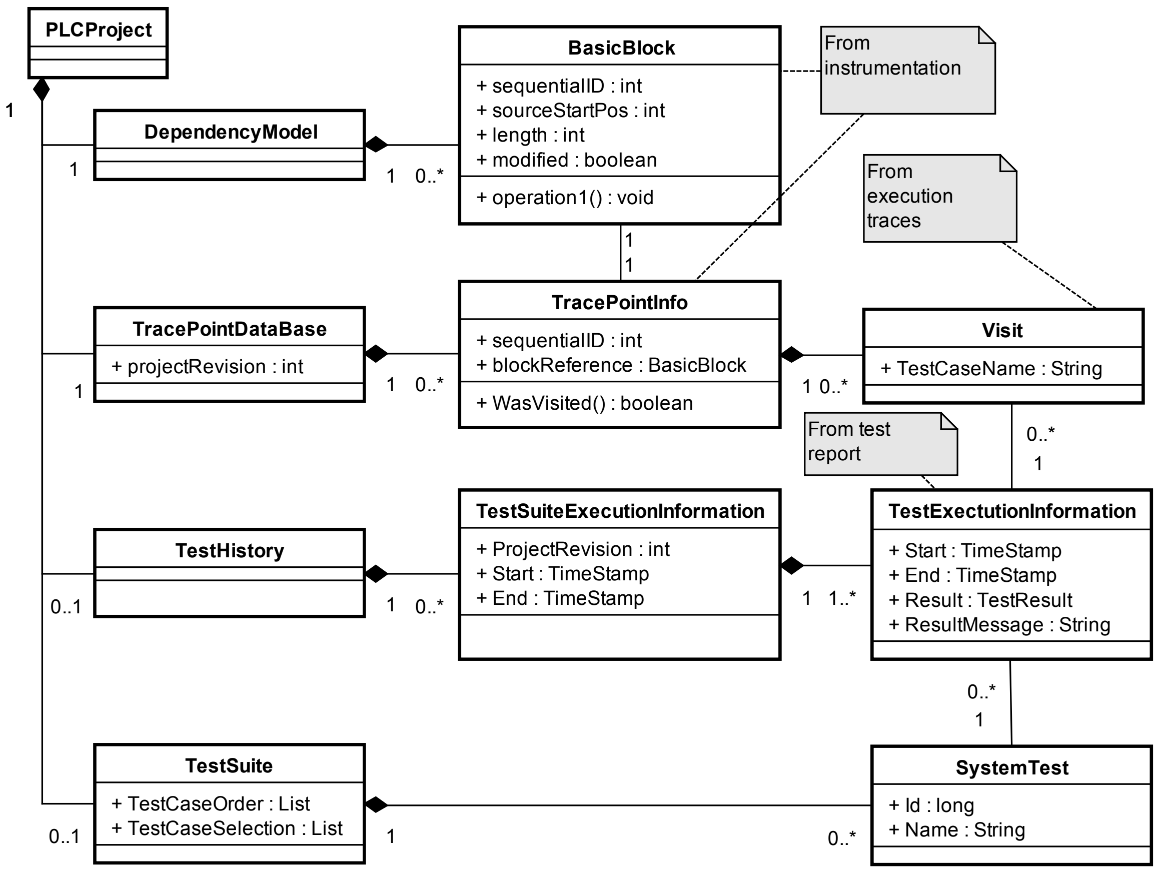


Figure 26: Connection to code and test cases

Information about all tracepoints and their connection is stored during instrumentation. Thus, the connection between system test and basic block is closed and the coverage of each basic block can be inferred. For handling of this information, a concept for visualizing this information was developed which will be presented in the next section.

5.3.5 Visualizing Test Coverage

As pointed out by Piwarowski (Piwowarski, Ohba and Caruso, 1993) and Yang (Yang, Li and Weiss, 2009), high coverage scores are difficult to achieve even regarding statement coverage. This may be due to unreachable code or complex conditions, among other reasons. This fact was also pointed out by the industry partners questioned in the initial requirements study: testing all behavior in every detail in an automated production system is not feasible. One reason why this is not possible in this particular field of industry is economics: testing is resource-intensive and performed under significant time pressure.

As the goal of the presented approach is to identify the untested behavior of the system, a quantitative measure as in a coverage percentage seemed unnecessary or unsuitable. Instead, a visual emphasis of untested code was chosen. This also allows for the detailed investigation by the tester to evaluate whether the untested parts are indeed critical and therefore might require additional test cases. Inspired by a traffic light color scheme, untested parts are marked "red", i.e. need investigation, and partially tested parts are marked "yellow", i.e. potentially critical. A "green" marking was deliberately not used as parts of the system that were fully covered might still contain faults (testing is *"hopelessly inadequate"* to show the absence of faults (Dijkstra, 1972)).

For a quick assessment of the coverage of the system, different views were chosen aggregating the underlying coverage (see Figure 27). In a software project call graph, all executable POUs are depicted starting from the task calling the first POU. Each POU is marked "yellow" or "red" depending on whether the steps (in the case of a POU programmed in SFC), actions or its basic blocks were only partially covered or not covered at all. Views that are more detailed are presented by clicking on the respective POUs. In the case of POUs programmed in SFC, the individual steps, as well as their transitions, are shown with a similar color-coding. The level closest to the code is a view depicting individual basic blocks.

For future work, a direct implementation into the development environment's editors could support industrial acceptance of the approach as no new concepts would have to be learned. A mock-up of this idea is depicted in Figure 28.

By allowing for the tester to quickly browse through the project to identify untested parts of the system, a quick ability to detect the untested behavior of the code is expected. If a complete POU is marked as untested, the user can quickly look into the code and decide whether this block was previously tested or needs further investigation. If an automatic step chain was only partially covered, the tester can identify the untested steps, which often correspond directly to behavior in the machine, and analyze the item for further investigation. This process can also be performed down to the basic block level, where individual lines can be identified as untested, as critical behavior or be deliberately omitted.

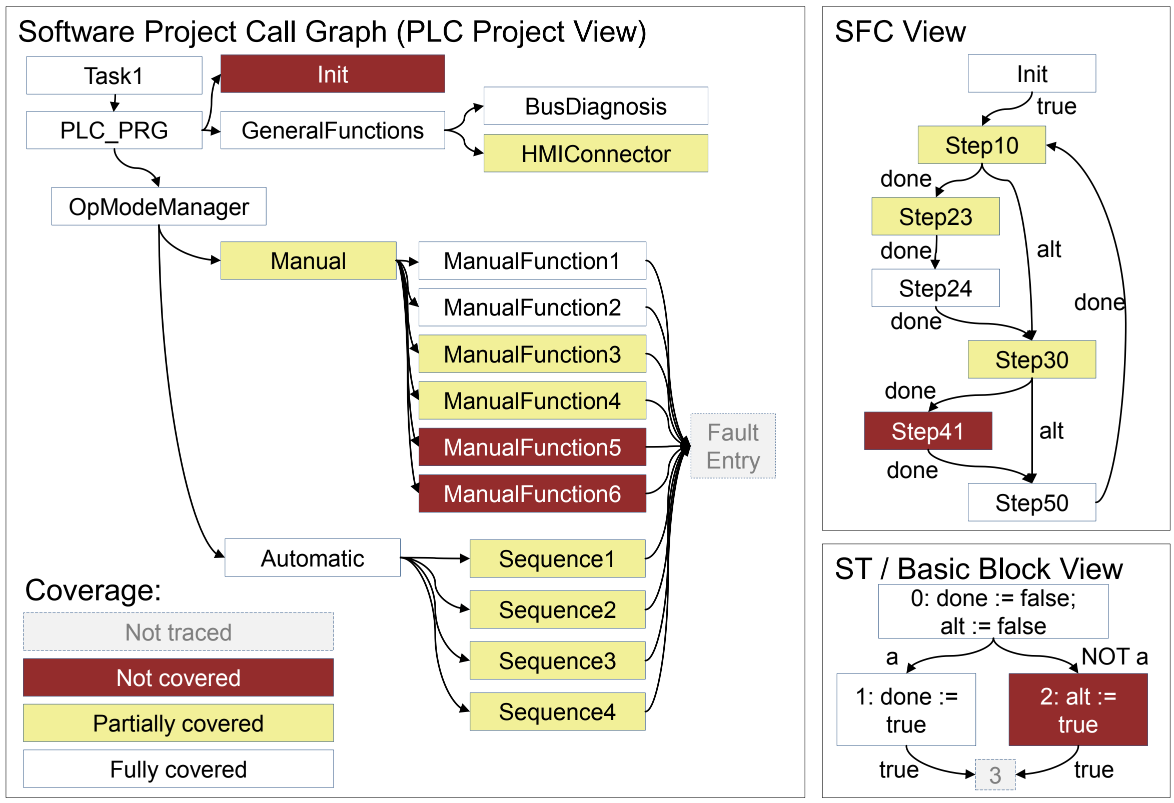
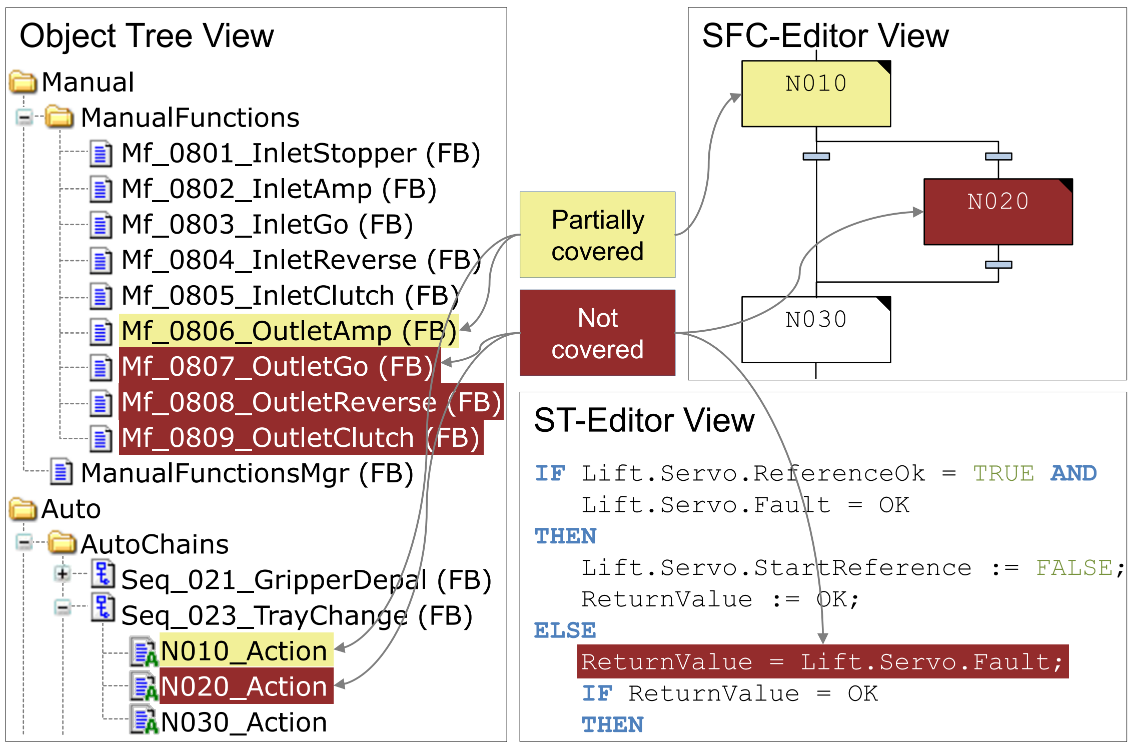


Figure 27: Hierarchical coverage views as developed for the approach: A software project call graph (left) gives a quick overview of covered (light gray and white) and uncovered (dark gray) POUs. More detailed views can unveil uncovered parts of the code from SFC level (upper right) to ST level (lower right).



```
IF Lift.Servo.ReferenceOk = TRUE AND
   Lift.Servo.Fault = OK
THEN
   Lift.Servo.StartReference := FALSE;
   ReturnValue := OK;
ELSE
   ReturnValue = Lift.Servo.Fault;
   IF ReturnValue = OK
   THEN
```

Figure 28: Coverage Visualization Concept

5.4 Prioritization of System Tests for Regression Testing

Regression testing focuses on prioritization of system tests for a previously tested system that has undergone changes. The goal of the prioritization is to efficiently find newly introduced faults using existing, previously successfully executed test cases. To enable this efficient regression testing process, the system tests are arranged in a way that test cases with a higher probability to find newly introduced faults are moved to the front of the queue of test cases to be executed (test prioritization). Through this, possible regressions of the system can be found earlier, enabling an optimized iterative debugging process (fixing the regression and repeating the regression testing process). This part of the approach was preliminarily published in (Ulewicz and Vogel-Heuser, 2016b, 2018b).

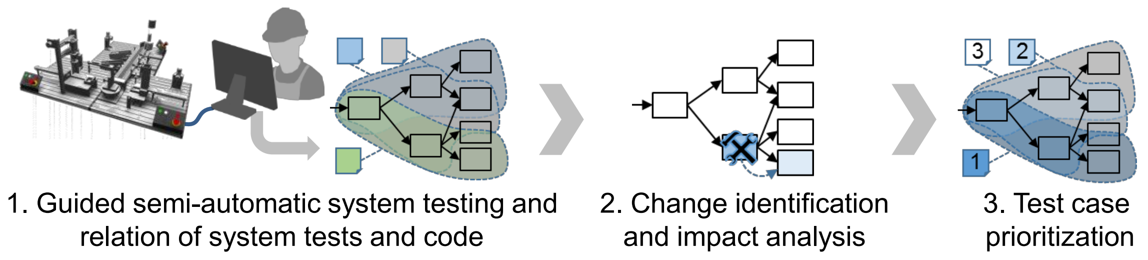

3
2
1
1. Guided semi-automatic system testing and relation of system tests and code
2. Change identification and impact analysis
3. Test case prioritization

Figure 29: System test prioritization in three steps

The approach comprises three steps:

Step 1 (see 5.4.1): Building a relation between test cases and the executed control program parts and acquiring timing information for each test case of the unchanged program.

Step 2 (see 5.4.2 and 5.4.3): Identifying changes in the changed program and possible impacts of the changes on the rest of the program.

Step 3 (see 5.4.4 and 5.4.5): Prioritizing system test cases for the changed program according to the possible impact of the change and the acquired runtime and timing information of each test case of the unchanged program.

The prioritization was developed with a similar idea as the work presented by Orso et al. (Orso, Apiwattanapong and Harrold, 2003): test cases that previously executed parts of the code that have now been changed or are affected by changes are more likely to yield different results. Thus, these test cases are more likely to find new faults and are given a higher priority.

As semi-automatic test cases require significant amounts of time for execution, two refined prioritization methods were developed to increase the efficiency of this basic prioritization further. Taking timing information about the test cases into account, these refined methods aim at 1) intensely testing changes to find sporadic faults and 2) testing all changes as quickly as possible. While the execution traces gathered during coverage assessment was already enough for the basic prioritization, the refined techniques required an extension of the tracing algorithm, which is described in Section 5.4.1.

5.4.1 Building a Relation between Test Cases and Executed Control Program Parts and Acquiring Timing Information

The relation built between test cases and executed code during coverage assessment (see Section 5.3.4) already allows for a basic relation between system tests and each basic block: Each executed SystemTest results in TestExecutionInformation, which can be related to Visits of TracePoints, which in turn are directly related to BasicBlocks (see Figure 26). Thus, if a basic block changed, the test cases previously executing this block can be identified, which – in combination with a change impact analysis (see 5.4.3) – allows for a basic prioritization (see 5.4.4).

As briefly mentioned in the previous section, two refined prioritization methods (see Section 5.4.5 for details) were developed in addition to the basic prioritization (see 5.4.4). These methods additionally take timing information and intensity of "visiting" basic blocks into account.

The information needed for the refined prioritization methods are the time to execute each test case (can be extracted from test report) and the number of times the test case passed through each part of the code and when it passed through it for the first time (needs additional instrumentation). Similarly to the testing coverage concept (Section 5.3), the code is instrumented: the original project is extended by functions and function calls automatically. For the refined prioritization methods, the tracing function and the trace array (see 5.3.4) were extended by four further variants.

Variant 1 - "Traversal": The same variant as used for coverage assessment. This tracing function will solely mark entries in the trace array as "true", hence only recording information, whether a part of the code was executed or not.

Variant 2 - "Intensity": The tracing function will increment the corresponding trace array entry, thus recording information about how often a part of the code was executed by the currently active test case. In contrast to variant 1, an array of integer values instead of Booleans is required for recording.

Variant 3 - "Quickness": The tracing function will record the time since starting the test case if traversed for the first time. With this method, the first time of traversal for each point of the code is recorded. Similarly to variant 2, a trace array of integer values is required.

Variant 4 - "All": This tracing function combines variant 2 and 3 to allow for recording of both information and choose the preferred type of prioritization after the test execution was performed. This variant requires two trace arrays.

These variants were created for comparison reasons regarding the required overhead (see Chapter 7.2.4). In addition to the variants of the tracing function `tpr(int i)`, the Visit-object (see Figure 26) was extended by the attributes visitCount and firstVisitMS, to enable the import and analysis of the recorded data.

5.4.2 Change Identification

Direct changes in the software can be identified by comparing a previous (unchanged) and a current (changed) software revision. Current Integrated Development Environments (IDEs) often already offer syntactic change analysis of control software, but lack identification of changes of the control and data flow. As the presented regression test prioritization is based on the relation between test cases and the program control flow, the comparison is directly performed with the revisions of the system's dependency model. Changes are identified in a top-down manner: *coarse-grained changes* are identified by comparing the items of the dependency model, subsequently, *fine-grained changes* are identified by comparing modified items in more detail.

Coarse-grained changes (software project level): Through comparison of the set of nodes and edges on the project level (POUs and calls), added, removed and modified nodes and edges can be identified. The relation between the items in the sets of old and new are generated by the items' qualified name (unique name based on their parent objects' names and own name). If a node or edge cannot be found in the old revision but the new, it is marked as "added". Conversely, items present in the old set but not the new have been removed. Items of the same name that have a changed checksum of their content are marked as "modified" (fine-grained changes) and are subsequently analyzed in more detail by regarding their sub-items. In addition, modifications on globally specified variables are analyzed in a similar way.

Fine-grained changes (source code level): Modified POUs are analyzed for their internal changes by comparing their control flow with the unchanged revision. Depending on the implementation of the POU, this is directly performed on the basic blocks and decisions stemming from an ST implementation or the SFC steps and transitions of an SFC implementation. Changes to SFC elements are identified similarly to coarse-grained changes: the set of SFC transition is compared by source and target node names as well as the contained conditions. This is followed by a comparison of all SFC steps using the steps' names. Actions related to the SFC steps are then compared, similarly to ST implementations. In contrast to the coarse-grained changes and the comparison of SFC implementations, the nodes of the ST control flow (basic blocks) cannot be easily identified by name as they are consecutively numbered during creation. Small changes in the control flow, i.e. changes that only modify basic blocks or transition conditions, are identified by comparing the control flow graphs (basic blocks and transitions) of the implementations using the consecutive numbering. All modified basic blocks are flagged accordingly. In case transitions were modified, the basic block following the source basic block of the edge are marked as modified, as the tracing algorithm focuses on the traversal of basic blocks.

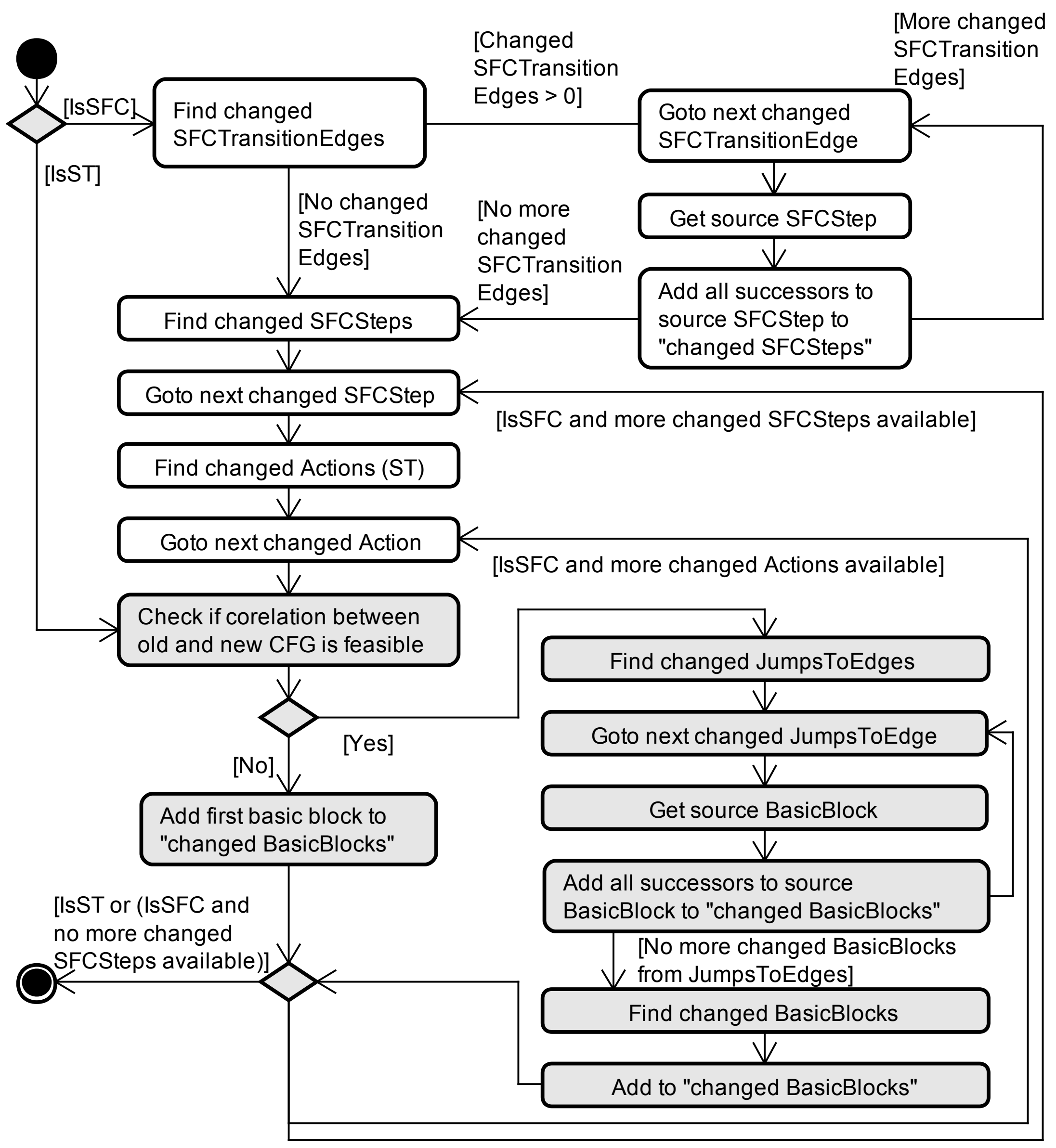

[IsSFC]
Find changed SFCTransitionEdges
[Changed SFCTransition Edges > 0]
Goto next changed SFCTransitionEdge
[More changed SFCTransition Edges]
[IsST]
[No changed SFCTransition Edges]
[No more changed SFCTransition Edges]
Get source SFCStep
Add all successors to source SFCStep to "changed SFCSteps"
Find changed SFCSteps
Goto next changed SFCStep
[IsSFC and more changed SFCSteps available]
Find changed Actions (ST)
Goto next changed Action
[IsSFC and more changed Actions available]
Check if corelation between old and new CFG is feasible
Find changed JumpsToEdges
[Yes]
Goto next changed JumpsToEdge
[No]
Get source BasicBlock
Add first basic block to "changed BasicBlocks"
Add all successors to source BasicBlock to "changed BasicBlocks"
[IsST or (IsSFC and no more changed SFCSteps available)]
[No more changed BasicBlocks from JumpsToEdges]
Find changed BasicBlocks
Add to "changed BasicBlocks"

Figure 30: Process of the fine-grained change identification resulting in a set of changed basic blocks (white: SFC, gray: ST)

Larger changes to the control flow, i.e. changes that add and remove basic blocks or transitions, cannot be safely identified by the comparison algorithm. This could be the focus of future work. In case the comparison algorithm fails to identify small changes, the first basic blocks is marked as changed, which leads to all test cases relating to this POU or SFC step to be prioritized. This fine-grained change identification process is depicted in detail in Figure 30.

5.4.3 Change Impact Analysis

A change to the control software can not only have a direct influence on the software's output signals and their timing but also on other elements in the code. If for example, a local variable is assigned a different value in a modified basic block, this can have an influence on the program's progression through the control flow. Regarding this indirect influence on the control software's behavior, a change impact analysis algorithm was developed to analyze the cross connections within the program based on the previously identified changes. For this, modified basic blocks, transitions between basic blocks and SFC steps and global variable assignment are analyzed in detail for three different types of possible influences: influence by changed assignments, calls or decisions (see Figure 31). For this, basic blocks are subdivided further into statements, for a differentiated analysis. Directly changed or indirectly changed items will be both called "modified" from this point for better readability.

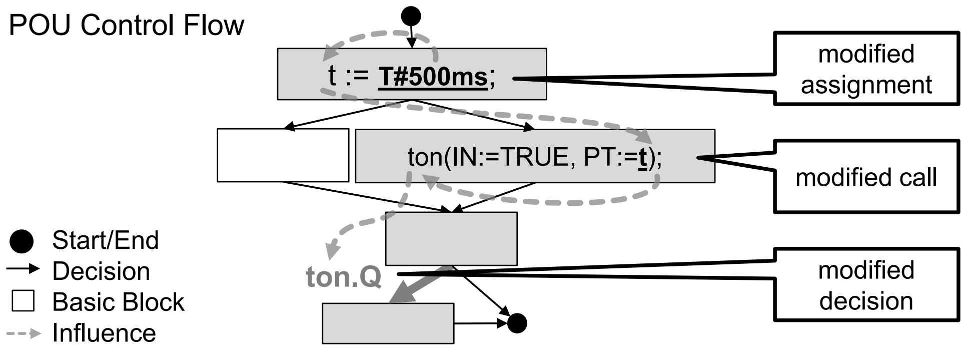


Figure 31: Change impact due to 1. modified assignments, 2. modified calls and 3. modified decisions; influenced basic blocks are light gray

```
a := b; //b influences a
```

Influence by modified assignments: A change of an assignment of a variable (write access) that is used in a different part of the code (read access) can have an influence on the progression through the code (control flow) or the output behavior of the control program. Thus, the newly assigned variable is marked as modified.

```
a(input:=b); //b influences a
```

Influence by modified calls: A change in passed values in a POU call will likely have an influence on its behavior. Therefore, the called POU is marked as modified in case of modified passed values in one or more arguments.

```
IF b THEN //b influences control flow
```

Influence on the control flow by modified decisions (edges between basic blocks): If a condition of a decision was modified by another change (see previous change types), the progression of the program through the code (control flow) is likely to exhibit differing behavior. As decisions are not instrumented directly, the previous and subsequent basic block of the decision are marked as modified.

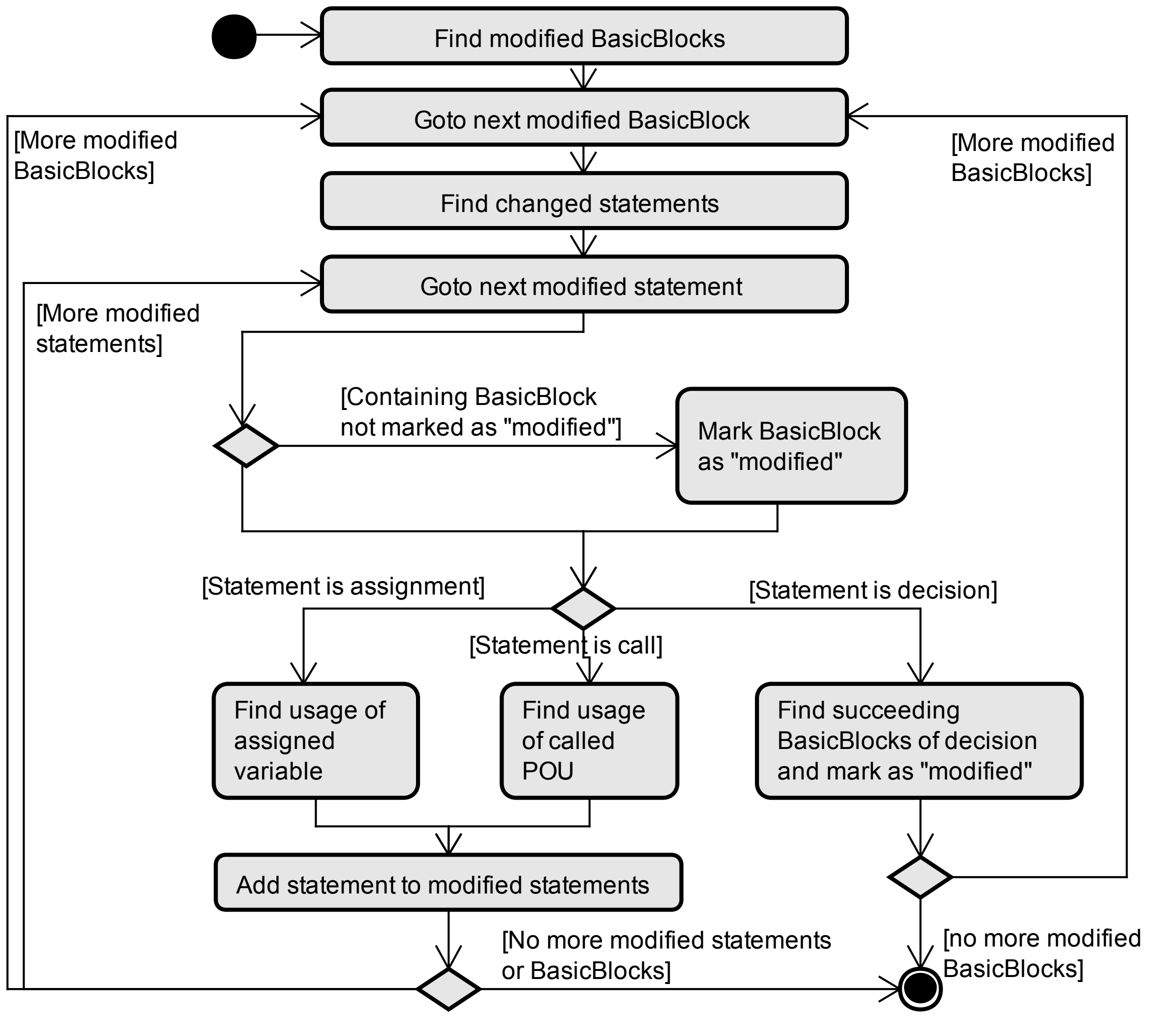

Find modified BasicBlocks
Goto next modified BasicBlock
[More modified BasicBlocks]
[More modified BasicBlocks]
Find changed statements
Goto next modified statement
[More modified statements]
[Containing BasicBlock not marked as "modified"]
Mark BasicBlock as "modified"
[Statement is assignment]
[Statement is decision]
[Statement is call]
Find usage of assigned variable
Find usage of called POU
Find succeeding BasicBlocks of decision and mark as "modified"
Add statement to modified statements
[No more modified statements or BasicBlocks]
[no more modified BasicBlocks]

Figure 32: Process of the Change Impact Analysis: BasicBlocks are analyzed for changed statements for whose impact on other statements and thus BasicBlocks is analyzed

Using this change impact algorithm (see Figure 32), the influence of a change can be analyzed and is stored in the dependency model: basic blocks are marked as modified, which is treated equivalently to a direct change of a basic block. The change can affect the control flow within a POU, but can also reach code outside of the initially investigated POU. While the influence could technically span throughout the whole program quickly, this problem was not encountered in any of the conducted preliminary and evaluation experiments. Still, optimizations of this algorithm can be the focus of future research.

5.4.4 Basic Prioritization

The goal of a prioritization is to quickly unveil regressions in the system. As test cases that only traverse code that has not possibly been influenced by modifications is more likely to yield the same result as the previously successful test execution, these test cases are given a lower priority. Conversely, test cases that traverse modifications might fail, unveiling a regression in the system. For this reason, the set of system tests is grouped in test cases that are modification traversing (high priority test cases) and those that are not (low priority test cases). The grouping is performed in four steps: 1) all possibly influenced basic blocks are identified in the dependency model, which stores information about changes and possible influence (modifications) after the change identification and change impact analysis. 2) An iteration through each execution trace is performed and "visited" (traversed) basic blocks that were possibly influenced are identified. 3) If an execution trace shows that a test case has visited a modified basic block, it is added to the group of high priority test cases. 4) After iteration through all execution traces, all test cases that have not been added to the group of high priority test cases are added to the group of low priority test cases.

A generic example for this prioritization is given in Figure 33: Given five test cases (*a* to *e*), each test case is investigated for its relation to changes. If a test case relates to a change, it is give a higher priority.

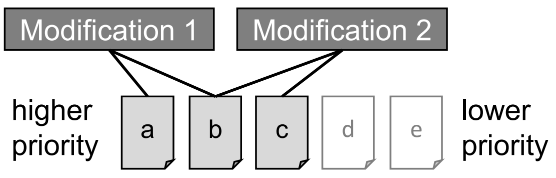


Figure 33: Basic prioritization: Test case a, b, and c are prioritized higher as they traversed the parts of the code that have now been changed (change 1 and 2).

5.4.5 Refined Prioritization

In practice, systems are tested with a set of many test cases. Thus, many test cases might be identified as high priority test cases. For a more efficient prioritization, a refined prioritization was developed. The refined prioritization aims at two types of changes: changes that cause sporadic problems and changes that have a wide influence on the system. A prioritization strategy aiming at intensely testing modifications for the former and another aiming at testing all modifications as quickly as possible for the latter. Through these refined prioritization techniques, the advantage of prioritizing test cases through the quicker unveiling of changes was expected to improve. The extension of used data for prioritization is shown in Figure 34: In contrast to the basic prioritization, not only the connection of test cases to changes, but also their intensity of traversal, as well as the total runtime and the first traversal of each change is regarded.

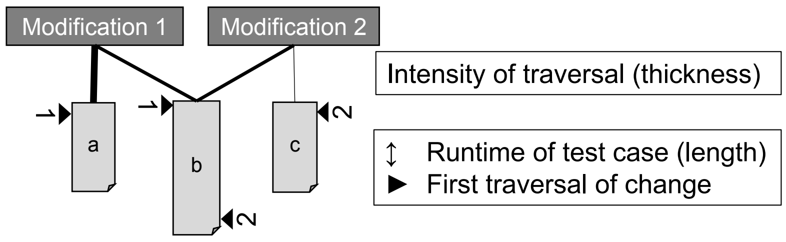

Modification 1
Modification 2
Intensity of traversal (thickness)
a
b
c
1
1
2
2
↕ Runtime of test case (length)
► First traversal of change

Figure 34: Refined prioritization: For a more detailed prioritization, the intensity of traversal and first traversal of each change by each test case is regarded in addition to the total runtime of the tests.

5.4.5.1 Prioritizing Test Cases That Intensely Traverse Modifications

Some changes of the system cause regressions that might not become apparent in the first traversal of modified code. This more detailed prioritization, therefore, gives test cases a higher priority that traverse modifications as much as possible in the least amount of time. It is aimed at finding sporadic faults caused by regressions of the system.

The information that is gathered for this prioritization is the number of traversals of possibly influenced parts of the code and the total execution time previously required by the test case. For each test case, a prioritization number p_{it} is calculated. This number represents the times per second the test case previously traversed now modified basic blocks. In the following, the number is calculated for a test case that visited modified basic blocks *0 to n* and within a total runtime of *t*:

$$p_{it} = \frac{\sum_{i=0}^{n} visits}{t}$$

Thus, p_{it} is the sum of all visits of all modified basic blocks divided by the seconds previously needed to execute the test case.

After calculation of the prioritization number p_{it}, a refined prioritization of all test cases can be performed. So far, the algorithm does not differentiate between different modifications. Thus, if a change has an impact on many parts of the code, this prioritization algorithm might not check the desired part of the functionality. This prioritization method is therefore aimed at changes that have little influence on the control code, but might fail due to sporadic faults, in particular in connection with the controlled hardware.

5.4.5.2 Prioritizing Test Cases That Traverse as Many Modifications as Fast as Possible

Some changes influence many different parts of the code that might all be related to regressions of the system. For this reason, this refined prioritization method tries to prioritize a set of test cases that executes all or as many modifications of the code as fast as possible. It is aimed at quickly finding faults that become apparent in the first traversal of the modified part of the system.

If a revision of the control software includes many modifications (changes to the control software and resulting possibly influenced parts of the code), there might not be a single test case traversing all of these modifications. Depending on the quality of the test set, all modifications might not even be traversed when executing all test cases. Thus, instead of prioritizing single test cases, modification traversing test combinations (MTTC) are arranged. These combinations are sequences of test cases to be executed to cover all or as many modifications as possible. For this, different MTTCs are arranged and rated for their total time until all modifications are traversed. The MTTCs are inferred using the following process:

1. For each modification traversing test case, a new MTTC is instantiated. If all modifications are traversed, the MTTC is completed and the process is repeated for the next test case.

2. If the MTTC does not cover all modifications yet, the set of remaining untraversed modifications and related test cases are collected. For each item of this new set of test cases, a new MTTC is instantiated and filled with the test cases that were chosen so far.

3. Step 2 is repeated until all modifications are covered or no test cases are left that cover remaining modifications.

4. For each MTTC, the total time needed to traverse all modifications is calculated, which is done by adding up all total execution times of the individual test cases, except for the last test case for which only the first traversal for the remaining untraversed modification is added.

5. Each MTTC instance is then rated by the needed total time to traverse all modifications. The MTTCs are prioritized in an order in which the fastest MTTCs is prioritized the highest. As test cases might be part of multiple MTTCs, only the test cases included in the shortest MTTC are prioritized the highest and in the respective order.

6. The process is repeated for the remaining modification traversing test cases.

A simple example for this prioritization is given in Figure 35: Given three test cases a, b, and c, and two modifications 1 and 2, different MTTCs are combined. In this example, MTTC3 will be the combination of test cases to traverse both changes the quickest.

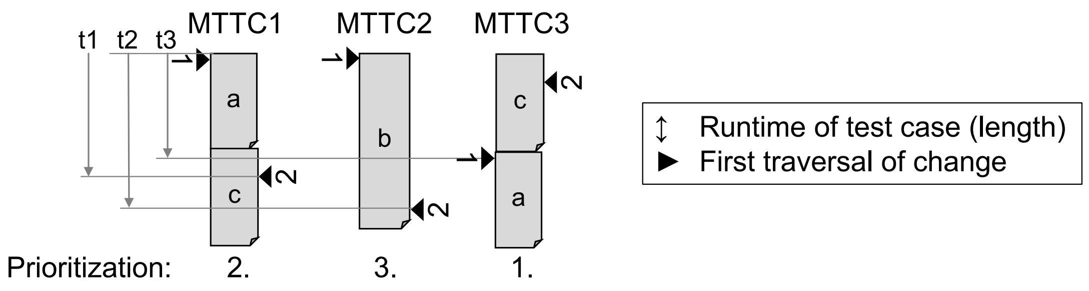


Figure 35: Comparing modification traversing test combinations (MTTC): MTTC1 will traverse both changes earlier (after t1) than MTTC2 (after t2), yet MTTC3 is the quickest (t3)

Through this prioritization, a combination of test cases that previously traversed all possibly influenced parts of the control software is executed first, followed by further combinations that try covering as many modifications as possible using the remaining modification traversing test cases. Non-modification traversing test cases follow as low priority test cases with no particular order. Using this method, a quick test of all possibly influenced parts of the code is achieved, enabling to unveil possibly introduced unwanted behavior in different functions of the code.

6 Implementation of the Approach for Efficient System Testing in Production Automation

In order to be able to prove the applicability of the presented concept within the production automation domain, a prototypical tool for defining and executing tests, measuring and assessing test coverage and regression test prioritization was implemented. The tool was implemented as a plug-in for the widely used CODESYS V3.5 Integrated Development Environment (IDE) for aPS programmed in the IEC 61131-3 standard (3S - Smart Software Solutions GmbH, 2016a). Through the close integration with the IDE, information about the source code, its instrumentation and the automation of the test execution and coverage measurement was achieved. Using the capabilities of the IDE, dependencies and the abstract syntax tree could be easily extracted from the *compile context*, i.e. the object model used as an input for the compiler. The tracepoint database generated during instrumentation is saved as an XML-file for later use during coverage assessment and test prioritization.

The test definition and test project generation was developed as an extension of the CODESYS Test Manager (3S - Smart Software Solutions GmbH, 2016c). Test cases can be directly added into the object tree, as shown in Figure 36. Historically, the object was designed as a unit test, being added as a child to a POU, i.e. testing this POU. When added to the entry point of the program, e.g. POU “PLC_PRG”, the unit test becomes a system test.

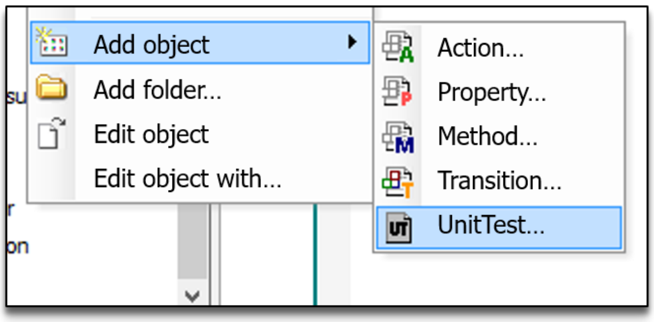
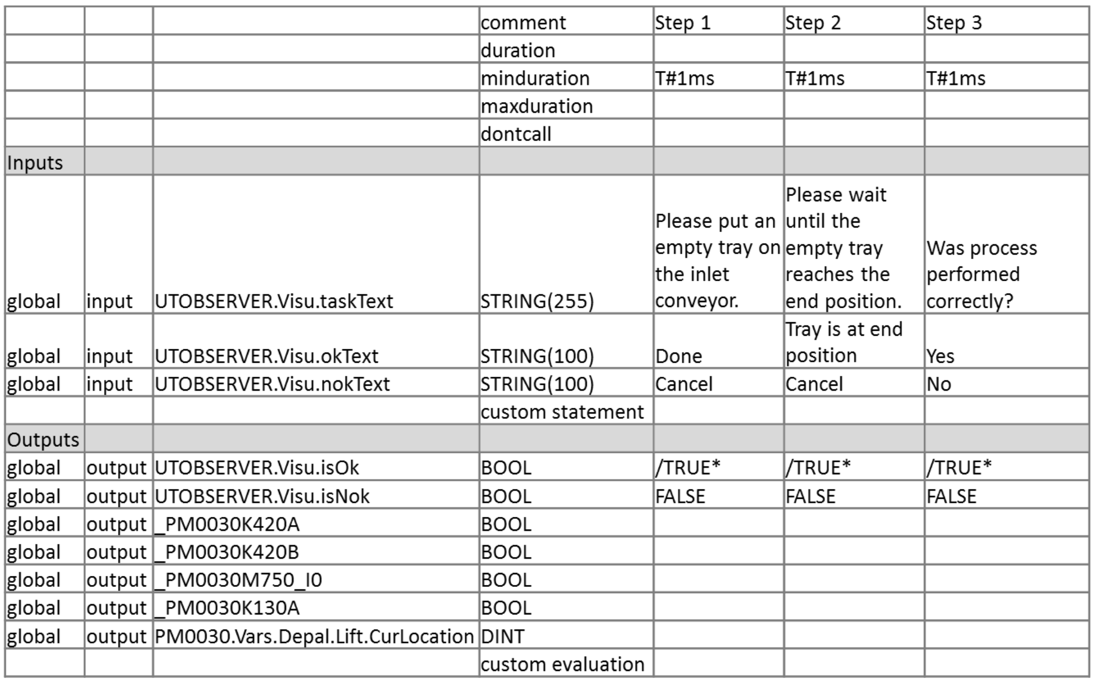


Figure 36: Test cases can be directly added to the object tree of the PLC software project (still called UnitTest for historic reasons; screenshot translated, as prototype is in German).

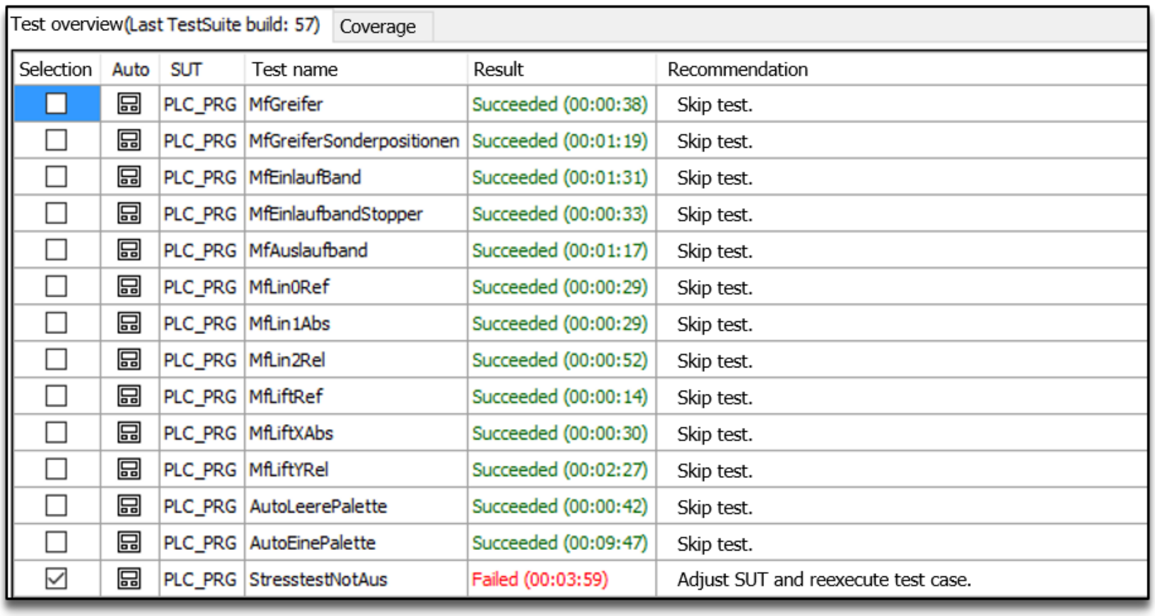
As shown in Figure 37, the definition of test cases was implemented using the comma-separated-value (CSV) format and a table calculation tool. This prototypical implementation allowed for the specification of test cases without a specialized editor or knowledge about XML (exchange format defined in cooperation with project partners).



			comment	Step 1	Step 2	Step 3
			duration			
			minduration	T#1ms	T#1ms	T#1ms
			maxduration			
			dontcall			
Inputs						
global	input	UTOBSERVER.Visu.taskText	STRING(255)	Please put an empty tray on the inlet conveyor.	Please wait until the empty tray reaches the end position.	Was process performed correctly?
global	input	UTOBSERVER.Visu.okText	STRING(100)	Done	Tray is at end position	Yes
global	input	UTOBSERVER.Visu.nokText	STRING(100)	Cancel	Cancel	No
			custom statement			
Outputs						
global	output	UTOBSERVER.Visu.isOk	BOOL	/TRUE*	/TRUE*	/TRUE*
global	output	UTOBSERVER.Visu.isNok	BOOL	FALSE	FALSE	FALSE
global	output	_PM0030K420A	BOOL			
global	output	_PM0030K420B	BOOL			
global	output	_PM0030M750_I0	BOOL			
global	output	_PM0030K130A	BOOL			
global	output	PM0030.Vars.Depal.Lift.CurLocation	DINT			
			custom evaluation			

Figure 37: Prototypical implementation of the test case definition using table calculation tools: the left rows define the interfacing variables (top: inputs to set, bottom: outputs to check). The test case is progressing from left to right.

As shown in Figure 38 test suites can be defined using available test cases by selecting (column "Selection") and manually prioritizing (using drag-and-drop) the items. In case tests have been executed before, information about result and runtime (column "Result"), as well as a recommendation whether to re-execute are given (column "Recommendation").



Test overview(Last TestSuite build: 57) | Coverage

Selection	Auto	SUT	Test name	Result	Recommendation
☐	▣	PLC_PRG	MfGreifer	Succeeded (00:00:38)	Skip test.
☐	▣	PLC_PRG	MfGreiferSonderpositionen	Succeeded (00:01:19)	Skip test.
☐	▣	PLC_PRG	MfEinlaufBand	Succeeded (00:01:31)	Skip test.
☐	▣	PLC_PRG	MfEinlaufbandStopper	Succeeded (00:00:33)	Skip test.
☐	▣	PLC_PRG	MfAuslaufband	Succeeded (00:01:17)	Skip test.
☐	▣	PLC_PRG	MfLin0Ref	Succeeded (00:00:29)	Skip test.
☐	▣	PLC_PRG	MfLin1Abs	Succeeded (00:00:29)	Skip test.
☐	▣	PLC_PRG	MfLin2Rel	Succeeded (00:00:52)	Skip test.
☐	▣	PLC_PRG	MfLiftRef	Succeeded (00:00:14)	Skip test.
☐	▣	PLC_PRG	MfLiftXAbs	Succeeded (00:00:30)	Skip test.
☐	▣	PLC_PRG	MfLiftYRel	Succeeded (00:02:27)	Skip test.
☐	▣	PLC_PRG	AutoLeerePalette	Succeeded (00:00:42)	Skip test.
☐	▣	PLC_PRG	AutoEinePalette	Succeeded (00:09:47)	Skip test.
☑	▣	PLC_PRG	StresstestNotAus	Failed (00:03:59)	Adjust SUT and reexecute test case.

Figure 38: Implementation of the test suite editor (screenshot translated, as prototype is in German)

Subsequent to defining the test suite, the testing process can be initiated (see Figure 39). The prototypical plug-in will instrument the code for tracing, create test tables (test objects compatible with the CODESYS TEST MANAGER) and a test script, which includes all instructions for the CODESYS TEST MANAGER. Subsequently, the test script is executed, invoking the test block creation (IECUnitTest-action in the CODESYS TEST MANAGER) and an upload and starting of the project on the PLC. After the semi-automatic system test is finished, execution traces, saved as text files on the PLC, will be automatically transferred to the development system. The traces and a test report from the CODESYS TEST MANAGER is imported (see 5.3.4) and a coverage calculation is performed. Apart from the test case definition, selection and the test execution itself, all processes are fully automatic.

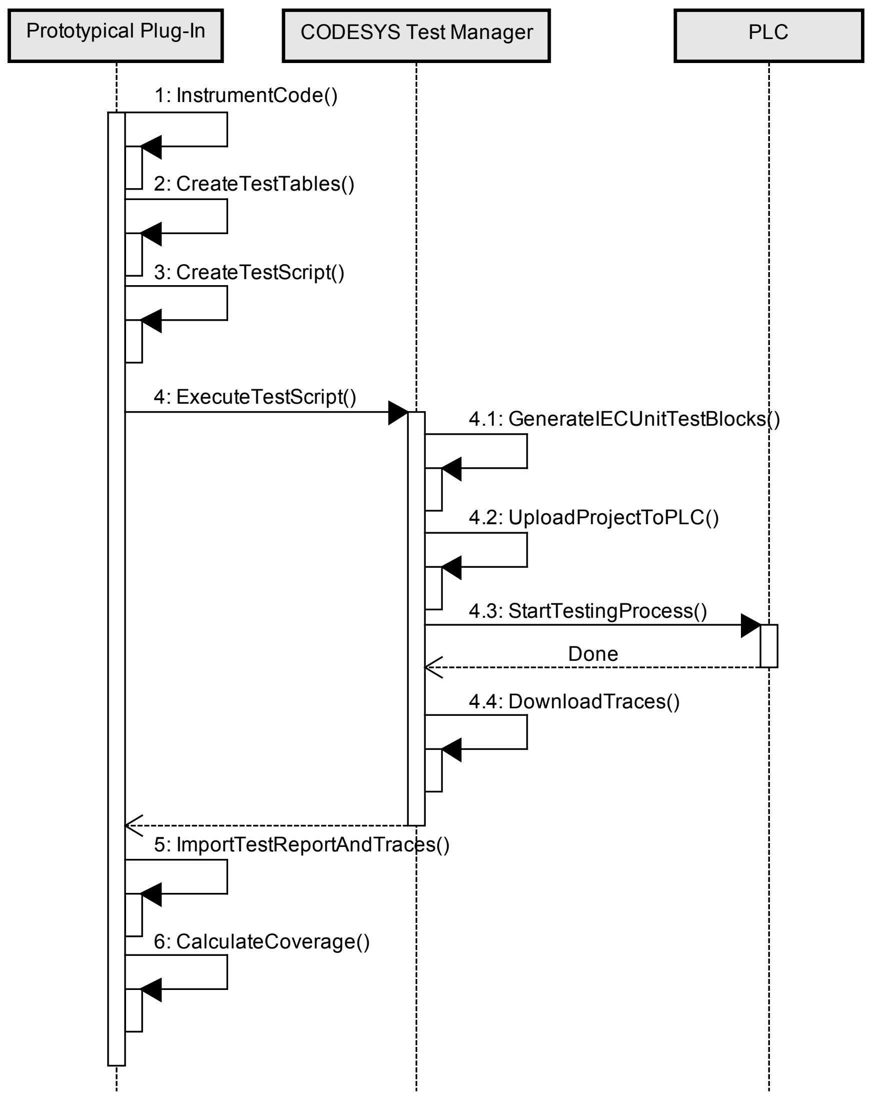
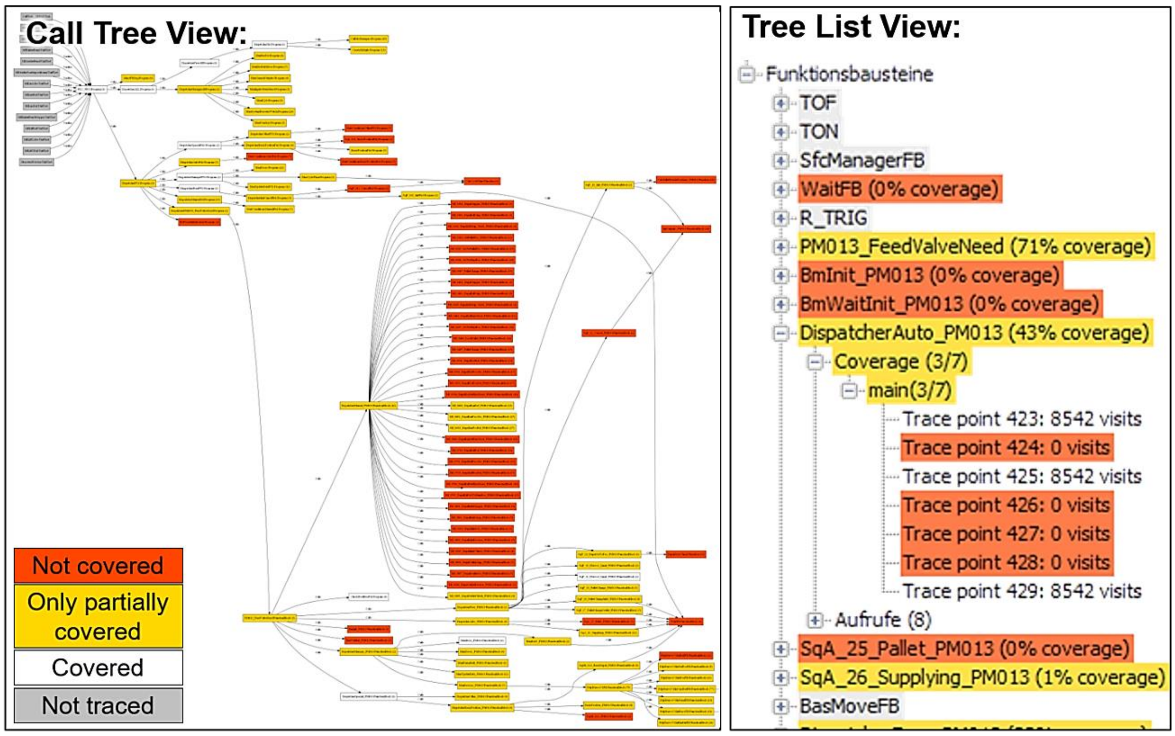


Figure 39: The implementation of the automated generation of the test project and retrieval of test data after testing

After test execution and download of execution traces, the developed plug-in automatically loads and analyzes the execution trace files coverage information and displays and browses this information visually (see Figure 40). For this, two views were implemented: a call tree view (Figure 40, left) and a tree list view (Figure 40, right, including block coverage values for discussion purposes). Both views allow for an exploration of covered, partially covered and uncovered code entities. For example, the tree view is controllable by clicking on POUs, which will display all contained objects, e.g. the ST-CFG (basic blocks and decisions) including their coverage. Similarly, the tree list view will allow unfolding individual components to gain a more detailed understanding about the coverage of the code and possibly critical uncovered code elements.



```
⊟ Funktionsbausteine
   ⊞ TOF
   ⊞ TON
   ⊞ SfcManagerFB
   ⊞ WaitFB (0% coverage)
   ⊞ R_TRIG
   ⊞ PM013_FeedValveNeed (71% coverage)
   ⊞ BmInit_PM013 (0% coverage)
   ⊞ BmWaitInit_PM013 (0% coverage)
   ⊟ DispatcherAuto_PM013 (43% coverage)
      ⊟ Coverage (3/7)
         ⊟ main(3/7)
               Trace point 423: 8542 visits
               Trace point 424: 0 visits
               Trace point 425: 8542 visits
               Trace point 426: 0 visits
               Trace point 427: 0 visits
               Trace point 428: 0 visits
               Trace point 429: 8542 visits
      ⊞ Aufrufe (8)
   ⊞ SqA_25_Pallet_PM013 (0% coverage)
   ⊞ SqA_26_Supplying_PM013 (1% coverage)
   ⊞ BasMoveFB
```

Figure 40: Prototypical coverage browser

7 Qualitative Evaluation of the Approach

In order to investigate the accordance of the developed approach to the initially specified requirements, an evaluation was performed using an industrial case study (see Chapter 7.1). For this, several experiments were designed in order to acquire meaningful data to support the notion that the approach did indeed fulfill the requirements (see Chapter 1). For some experiments, an experienced industrial expert supported the design of realistic test plans and change scenarios. The experiments were performed by the author, during which different measurements, e.g. regarding the runtime overhead, were performed. The measurements and findings were processed and used in an expert evaluation (see Chapter 7.3). In this expert evaluation, the findings were presented and discussed with a group of six experienced experts from an internationally successful company in the field of aPS engineering. In addition, a questionnaire was filled out by all experts. The results of this workshop discussion and the questionnaire are then discussed in relation to the initially imposed requirements (see Chapter 1).

The case study, the representative group of participants and the measured data were intentionally chosen as proposed by (Runeson *et al.*, 2012) to allow for an evaluation of the initial requirements.

7.1 Description of the Case Study

The system used for experimentation had been part of a real industrial factory automation system for depalletizing trays and passing the individual items on to the next station (see Figure 41). Trays with parts are fed into the machine using conveyor belts. A lift system is used for locking the tray in position for picking, and subsequent transport to a conveyor system transporting the empty tray out of the system. A 3-axis pick and place unit (PPU) is used to pick up individual pieces off the locked tray and place them into the next machine, representing the next process step (this process step was not regarded in this work). A schematic view of the machine is depicted in Figure 42.

For interaction with the hardware (including 3 drives), 69 input variables and 26 output variables are available (mostly Booleans). The control program was written by the company (not by any of the participants) and contained 119 program organization units adding up to about 15500 lines of code. The program includes two tasks, one for the control code, running at a fixed PLC scan cycle time of 10ms, and one task for updating the visualization, running at 100ms. The used programming languages were IEC 61131-3 Structured Text (ST) and Sequential Function Chart (SFC). Thus, the size and complexity of the program represent a realistic application example. The program was initially written in the integrated development environment CODESYS V2 and ported to CODESYS V3.5 for this evaluation as the plug-in was developed for the newer version.

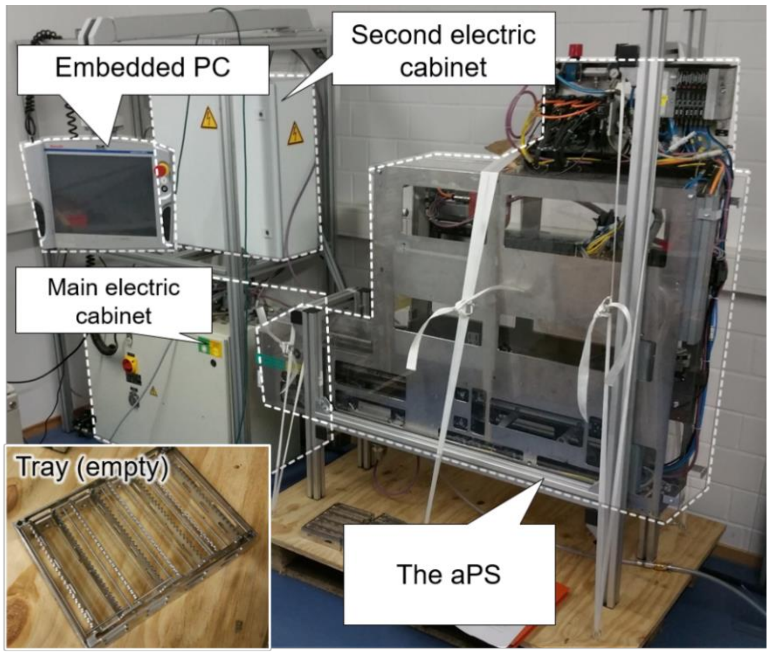


Figure 41: The aPS used in the case study including electric cabinets and embedded PC depalletizes trays filled with needles (bottom left)

The system's control hardware is a BOSCH REXROTH INDRACONTROL VPP 21 embedded PC with a PENTIUM III 701MHz processor and 504MB of RAM. The embedded PC runs a CODESYS CONTROL RTE V3.5.5.20 real-time capable runtime. An Ethernet connection was used to connect the embedded PC to a development PC running the plug-in and uploading the test project to the embedded PC for real-time capable execution.

The development system used for generating the instrumented code, the test project and the coverage assessment was a consumer laptop with an INTEL® CORE™ I7 5600U CPU at 2.6GHz, 8GB RAM and running Microsoft Windows 10 64-bit. CODESYS V3.5 SP8 Patch 1 and CODESYS TEST MANAGER Version 4.0.1.0. were used, including the developed plug-in.

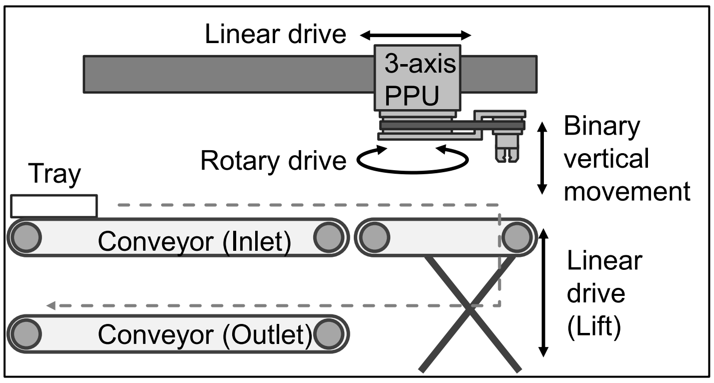


Figure 42: Schematic view of the system under test (SUT)

As described above, the hardware and technical process in the system is controlled by an embedded PC. This PC integrates an HMI, which is shown in Figure 43, and consists of a touchscreen display and several buttons. In regular operation, the HMI shows information about the machine and permits changing the operating mode (top right) between manual mode, special functions for calibration and automatic operation. In the manual operating mode, the display is used to choose appropriate manual functions, such as a function for opening and closing the gripper or turning on the inlet conveyor. After selecting a manual function, the buttons on the side can be used to perform the manual function. The HMI was ported from the originally used program (CODESYS V2) and extended by test visualization on the bottom right for the semi-automatic system testing approach.

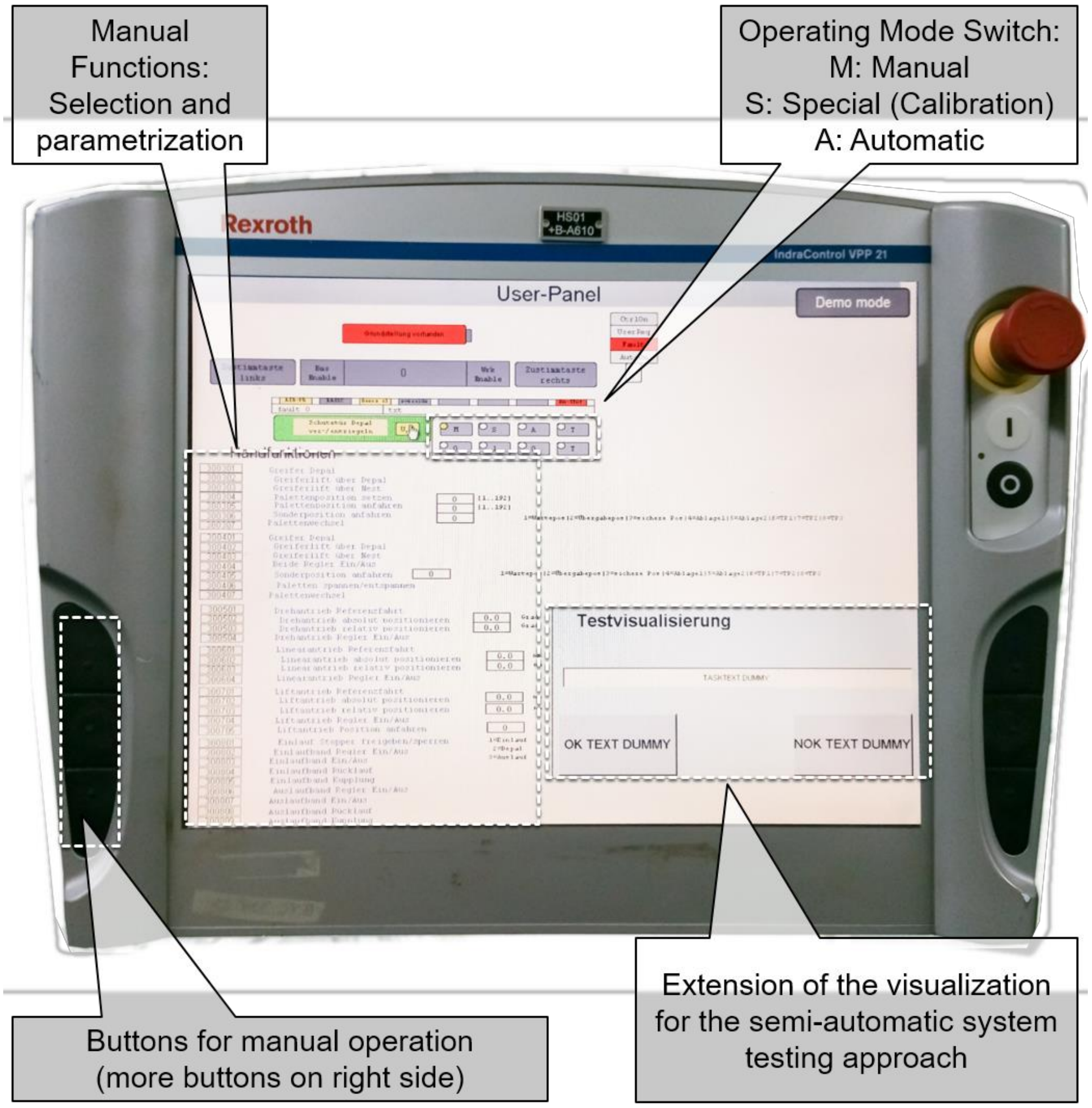

Manual Functions: Selection and parametrization
Operating Mode Switch: M: Manual S: Special (Calibration) A: Automatic
Rexroth
HS01 +B-A610
IndraControl VPP 21
User-Panel
Demo mode
Testvisualisierung
OK TEXT DUMMY
NOK TEXT DUMMY
Buttons for manual operation (more buttons on right side)
Extension of the visualization for the semi-automatic system testing approach

Figure 43: Description of the HMI used in the case study

7.2 Experiments

All experiments were designed to acquire insights about the fulfillment of one or more requirements. Both requirements and experiments focus on the technical side of the approach, excluding an investigation of the usability of the utilized prototypical software tool. An overview over the connection of requirements and performed experiments is shown in Table 9.

Table 9: Evaluation of requirements: An overview over the connection of requirements and performed experiments

	Requirement	**How fulfillment of the requirement was evaluated**
Applicability	R_{IEC} – Support of IEC 61131-3	All experiments: Application of approach on industrial aPS program
	R_{SW} – Support of industrial code complexity	All experiments: Application of approach on industrial aPS program
	R_{Int} – Support of interaction with the integrated system	Experiment I: Application of approach using industrial test plan and industrial case study
	R_{HWB} – Inclusion of valid hardware behavior	Experiment I: Application of approach using industrial test plan and industrial case study
	R_{Sim} – Independence from behavior simulations	All experiments: Application of approach on industrial case study without additional models
	R_{RT} – Insignificant influence on real-time properties	Experiment IV: Measurements regarding overhead and discussion with experts
	R_{Mem} – Insignificant influence on memory size	Experiment IV: Measurements regarding overhead and discussion with experts
Improvements	R_{Rep} – Improved Repeatability	Experiment I: Repetition and comparison of several test runs
	R_{Doc} – Improved Documentation	Experiment I: Application of approach on industrial case study and demonstration of documentation
	R_{TA} – Support for Test Adequacy Assessment	Experiment II: Application of approach on industrial case study and discussion of results with experts
	R_{Reg} – Support for Regression Testing	Experiment III: Application of approach on industrial case study and discussion of results with experts

All experiments and measurements were performed by the author and will be described in more detail in the following sections.

7.2.1 Experiment I: Guided System Testing

A test suite was created for the system presented in the previous section based on a test plan provided by one of the industry partners. The test suite consists of 15 system test cases with a total runtime of about 25 minutes, directly testing the machine (R_{HWB}) in different operating modes (manual and automatic) and in the case of an operating mode switch during automatic operation. All test cases include manual operations by the operator (R_{Int}), such as putting a filled tray into the machine or acknowledging that the gripper is indeed closed. The test suite was created with the notion that most important functions in the machine were tested. The set of test cases was approved by an industrial expert from the company. It was reused for the other experiments.

Table 10: List of test cases as compiled for the test suite used in the case study

#	TC-Name	Approx. runtime [m:ss]	Description of performed actions
1	MfGreifer	0:38	Manual function (Mf): Open and close the gripper
2	MfGreiferSonderpositionen	1:19	Mf: Move gripper to special positions
3	MfEinlaufBand	1:31	Mf: Switch inlet conveyor on and off
4	MfEinlaufbandStopper	0:33	Mf.: Switch stopper at inlet to stop and back
5	MfAuslaufband	1:17	Mf: Switch outlet conveyor on and off
6	MfLin0Ref	0:29	Mf: Perform reference function with linear gripper drive
7	MfLin1Abs	0:29	Mf: Perform absolute movements with linear gripper drive
8	MfLin2Rel	0:52	Mf: Perform relative movements with linear gripper drive
9	MfLiftRef	0:14	Mf: Perform reference function with lift drive
10	MfLiftXAbs	0:30	Mf: Perform absolute movements with lift drive
11	MfLiftYRel	2:27	Mf: Perform relative movements with lift drive
12	AutoLeerePalette	0:42	Perform automatic operation with empty tray
13	AutoHalbvollePalette	1:33	Perform automatic operation with partially filled tray
14	AutoEinePalette	9:47	Perform automatic operation with full tray
15	StresstestOpMode	4:10	Change operating mode during automatic operation

Repeatability (R_{Rep}) was improved by the detailed definition of individual test steps. Furthermore, during the repetition of test cases (experiment IV), all tests all had the same result. Documentation of the test cases (R_{Doc}) is improved by recording detailed value sequences during test execution and the possibility to document coverage or additional variable values of choice. The recorded values are stored as CSV-Files and can easily be reused for documentation purposes.

7.2.2 Experiment II: Coverage Investigation

The feasibility of the coverage assessment approach was investigated by tracing and visualizing the coverage of the test suite from experiment I (see Table 10). As coverage was never previously calculated or displayed, this was an interesting property of the experiment. As expected, the test suite did cover most of the code but did not cover every detail although the test suite was designed according to the notion that most important behavior in the machine was included. Many manual functions were not covered, as no test was designed to specifically allow this, which was decided due to the similarity to the other test cases for manual functions.

In real situations, these tests would have to have been specified. Some function blocks representing initialization functions were not covered as these were executed only once at program startup and thus not recorded during the actual test case execution. Some step chains were not covered as these represent behavior in case of an emergency shut down. Most POUs regarding the behavior of the machine that were addressed by test cases were partially covered. In all cases, specific behavior of the system was not included in the test case, mostly functionality related to fault detection. As an example, the behavior of the system in case of cycle time overrun was not investigated as no such situation occurred during test execution (see Figure 44). Another interesting finding was that unneeded code was detected: Due to time restrictions, several step chains were copied and modified resulting in complete branches of legacy SFC chains not being executed (see Figure 45).

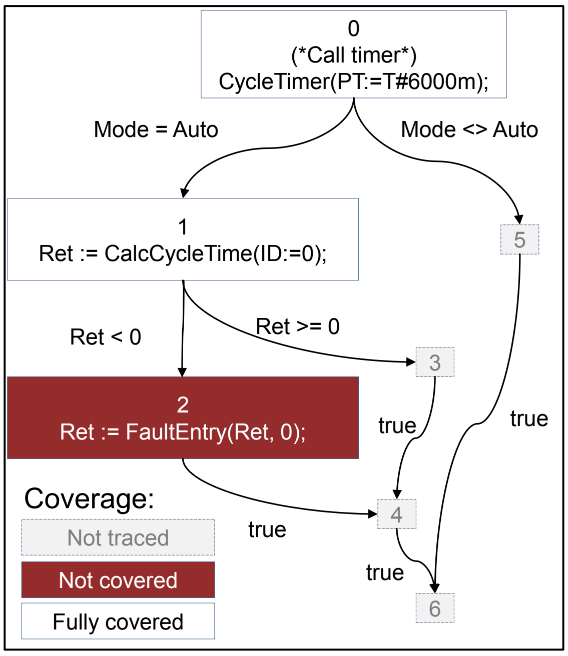


Figure 44: Coverage assessment unveils untested fault handling routine: The code branch for documenting faulty cycle timing was never executed during the tests

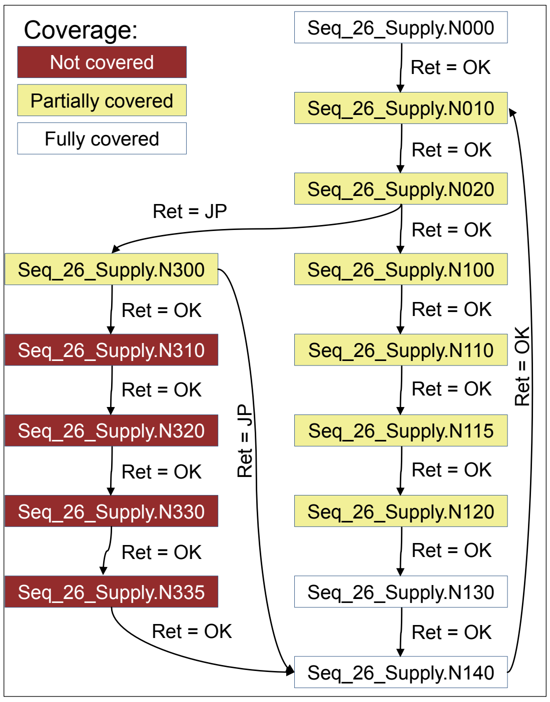
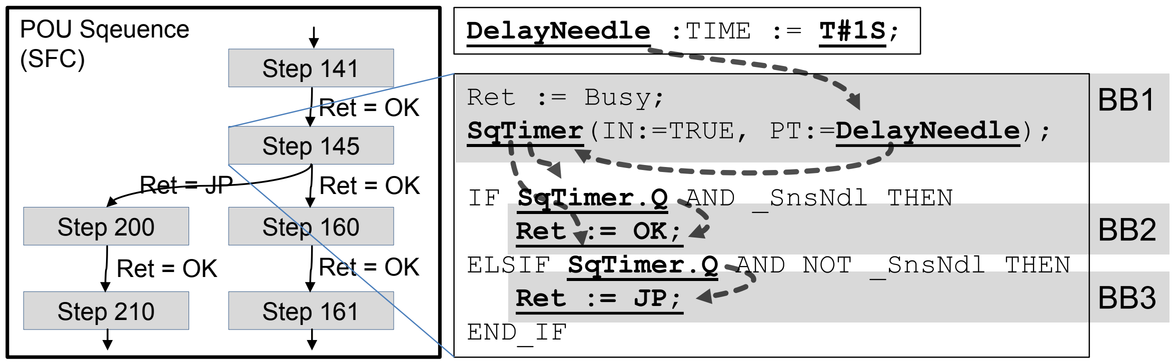


Figure 45: Coverage assessment uncovers unneeded legacy code: A complete branch of the SFC code was never used and turned out to be obsolete upon closer inspection

7.2.3 Experiment III: Regression Testing

In the third experiment, a change scenario on a previously tested system was conducted to gain knowledge about the properties of the approach regarding prioritization. In this scenario, the timing of the gripper of the pick and place unit was adjusted to allow more consistent results regarding the identification of picked up workpieces (needles). In sporadic cases, the gripper would not recognize a gripped needle even though it was holding on to one. The identification of gripped workpieces is achieved by gripping, waiting and then checking a vacuum sensor (Figure 46, "_SnsNdl") that yields a different result in case a workpiece is present. The change relates to the waiting time, which was prolonged to allow for a more consistent buildup of vacuum and thus a more consistent identification of gripped needles. For this, the assignment of a global variable "DelayNeedle" is adjusted which is referring to the waiting time before checking the vacuum sensor. The modification is regarding a part of the code that is not directly executed by test cases (the test cases do not run through the global variable assignments). Yet, this modified variable value is used in parts of the code, which are executed by test cases. Thus, a possible influence of the assignment on different parts of the code exists.

A schematic view of the change scenario is depicted in Figure 46. By changing the assignment of the global variable "DelayNeedle", several influenced parts of the program can be identified. The changed assignment renders the variable "DelayNeedle" modified. As it is used as an input for a call of the timer-POU "SqTimer", the called POU is also possibly affected by the change. Thus, the basic block (Figure 46, "BB1") containing the timer is added to the set of modified basic blocks. The timer is used in two decisions (needle detected or not), possibly changing the progression of the program through the code. As decisions are not directly instrumented, the previous and subsequent basic blocks are added to the modified basic blocks. In this case, the previous basic block has already been added to this set (Figure 46, "BB1"), whereas the subsequent basic blocks (Figure 46, "BB2" and "BB3") are newly added.



```
DelayNeedle :TIME := T#1S;
```

```
Ret := Busy;
SqTimer(IN:=TRUE, PT:=DelayNeedle);

IF SqTimer.Q AND _SnsNdl THEN
    Ret := OK;
ELSIF SqTimer.Q AND NOT _SnsNdl THEN
    Ret := JP;
END_IF
```

Figure 46: Experiment III: Influence of a software change regarding a global variable assignment, possibly influencing the control flow through several Basic Blocks (BB).

This information is used to relate the modified parts of the code to the timing information acquired during the previous execution of the test cases, which is depicted in Table 11. It becomes apparent that not all test cases traverse all modifications: the manual function tests (1–11), as well as the test relating to the change in the operating mode (15), did previously not traverse the now modified parts of the code. In contrast to this, the three test cases "12. Empty tray", "13. Partially filled tray" and "14. Full tray" all traverse some or all of the modifications. For each test case and each modification, additional data about the number of traversals and the timing information about the first traversal are retrieved. Furthermore, the total execution time of each test trace is retrieved.

Table 11: Timing information of all system test cases from experiment II regarding the identified change and change impact

System tests (Total execution time)	**Basic block 1 traversal**	**Basic block 2 traversal**	**Basic block 3 traversal**
1.-11. Manual functions (14s-91s)	No traversal	No traversal	No traversal
12. Empty tray (40s)	5 times, first after 23s	No traversal	5 times, first after 23s
13. Partially filled tray (1m 33s)	13 times, first after 25s	8 times, first after 25s	5 times, first after 52s
14. Full tray (9m 47s)	192 times, first after 24s	192 times, first after 24s	No traversal
15. Op-Mode-Change (3m 59s)	No traversal	No traversal	No traversal

Test cases for manual functions (1.-11.) are combined in this table as none of these test cases traverses the modifications.

Based on this information, the basic prioritization (modification traversing) and the refined prioritization methods (intense traversal and quick traversal) were performed. The basic prioritization adds test cases 12, 13 and 14 to the group of high priority test cases, whereas the rest of test cases is assigned a low priority. For the refined prioritization, the order of these three test cases is calculated.

The refined prioritization method regarding intense traversal uses the traversal count of each test case to calculate the number of times the test case interacts with a changed part of the control program. In this case, the result would be p_{it}=0.25 (modification traversals per second) for test case 12 ((5+5) traversals / 40s = 0.25 traversals per second), 0.28 for test case 13 and 0.65 for test case 14. Thus, the test case order would be 14, 13 and 12, followed by the rest of the test cases in no particular order. In test case 14, a full tray is depalletized, picking off 192 needles. Thus, the new timing is tested the most by this test case, but not in all situations: there never is an empty spot on the tray, not testing whether an empty gripper is correctly recognized by the control program.

The calculation of the prioritization for quick modification traversal returns several modification-traversing test combinations (MTTC):

MTTC 1: Test cases 12 + 13, resulting in a total time to traverse all modifications of 1m 5s (40s + 25s)

MTTC 2: Test cases 12 + 14, resulting in a total time to traverse all modifications of 1m 4s (40s + 24s)

MTTC 3: Test case 13, resulting in a total time to traverse all modifications of 52s

MTTC 4: Test cases 14 + 12, resulting in a total time to traverse all modifications of 10m 10s (9m 47s + 23s)

MTTC 5: Test cases 14 + 13, resulting in a total time to traverse all modifications of 10m 39s (9m 47s + 52s)

Prioritizing the quickest combination, MTTC 3 is chosen, containing only a single test case (13). This test case uses a partially filled pallet, thus including two scenarios for the gripper: occupied and empty spaces on the tray. Therefore, this test case would test all modifications the quickest. Another close competitor would have been the combinations of an empty tray followed by a partially filled (MTTC 1) or full tray (MTTC 2).

7.2.4 Experiment IV: Runtime and Memory Overhead

To acquire information about the runtime properties of the approach (R_{RT}), two test cases were each executed five times for six different configurations of the control software. The six configurations represent different levels of the implementation of the approach (see Table 12). They were chosen to acquire a deeper understanding of the runtime properties of the individual extensions of the approach.

Table 12: Configurations of the control program for experiment IV

Configuration	Description
1: “Original”	Uninstrumented, original control program
2: “Test only”	Instrumented program, implementing only the semi-automatic testing approach without any runtime information acquisition
3: “Traversal”	Instrumented program, implementing tracing that only allows for the unrefined prioritization of modification traversing test cases
4: “Intensity”	Instrumented program, implementing the refined prioritization using intense traversal
5: “Quickness”	Instrumented program, implementing the refined prioritization using quick traversal of all modifications
6: “All”	Instrumented program, allowing all prioritization methods

The two test cases chosen for the acquisition of runtime information were a test of a manual operation (Table 10, test case #1) and one for automatic operation (Table 10, test case #13), as these are the most different regarding the involved code. During execution of the test cases, the average and maximum execution time of the PLC scan cycle, including reading sensor and writing actuator values was measured using the task monitor of the IDE. From all measurements acquired during the execution of both test cases and all their repetitions, the average and maximum value were calculated. In addition, the required time to generate the dependency model and the required memory of the compiled program for each configuration was recorded.

The instrumentation of the project required less than one second, generating the dependency model and inserting 2261 trace function calls into the code. Each system test was executed completely without breaking real-time restrictions (10ms for the logic task), while all execution traces being written into the memory of the embedded execution hardware. The transfer from the memory to the hard drive of the embedded execution hardware was performed asynchronously after each test completion to avoid influence on real-time properties during test execution and did not exceed 10 PLC scan cycles during which the program was still executed but in an idle state.

The required execution time for each of the configurations for the given application example (see Chapter 7.1) is shown in Figure 47. Compared to the original control program, the average scan cycle time increases by 47% when using the guided testing approach alone (Figure 47, "Test only") and another 10%-14%, when recording execution data for the different prioritization algorithms (Figure 47, "Traversal" – "All"). While the average increase is quite significant, this increase has no direct influence on real-time requirements. For this, the maximum required execution time is relevant, which also increases, but to a smaller extent. The greatest increase is once again when implementing the guided testing approach (10%) and another 6%-21%, depending on the type of tracing that is performed for acquiring data for the different prioritization algorithms. Especially the quickest modification traversal prioritization (Figure 47, "Quickness") accounts for a moderate total increase of about 19% compared to the original program. This is most likely due to the slightly more complex tracing function. With the given maximum scan cycle time of 10ms for this case study, all approaches are well within the bounds of real-time requirements, none exceeding 5ms. Therefore, all prioritization methods would be applicable to the given example (Chapter 7.1).

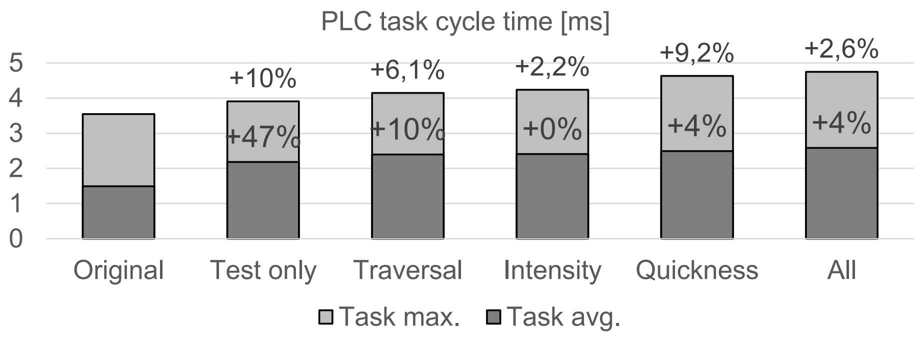


Figure 47: Comparison of the needed maximum and average PLC scan cycle time for the different prioritization approaches

Regarding required memory (R_{Mem}) on the execution hardware (see Figure 48), the additionally needed space is only increasing moderately with about 22% for each of the prioritization methods. Yet, the increase in needed global data increases significantly, resulting in an overall increase of about 148% for each of the prioritization methods. The increase is mostly due to the prototypical implementation of the guided system testing approach, accounting for the biggest increase. While a significant increase in required memory was found, the overall required memory is still low with less than 5MB.

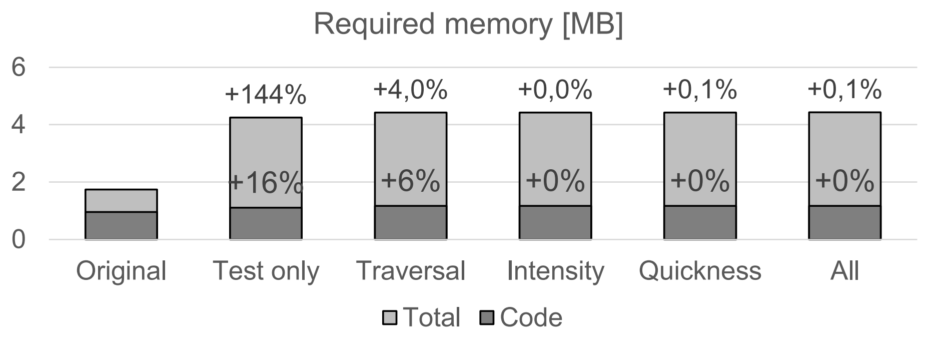


Figure 48: Comparison of the required memory for the different prioritization approaches

7.3 Expert Evaluation

The results of the measurements, as well as the approach itself, were discussed and evaluated in a group of six experts in the field of automated production systems. The group comprised employees active in the fields of commissioning, technical maintenance, aPS software engineering and group management (technical development) from the company engineering the machine used in the case study. The measurements in the previous section were presented to the group and subsequently discussed in terms of the requirements initially imposed on the approach. In addition, a questionnaire was filled out by each expert to quantify the results. Certainly, the group size does not allow for a quantitative rating of the approach although qualitative conclusions were rendered a bit more precisely.

The questions posed in the questionnaire allowed for the experts to mark their approval on a discrete scale from 1 to 7 or a continuous scale from 0 to 100. For processing, each answer was normalized to a scale from 0% to 100%. Thus, for a scale from 1-7, where 1 is “fully agree” and 7 is “fully disagree”, 1 evaluates to 100%, 2 evaluates to 83% and so on. All answers from each participant were then integrated into an average value for each question using the arithmetic mean. These values are used throughout the text. Thus, if it is stated that a claim acquired a 60% approval, this could have been caused by three experts each voting 50%, 60%, and 70% respectively ((0.5 + 0.6 + 0.7)/3).

The discussions with the experts and the numbers acquired through the questionnaire will be discussed in the following sections.

7.3.1 Evaluation of the Applicability of the Approach

As the system under test used in the case study was provided by the company as a representative example, the applicability of the support of industrial software properties was agreed upon by the experts. The IEC 61131-3 programming languages used in the case study are the only programming languages used by this company. The size of the code was also seen as representative.

Requirement R_{IEC} and R_{SW} were fulfilled.

The test cases were based on a test plan provided by the company and developed in cooperation with one of their experts, thus representing realistic test cases with realistic manipulation tasks. Therefore, the requirement for allowing for the manipulation of hardware and process during test execution was approved.

Requirements R_{Int} and R_{HWB} were fulfilled.

In addition, no simulations were required for test execution and valid hardware behavior during testing was included.

Requirement R_{Sim} was fulfilled.

While the overhead in execution time did not represent a problem in the case study, the experts agreed that this fact could be problematic for machines with very short scan cycle times, e.g. highly automated mass production machines. In these cases, scan cycle times are kept as low as possible to increase production speed. Even slight increases in scan cycle time can result in noticeable and mostly unacceptable increases in production cycles (time needed for processing one product). This is due to SFC steps being executed for at least one scan cycle, with each increase in scan cycle time adding up for each step used in the production cycle. To quantify the criticism mentioned by the experts in their questionnaire answers, it was estimated that the presented approach – in its current state (not including the optimizations presented in Chapter 8.1) – could only be applied to about 1/5 – 1/3 of the machines produced by the company based on the increase in runtime overhead.

Requirement R_{RT} was partially fulfilled.

The overhead regarding memory was not seen as critical at all. Although the percentage increase seems large, current systems used by the company never came close to running into problems regarding memory. The experts estimated the approach to be applicable to about 90% of the machines produced by the company.

Requirement R_{Mem} was fulfilled.

7.3.2 Improvement of the Current Situation in System Testing

As shown in Figure 49, the potential for improvement of the current situation in system testing in aPS using the presented approach was agreed upon by the experts.

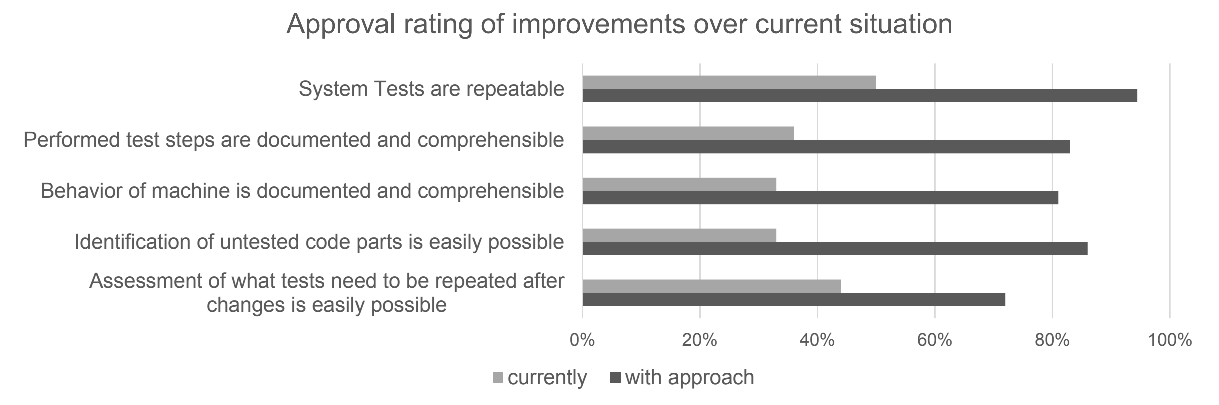


Figure 49 Approval rating of improvements over current situation as extracted from the questionnaire

In the following sections, additional details about the experts' opinions and arguments will be given.

7.3.2.1 Improvement of Repeatability and Transparency of the Testing Process

The improvement in repeatability was acknowledged by the experts in the questionnaire (94% agreement) and discussion. Still, it was noted that the resources needed to specify the test cases might not be reasonable for unique machines; reusing complex sub-modules several times might help to distribute the initial specification costs between multiple machines. According to the answers given in the questionnaire, it was estimated that the approach could be applied to half of the machines produced by this company.

To get a differentiated view on this applicability property, a rating using a flipchart and the possibility to attach stickers to a two-dimensional rating, relating to different types of machines was performed. Each participant had three stickers in different colors, representing the application of the approach on submodules of aPS (produced many times and not too complex), complex aPS (small lot size) and complex aPS (larger lot size). The results of this process are shown in Figure 50. As can be seen, the tradeoff between benefit and required resources was seen in particular for submodules of aPS and complex aPS of larger lot sizes. The initial investment for complex aPS in small lot sizes was seen as questionable.

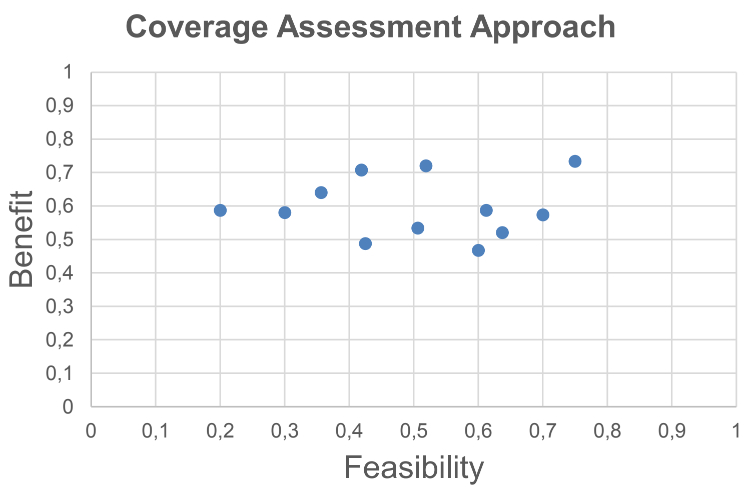


Figure 50: Qualitative evaluation of benefit vs. cost (efficiency inverted) regarding semi-automatic system testing (top right is best)

Yet, overall the requirement for improved repeatability was seen as fulfilled.

Requirement R_{Rep} was fulfilled.

Regarding the improvements in documentation, the experts also agreed on the potential benefit of the approach (see Figure 49, second and third line). The comparison of the current situation with the approach showed that the experts would expect detailed documentation of test steps and comprehensible documentation of machine behavior during testing to improve significantly (~33% agreement currently to ~80% with new approach).

Requirement R_{Doc} was fulfilled.

7.3.2.2 Support the Assessment of Test Adequacy

The experts evaluated the ability to quickly identify untested parts of the code as very beneficial. It had previously not been possible to get an overview of the executed parts of the code during testing, so test adequacy solely relied on the individual's estimation. Through easy identification of untested parts, it is possible to (quickly) assess whether additional tests are needed and adjust the test suite accordingly. This results in the experts' opinion that the approach improves the assessment of test adequacy (94% agreement).Yet, it was noted that marking the code as "tested" could be misinterpreted as "sufficiently tested".

In comparison to numbers, the visual approach was seen as more applicable. Numbers, such as "90% coverage" were seen as misleading, as it is unclear if the missing 10% are important or not. Still, numerical measurements could be interesting for controlling, e.g. as a key performance index.

The approach in its current state was seen as applicable to an agreement of 63%. Again, a flip chart was used to assess the benefit of the approach, yet this time in comparison to its feasibility (as technically, no extra cost would be required). As can be seen in Figure 51, the experts saw benefit in the approach, yet the expectation of feasibility varied. It has to be noted that due to a misunderstanding, some participants used three stickers, as in the previous example, resulting in the high number of measurement points. The previously mentioned qualitative conclusion is still possible in the author's opinion.

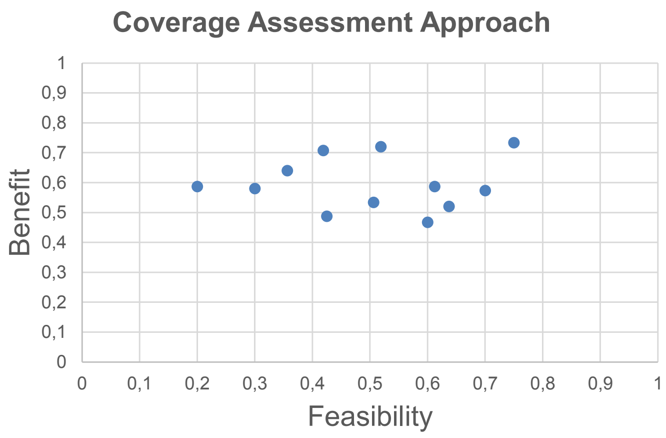


Figure 51: Qualitative evaluation of benefit vs. feasibility regarding coverage assessment (top right is best)

To conclude, an improvement by using the approach was agreed upon (94% agreement, see also Figure 49, fourth line), yet applicability in the current state of the approach was still seen as questionable (64%). The requirement for an improvement for test adequacy assessment by identifying untested behavior is seen as fulfilled.

Requirement R_{TA} was fulfilled.

As a further improvement, the experts noted that not all parts of a program are relevant for tracing. The users should thus be given the possibility to exclude parts of the code from coverage assessment.

7.3.2.3 Increase in Efficiency During the Testing Process of Changes to a Previously Tested Control Software

The requirement regarding the increase in efficiency during the testing process of changes was seen as fulfilled. It was agreed upon that the approach would represent a valuable support in preselecting and prioritizing suitable test cases, which then could be further prioritized by the testing technician. This initial prioritization of test cases would save the involved personnel significant amounts of scarcely available time. While this property of the approach could not be quantified in the presented case study, the improvement of the regression testing process was seen as improved significantly (see Figure 49, last line).

Regarding the applicability, the experts attributed 40% to this property in the current state of the prototypical implementation. This was mostly due to two discussion points. Firstly, a fully automatic prioritization was seen as critical, as it might be misused for test selection (only executing the first test cases), possibly missing important tests. Secondly, the prioritization parameters “intensity” and “quickness” did not fully convince the experts. This result can also be extracted from Figure 52, where neither the efficiency nor benefit of the refined prioritization techniques stands out.

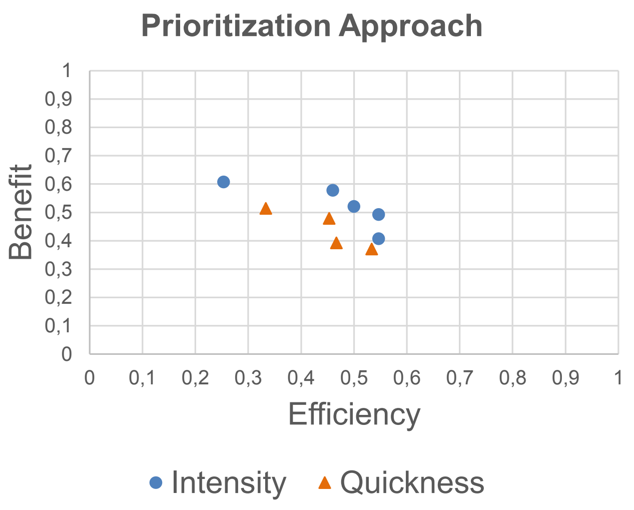


Figure 52: Qualitative evaluation of benefit vs. cost (efficiency inverted) regarding prioritization (top right is best)

Upon further inquiry, it was stated that the importance and quality of the test case are important factors for prioritization in current practice. The experts explained that different factors influence this property: relation to safety features, the influence of the tested functionality on product quality and more. Yet, some of these factors could not be extracted out of the source code alone, according to the experts’ opinions. Thus, the prioritization was seen as a helpful tool for an automatic suggestion for prioritization, yet a possibility for further changes of the order was seen as mandatory.

In total, the experts agreed that using the approach, a more structured and thus improved test prioritization can be achieved using the approach (72% agreement). Yet, more research would have to be done to achieve full industrial applicability.

Requirement R_{Reg} was partially fulfilled.

7.3.3 Overall Satisfaction of Requirements

As shown in the previous sections, most requirements were approved to be fulfilled by the group of industrial experts. The rating of the fulfillment of each requirement is summarized in Table 13.

Most requirements were satisfied for the representative case study and discussion showed that the approach in its current prototypical form (excluding the optimizations presented in

Chapter 8.1) is already applicable for a significant part of the machines produced by this company and aPS engineering.

The main points still restricting applicability are the initial cost of test specification, especially for very complex systems that are built in small numbers, and the influence on real-time. In the author's opinion, properties relating to these factors could be streamlined significantly for a commercial software product.

Table 13: Summary of the rating of the fulfillment of the requirements

	Requirement	Rating	Detailed Rating and Fulfillment of Requirements
Applicability	R_{IEC} – Support of IEC 61131-3	+	Fulfilled – An industrial example was used in the experiments (7.2), which was confirmed to be representative by the experts (7.3.1)
	R_{SW} – Support of industrial code		
	R_{Int} – Support of interaction with the integrated system	+	Fulfilled – Manual manipulation was part of the test cases in the experiments (7.2.1) and approved to be realistic by the experts (7.3.1)
	R_{HWB} – Inclusion of valid hardware behavior	+	Fulfilled – Real hardware in combination with the software was used for system testing in the experiments (7.2), approved to be an realistic example of an aPS by the experts (7.3.1)
	R_{Sim} – Independence from behavior simulations	+	Fulfilled – No simulations were used or required in the performed experiments (7.2)
	R_{RT} – Insignificant influence on real-time properties	○	Partially fulfilled – Influence of the approach was measured in experiment IV (7.2.4) and rated by the experts to result in an applicability of the approach on 1/5 - 1/3 of machines produced by the experts' company (7.3.1)
	R_{Mem} – Insignificant influence on memory size	+	Fulfilled – Influence of the approach was measured in experiment IV (7.2.4) and rated by the experts to result in an applicability on about 90% of machines (7.3.1)
Improvement	R_{Rep} – Improved Repeatability	+	Fulfilled – Investigated in experiment I (7.2.1) and rated by the experts to be a significant improvement regarding repeatability (7.3.2.1)
	R_{Doc} – Improved Documentation	+	Fulfilled – Investigated in experiment I (7.2.1) and rated by the experts to be a significant improvement regarding documentation; applicable for about 1/2 of machines (7.3.2.1)
	R_{TA} – Support for Test Adequacy Assessment	+	Fulfilled – Investigated in experiment II (7.2.2) and rated by the experts to be a significant improvement regarding identification of untested behavior and expected to increase overall test coverage; applicable for about 2/3 of the machines (7.3.2.2)
	R_{Reg} – Support for Regression Testing	○	Partially fulfilled – Investigated in experiment III (7.2.3) and rated by the experts to be a valuable support for pre-selection of test cases in case of changes; applicable for about 1/3-1/2 of machines; more research on prioritization factors needed (7.3.2.3)

8 Post-Evaluation Performance Optimization and Scalability Estimation

Summarizing the results of the case study and the expert evaluation, the approach was able to address all requirements, while in some areas room for improvements were identified to extend the approach's applicability. Especially the overhead in runtime in the current version of the approach was seen as critical for some applications in the field of industrial production automation. For this reason, an optimization of the approach regarding its performance was performed. In addition, an estimation regarding the scalability of the tracing approach was performed, to give an idea about the applicability on other projects. These findings will be presented in the following sections.

8.1 Optimization of the Runtime Overhead Generated by the Approach

Due to the identified shortcomings of the approach regarding real-time performance that were identified during the evaluation, several improvements were performed on the approach. These included 1) using inline tracing statements rather than using call-by-value function calls for each trace point, 2) using structures instead of arrays for saving traces during runtime and 3) the removal of unnecessary tracing functionality used for the debugging of the prototypical tool. An example for 1) and 2) is the substitution of a call-by-value tracing function call "`tpr(i:=25);`" (with `tpr()` accessing an array using the passed trace point ID "`i`"), with the inline call "`tp.x25 := TRUE;`".

The resulting improvements of the approach regarding PLC scan cycle overhead are depicted in Figure 53 for the most important configurations of the approach. The data was collected during the execution of twenty test case executions (ten repetitions of the two test cases used in section 7.2.4)). The increase in average scan cycle time (Figure 53, bar chart, bottom) of the old approach of up to 66% could be reduced to an increase of 12% of required execution time in comparison to the original, unchanged program. Regarding the maximum required scan cycle times (Figure 53, box chart, top), the longest observed scan cycle times of the new versions of the approach were shorter in comparison to the original program (Figure 53, box chart, top, percentage values without parentheses). Within the interquartile range of the observed values, however (Figure 53, box chart, top, percentage values in parentheses), an increase of about 15% of the required maximum execution time was observed.

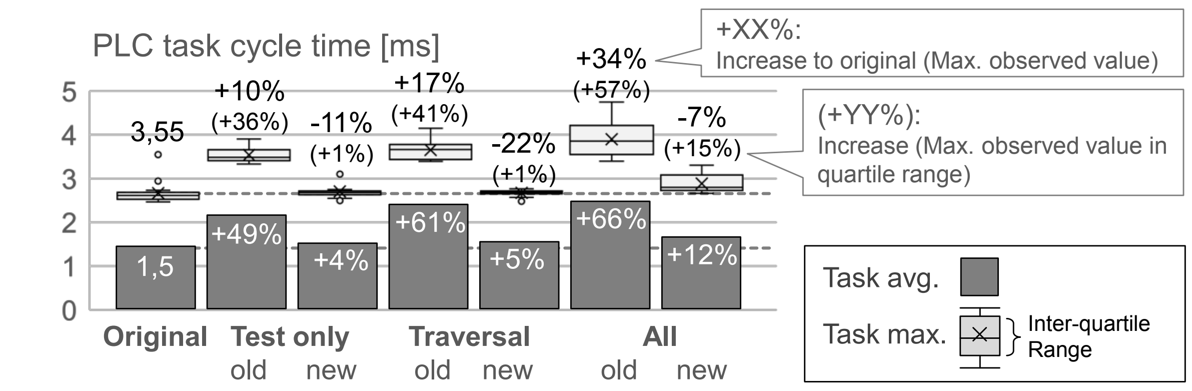


Figure 53: Required maximum and average PLC scan cycle times for the different prioritization approaches (old version used in the evaluation and new, improved version) in comparison to the original program.

Regarding the required memory, the optimized approach only required up to 22% more memory (configuration “all”) compared to the 136% required in the old version. This was mostly due to the removal of unneeded tracing functionality used for debugging the prototypical implementation itself.

With the given improvements on the overhead characteristics of the approach, a higher applicability than identified in the evaluation would be possible.

8.2 Extrapolation of the Evaluation Results regarding Scan Cycle Time Overhead

To gain a better understanding about the scalability of the approach, the data acquired during the case study performed with the optimized tracing algorithms was extrapolated for other numbers of scan cycle times and trace function invocations. With the given code, consisting of 119 POUs with an average cyclometric complexity of 9.47 and 372 actions with an average complexity of 2.13, 728 statements were executed on average during the execution of the test cases used in the evaluation. Using the average measured number of trace function invocations per scan cycle (395) and the average increase of scan cycle time attributed to the tracing approach (“traversal” 0.02ms, “all” 0.12ms) with the given setup (701MHz, scan cycle of 10ms), Table 14 was created. This data represents an estimation of the percentage of scan cycle time solely required on average for the invocation of trace function calls in regards to different scan cycle times. This was performed for the tracing configuration “traversal” and “all” (see Table 12). As expected, the percentage of required time rises with the number of trace function calls and shorter scan cycle times. Whether the remaining scan cycle time suffices for holding real time requirements (reading and writing input and output variables and executing the rest of the statements) strongly depends on the amount of statements invoked in a worst-case scenario (most computational intensive path through the code, including the trace function calls). This property is very specific for different aPS and their control software.

Table 14: Extrapolation of percentage of PLC scan cycle time required solely for execution tracing in the configurations "traversal" and "all" (see Table 12)

Trace function calls per scan cycle	**PLC scan cycle time (config. "traversal")**			**PLC scan cycle time (configuration "all")**		
	10ms	**5ms**	**1ms**	**10ms**	**5ms**	**1ms**
10	0.01%	0.01%	0.05%	0.03%	0.06%	0.31%
50	0.03%	0.05%	0.26%	0.15%	0.31%	1.53%
100	0.05%	0.11%	0.53%	0.31%	0.61%	3.06%
200	0.11%	0.21%	1.05%	0.61%	1.22%	6.12%
300	0.16%	0.32%	1.58%	0.92%	1.83%	9.17%
400	0.21%	0.42%	2.11%	1.22%	2.45%	12.2%
1000	0.53%	1.05%	5.27%	3.06%	6.12%	30.6%

While this extrapolation gives an idea about the scalability properties of the approach, further investigation would be beneficial. Yet, many factors attribute to the real time capability of a system, as mentioned above.

9 Conclusion and Outlook

System tests in automated production system (aPS) engineering in production automation are often performed under high time pressure, in an uncomfortable on-site environment at the customer's premises and lacking detailed specifications. This leads to insufficient repeatability due to a lack of documentation, uncertainty of test adequacy and inefficient or inadequate testing in case of changes. The rising complexity in the aPSs' control software and a lack of tool support increase this problem.

The aim of this thesis was to tackle these problems by developing an approach for a more structured system testing approach that enables an analysis of test coverage and a support for prioritizing test cases in case of changes to the system. The approach's foundation is a guided semi-automatic system testing approach that partially automates the system testing process while including a human tester by giving her or him tasks via a Human Machine Interface. During test execution, data regarding timing and executed statements in the code are recorded, which allow for a subsequent analysis of test coverage to identify untested system behavior. In case modifications are performed on an aPS, this information is reused in combination with a change impact analysis to prioritize test cases for regression testing, i.e. to find newly introduced faults in the code.

The approach was developed in accordance with industrial requirements, which were compiled in cooperation with several experienced experts from reputable companies active in the field of aPS engineering development. Based on these requirements, the approach was evaluated by performing several experiments in an industrial case study. The results were subsequently discussed and rated within an expert evaluation.

According to the industrial experts' opinion, the approach shows promising results in tackling the problems in aPS system testing, improving the system testing and regression testing process of aPS regarding efficiency and testing quality. Evaluation results show that when using the approach, system tests can be performed in a more repeatable manner, reducing deviations in testing quality, and improving reproducibility through automated, detailed documentation. In addition, untested functionality can be revealed, which was never possible before, increasing test coverage and reducing the possibility of remaining critical faults in aPS software. Furthermore, previously tested systems can be retested more efficiently in case of changes: the prioritization helps saving time by proposing test sequences based on the performed changes that are more likely to unveil regressions earlier.

The industrial requirements were satisfied for a representative case study and discussion showed that the approach – in its current prototypical form – is already applicable for a significant part of the machines produced by this company and aPS engineering in general. To gain a better understanding about the applicability of the approach regarding other companies and a wider range of applications, more case studies applying the approach should be conducted. Here, interesting properties to investigate in more detail could be runtime properties and benefits of the approach regarding coverage visualization and regression test prioritization with different case studies. In addition, several limits were identified in the approach's current state, which should be addressed in future research.

The runtime overhead reduces the applicability of the approach in is current state, even after significant improvements were achieved during optimization. Theoretically, this limitation stems from two factors: remaining execution time in each PLC scan cycle and complexity of the code regarding its control flow. With more control statements, more code instrumentation is required, requiring more execution time, as more inserted tracing functions need to be executed. This might lead to breaking real time requirements if not enough remaining execution time is available. Thus, the limit cannot be directly related to the code size, but might have to be calculated for each individual application using worst-case execution time analysis (Wilhelm *et al.*, 2008), e.g. using static code analysis. Yet, code structure is not the only factor influencing execution properties, but also others, such as task scheduling, caching and scan cycle time jitter. In practice, the instrumented code could be treated as regular (more computationally expensive) code and enforcing a minimum remaining execution time, e.g. using a maximum of 80% of scan cycle time as a practical measure to avoid breaches of hard real time. The runtime overhead could be improved by developing more efficient tracing algorithms or enabling coverage assessment without code instrumentation. While significant improvements of the approach used in the evaluation were achieved, efficient tracing approaches like the one proposed by Prähofer et al. (Prähofer *et al.*, 2011) that seems to have only minimal influence on runtime behavior, while allowing for recording of detailed runtime information down to variable values, could open up new possibilities for coverage analysis and regression test prioritization. Another approach, which currently is a work in progress, tries to calculate structural coverage without instrumentation and even without the need for test execution (Ulewicz *et al.*, 2017). This could enable prioritization of programs not fully tested and without influencing runtime behavior. This approach could complement the one presented in this thesis by supporting the estimation of coverage for systems with severe restrictions regarding run-time overheads.

The prioritization criteria still showed room for improvement. According to the experts, criteria such as testing modifications as quick or intensely as possible are only part of the considered properties when performing regression testing. The experts argued that relation of the test case to critical functions or influences regarding product quality were most important when selection and prioritizing test cases. Therefore, in this field, additional research regarding suitable prioritization criteria and methods for obtaining such information is required.

10 References

3S - Smart Software Solutions GmbH (2016a) *CODESYS Development System*. Available at: https://www.codesys.com/products/codesys-engineering/development-system.html (Accessed: 13 February 2018).

3S - Smart Software Solutions GmbH (2016b) *CODESYS Static Analysis*. Available at: http://store.codesys.com/codesys-static-analysis.html?___store=en (Accessed: 13 February 2018).

3S - Smart Software Solutions GmbH (2016c) *CODESYS Test Manager*. Available at: http://store.codesys.com/codesys-test-manager.html (Accessed: 13 February 2018).

Abele, S. and Weyrich, M. (2016) 'Supporting the regression test of multi-variant systems in distributed production scenarios', in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–4. doi: 10.1109/ETFA.2016.7733652.

Abele, S. and Weyrich, M. (2017) 'Decision Support for Joint Test and Diagnosis of Production Systems based on a Concept of Shared Knowledge', *IFAC-PapersOnLine*, 50 (1), pp. 15227–15232. doi: 10.1016/j.ifacol.2017.08.2374.

Abele, S., Zeller, A., Jazdi, N. and Weyrich, M. (2017) 'Agentenbasierte Testplanung für industrielle IT-Systeme in der Fertigung', *atp edition*, 59 (9), p. 28. doi: 10.17560/atp.v59i09.1880.

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers - Principles, Technique and Tools*. Addison Wesley. doi: 10.1007/s13398-014-0173-7.2.

Akesson, K., Fabian, M., Flordal, H. and Malik, R. (2006) 'Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems', in *International Workshop on Discrete Event Systems*. IEEE, pp. 384–385. doi: 10.1109/WODES.2006.382401.

Alagöz, I., Herpel, T. and German, R. (2017) 'A selection method for black box regression testing with a statistically defined quality level', in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 114–125. doi: 10.1109/ICST.2017.18.

ALL4TEC (2017) *MaTeLo*. Available at: http://www.all4tec.net/MaTeLo/homematelo.html (Accessed: 13 February 2018).

Alvarez, M. L., Sarachaga, I., Burgos, A., Estevez, E. and Marcos, M. (2016) 'A Methodological Approach to Model-Driven Design and Development of Automation Systems', *IEEE Transactions on Automation Science and Engineering*, pp. 1–13. doi: 10.1109/TASE.2016.2574644.

Ammann, P. E. and Black, P. E. (1999) ‘A specification-based coverage metric to evaluate test sets’, in *IEEE International Symposium on High-Assurance Systems Engineering*. IEEE, pp. 239–248. doi: 10.1109/HASE.1999.809499.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J. and McMinn, P. (2013) ‘An orchestrated survey of methodologies for automated software test case generation’, *Journal of Systems and Software*, 86 (8), pp. 1978–2001. doi: 10.1016/j.jss.2013.02.061.

Angerer, F., Prähofer, H., Ramler, R. and Grillenberger, F. (2013) ‘Points-to analysis of IEC 61131-3 programs: Implementation and application’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. doi: 10.1109/ETFA.2013.6648062.

Atlassian (2016) *Atlassian Clover*. Available at: https://www.atlassian.com/software/clover (Accessed: 13 February 2018).

Baller, H., Lity, S., Lochau, M. and Schaefer, I. (2014) ‘Multi-objective test suite optimization for incremental product family testing’, in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 303–312. doi: 10.1109/ICST.2014.43.

Barth, M. and Fay, A. (2013) ‘Automated generation of simulation models for control code tests’, *Control Engineering Practice*. Elsevier, 21 (2), pp. 218–230. doi: 10.1016/j.conengprac.2012.09.022.

Basile, F., Chiacchio, P. and Gerbasio, D. (2013) ‘On the Implementation of Industrial Automation Systems Based on PLC’, *IEEE Transactions on Automation Science and Engineering*, 10 (4), pp. 990–1003. doi: 10.1109/TASE.2012.2226578.

Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M. and Stursberg, O. (2004) ‘Verification of PLC Programs Given as Sequential Function Charts’, in *Integration of Software Specification Techniques for Applications in Engineering*. Springer Berlin Heidelberg, pp. 517–540. doi: 10.1007/978-3-540-27863-4_28.

Biallas, S. (2016) *Verification of Programmable Logic Controller Code using Model Checking and Static Analysis*. RWTH Aachen.

Biallas, S., Brauer, J. and Kowalewski, S. (2012) ‘Arcade.PLC: A verification platform for programmable logic controllers’, in *IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE, pp. 338–341. doi: 10.1145/2351676.2351741.

Bohlender, D., Simon, H., Friedrich, N., Kowalewski, S. and Hauck-Stattelmann, S. (2016) ‘Concolic test generation for PLC programs using coverage metrics’, in

International Workshop on Discrete Event Systems (WODES). IEEE, pp. 432–437. doi: 10.1109/WODES.2016.7497884.

Bohner, S. A. and Arnold, R. S. (1996) *Software Change Impact Analysis*. Los Alamos, CA: The Institute of Electrical and Electronic Engineers, Inc.

Bullseye (2016) *BullseyeCoverage*. Available at: http://www.bullseye.com/productInfo.html (Accessed: 13 February 2018).

Burnstein, I. (2003) *Practical Software Testing*. New York: Springer-Verlag New York, Inc. (Springer Professional Computing). doi: 10.1007/b97392.

Buzhinsky, I. and Vyatkin, V. (2017) 'Testing automation systems by means of model checking', in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–7. doi: 10.1109/ETFA.2017.8247579.

Caliebe, P., Herpel, T. and German, R. (2012) 'Dependency-based test case selection and prioritization in embedded systems', in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, pp. 731–735. doi: 10.1109/ICST.2012.164.

Cephalos GmbH (2017) *Trysim*. Available at: http://www.trysim.de/ (Accessed: 13 February 2018).

Dijkstra, E. W. (1972) 'The humble programmer', *Communications of the ACM*, 15 (10), pp. 859–866. doi: 10.1145/355604.361591.

Doganay, K., Bohlin, M. and Sellin, O. (2013) 'Search based testing of embedded systems implemented in IEC 61131-3: An industrial case study', in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 425–432. doi: 10.1109/ICSTW.2013.78.

Engström, E., Runeson, P. and Skoglund, M. (2010) 'A systematic review on regression test selection techniques', *Information and Software Technology*, 52 (1), pp. 14–30. doi: 10.1016/j.infsof.2009.07.001.

Enoiu, E. P., Doganay, K., Bohlin, M., Sundmark, D. and Pettersson, P. (2013) 'MOS: An integrated model-based and search-based testing tool for Function Block Diagrams', *International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pp. 55–60. doi: 10.1109/CMSBSE.2013.6605711.

Enoiu, E. P., Sundmark, D. and Pettersson, P. (2013) 'Model-based test suite generation for Function Block Diagrams using the UPPAAL model checker', in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 158–167. doi: 10.1109/ICSTW.2013.27.

Ernst, M. D. (2003) ‘Static and dynamic analysis: synergy and duality’, *ICSE Workshop on Dynamic Analysis (WODA)*, pp. 24–27.

Estévez, E., Pérez, F., Orive, D. and Marcos, M. (2017) ‘A novel approach for Flexible Automation Production Systems’, in *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 695–699.

Feldmann, S., Hauer, F., Ulewicz, S. and Vogel-Heuser, B. (2016) ‘Analysis framework for evaluating PLC software: An application of Semantic Web technologies’, in *IEEE International Symposium on Industrial Electronics (ISIE)*. IEEE, pp. 1048–1054. doi: 10.1109/ISIE.2016.7745037.

Feldmann, S., Ulewicz, S., Diehm, S. and Vogel-Heuser, B. (2016) ‘Strukturelle Codeanalyse’, *atp edition*, 58 (9), pp. 54–63. doi: 10.17560/atp.v58i09.578.

Fernández Adiego, B., Darvas, D., Blanco Viñuela, E., Tournier, J.-C., Bliudze, S., Blech, J. O. and González Suárez, V. M. (2015) ‘Applying model checking to industrial-sized PLC programs’, *IEEE Transactions on Industrial Informatics*, 11 (6), pp. 1400–1410. doi: 10.1109/TII.2015.2489184.

Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A. and Marinov, D. (2013) ‘Comparing non-adequate test suites using coverage criteria’, in *International Symposium on Software Testing and Analysis (ISSTA)*. New York, New York, USA: ACM Press, p. 302. doi: 10.1145/2483760.2483769.

Gopinath, R., Jensen, C. and Groce, A. (2014) ‘Code coverage for suite evaluation by developers’, *International Conference on Software Engineering (ICSE)*, pp. 72–82. doi: 10.1145/2568225.2568278.

Gourcuff, V., de Smet, O. and Faure, J.-M. (2008) ‘Improving large-sized PLC programs verification using abstractions’, *IFAC Proceedings Volumes*, 41 (2), pp. 5101–5106. doi: 10.3182/20080706-5-KR-1001.00857.

Hackenberg, G., Campetelli, A., Legat, C., Mund, J., Teufl, S. and Vogel-Heuser, B. (2014) ‘Formal Technical Process Specification and Verification for Automated Production Systems’, in *International Conference on System Analysis and Modeling (SAM): Models and Reusability*, pp. 287–303. doi: 10.1007/978-3-319-11743-0_20.

Hametner, R., Kormann, B., Vogel-Heuser, B., Winkler, D. and Zoitl, A. (2011) ‘Test case generation approach for industrial automation systems’, in *International Conference on Automation, Robotics and Applications (ICARA)*. IEEE, pp. 57–62. doi: 10.1109/ICARA.2011.6144856.

Hametner, R., Zoitl, A. and Semo, M. (2010) ‘Automation component architecture for the efficient development of industrial automation systems’, in *IEEE International*

Conference on Automation Science and Engineering (CASE). IEEE, pp. 156–161. doi: 10.1109/COASE.2010.5584013.

Hoffmann, D. W. (2013) *Software-Qualität*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-35700-8.

IBM (2016) *Rational DOORS*. Available at: http://www-03.ibm.com/software/products/en/ratidoor (Accessed: 13 February 2018).

IEC (2003) 'IEC 61131 Programmable Controllers - Part 3: Programming Languages (Second Edition)'. International Electrotechnical Commission Std.

IEC (2012) 'IEC 61499 Function Blocks - Part 1: Architecture'. International Electrotechnical Commission Std.

IEC (2013) 'IEC 61131 Programmable Controllers - Part 3: Programming Languages (Third Edition)'. International Electrotechnical Commission Std.

IEEE (2008) 'IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation'. doi: 10.1109/IEEESTD.2008.4578383.

ISO/IEC/IEEE (2010) 'ISO/IEC/IEEE 24765:2010 Systems and Software Engineering - Vocabulary'. ISO/IEC/IEEE. doi: 10.1109/IEEESTD.2015.7106438.

Itris Automation (2017) *PLC Checker*. Available at: http://www.itris-automation.com/plc-checker/ (Accessed: 13 February 2018).

Jee, E., Kim, S., Cha, S. and Lee, I. (2010) 'Automated test coverage measurement for reactor protection system software implemented in function block diagram', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6351 LNCS, pp. 223–236. doi: 10.1007/978-3-642-15651-9_17.

Jee, E., Yoo, J. and Cha, S. (2005) 'Control and data flow testing on function block diagrams', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3688 LNCS, pp. 67–80. doi: 10.1007/11563228_6.

Jee, E., Yoo, J., Cha, S. and Bae, D. (2009) 'A data flow-based structural testing technique for FBD programs', *Information and Software Technology*. Elsevier B.V., 51 (7), pp. 1131–1139. doi: 10.1016/j.infsof.2009.01.003.

Jones, J. A. and Harrold, M. J. (2001) 'Test-suite reduction and prioritization for modified condition/decision coverage', in *IEEE International Conference on Software Maintenance*. IEEE Comput. Soc., pp. 92–101. doi: 10.1109/ICSM.2001.972715.

Jung, S., Yoo, J. and Lee, Y.-J. (2017) ‘A PLC platform-independent structural analysis on FBD programs for digital reactor protection systems’, *Annals of Nuclear Energy*. Elsevier, 103, pp. 454–469. doi: 10.1016/j.anucene.2017.02.006.

Kim, J.-M. and Porter, A. (2002) ‘A history-based test prioritization technique for regression testing in resource constrained environments’, in *IEEE International Conference on Software Engineering*. New York, New York, USA: ACM Press, p. 119. doi: 10.1145/581339.581357.

Kormann, B., Tikhonov, D. and Vogel-Heuser, B. (2012) ‘Automated PLC software testing using adapted UML sequence diagrams’, in *IFAC Proceedings Volumes (IFAC-PapersOnline)*. Bucharest, Romania, pp. 1615–1621. doi: 10.3182/20120523-3-RO-2023.00148.

Kottler, S., Khayamy, M., Hasan, S. R. and Elkeelany, O. (2017) ‘Formal verification of ladder logic programs using NuSMV’, in *SoutheastCon 2017*. IEEE, pp. 1–5. doi: 10.1109/SECON.2017.7925390.

Krause, J. (2012) *Testfallgenerierung aus modellbasierten Systemspezifikationen auf der Basis von Petrinetzentfaltungen*. ifak Magdeburg.

Kumar, B., Gilani, S. S., Niggemann, O. and Schäfer, W. (2013) ‘Automated test case generation from complex environment models for PLC control software testing and maintenance’, in *VDI-Kongress Automation*, pp. 129–134.

Lauber, R. and Göhner, P. (1999) *Prozessautomatisierung 1*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-58446-6.

Liu, Z., Magnus, S., Krause, J. and Diedrich, C. (2014) ‘Concept for modelling and testing of individual mechatronic components for manufacturing plant simulation’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–7. doi: 10.1109/ETFA.2014.7005147.

Ljungkrantz, O., Åkesson, K., Fabian, M. and Chengyin Yuan (2010) ‘Formal Specification and Verification of Industrial Control Logic Components’, *IEEE Transactions on Automation Science and Engineering*, 7 (3), pp. 538–548. doi: 10.1109/TASE.2009.2031095.

Lochau, M., Bürdek, J., Lity, S., Hagner, M., Legat, C., Goltz, U. and Schürr, A. (2014) ‘Applying model-based Software Product Line testing approaches to the automation engineering domain’, *at - Automatisierungstechnik*, 62 (11), pp. 771–780. doi: 10.1515/auto-2014-1099.

Logic Design Inc. (2017) *PLCLogix*. Available at: https://www.plclogix.com/ (Accessed: 13 February 2018).

Louridas, P. (2006) ‘Static code analysis’, *IEEE Software*, 23 (4), pp. 58–61. doi: 10.1109/MS.2006.114.

De Lucia, A., Fasano, F. and Oliveto, R. (2008) ‘Traceability management for impact analysis’, in *Frontiers of Software Maintenance*. IEEE, pp. 21–30. doi: 10.1109/FOSM.2008.4659245.

Ma, C. and Provost, J. (2017a) ‘A model-based testing framework with reduced set of test cases for programmable controllers’, *IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 944–949.

Ma, C. and Provost, J. (2017b) ‘Using plant model features to generate reduced test cases for programmable controllers’, *IFAC-PapersOnLine*, 50 (1), pp. 11163–11168. doi: 10.1016/j.ifacol.2017.08.1238.

Malz, C. and Göhner, P. (2011) ‘Agent-Based Test Case Prioritization’, in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 149–152. doi: 10.1109/ICSTW.2011.81.

Malz, C., Jazdi, N. and Göhner, P. (2012) ‘Prioritization of test cases using software agents and fuzzy logic’, in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 483–486. doi: 10.1109/ICST.2012.131.

Maruchi, K., Shin, H. and Sakai, M. (2014) ‘MC/DC-like structural coverage criteria for Function Block Diagrams’, in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, pp. 253–259. doi: 10.1109/ICSTW.2014.27.

Mewes & Partner GmbH (2017) *WinMOD*. Available at: http://www.winmod.de/en/ (Accessed: 13 February 2018).

Nair, S., Jetley, R., Nair, A. and Hauck-Stattelmann, S. (2015) ‘A static code analysis tool for control system software’, in *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, pp. 459–463. doi: 10.1109/SANER.2015.7081856.

Neumann, P., Grötsch, E., Lubkoll, C. and Simon, R. (2000) *SPS-Standard: IEC 61131: Programmierung in verteilten Automatisierungssystemen*. 3rd Editio. München: Oldenburg Industieverlag.

NuSMV (2015). Available at: http://nusmv.fbk.eu/ (Accessed: 13 February 2018).

nuXmv (2014). Available at: https://nuxmv.fbk.eu/ (Accessed: 13 February 2018).

Orso, A., Apiwattanapong, T. and Harrold, M. J. (2003) ‘Leveraging field data for impact analysis and regression testing’, *ACM SIGSOFT Software Engineering Notes*, 28 (5), p. 128. doi: 10.1145/949952.940089.

Parr, T. (2007) *The Definitive ANTLR Reference - Building Domain-Specific Languages*. The Pragmatic Bookshelf.

Pinkal, K. and Niggemann, O. (2017) ‘A new approach to model-based test case generation for industrial automation systems’, in *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, pp. 53–58. doi: 10.1109/INDIN.2017.8104746.

Piwowarski, P., Ohba, M. and Caruso, J. (1993) ‘Coverage measurement experience during function test’, *Proceedings of 1993 15th International Conference on Software Engineering*, pp. 287–301. doi: 10.1109/ICSE.1993.346035.

PLCopen (2013) *PLCopen IEC 61131-3: a standard programming resource*. Available at: http://www.plcopen.org/pages/promotion/publications/downloads/intro_iec_ma rch2013.pdf (Accessed: 5 May 2017).

PLCopen (2017) *PLCopen Website*. Available at: http://www.plcopen.org (Accessed: 13 February 2018).

Prähofer, H., Angerer, F., Ramler, R. and Grillenberger, F. (2016) ‘Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application’, *IEEE Transactions on Industrial Informatics*, 3203 (ii), pp. 1–10. doi: 10.1109/TII.2016.2604760.

Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H. and Grillenberger, F. (2012) ‘Opportunities and challenges of static code analysis of IEC 61131-3 programs’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. doi: 10.1109/ETFA.2012.6489535.

Prähofer, H., Schatz, R., Wirth, C. and Mössenböck, H. (2011) ‘A comprehensive solution for deterministic replay debugging of SoftPLC Applications’, *IEEE Transactions on Industrial Informatics*, 7 (4), pp. 641–651. doi: 10.1109/TII.2011.2166768.

Provost, J., Roussel, J.-M. and Faure, J.-M. (2011) ‘Translating Grafcet specifications into Mealy machines for conformance test purposes’, *Control Engineering Practice*, 19 (9), pp. 947–957. doi: 10.1016/j.conengprac.2010.10.001.

Provost, J., Roussel, J.-M. and Faure, J.-M. (2014) ‘Generation of Single Input Change Test Sequences for Conformance Test of Programmable Logic Controllers’, *IEEE Transactions on Industrial Informatics*, 10 (3), pp. 1696–1704. doi: 10.1109/TII.2014.2315972.

Puntel-Schmidt, P. and Fay, A. (2015) ‘Levels of Detail and Appropriate Model Types for Virtual Commissioning in Manufacturing Engineering’, *IFAC-PapersOnLine*. Elsevier Ltd., 48 (1), pp. 922–927. doi: 10.1016/j.ifacol.2015.05.027.

Puntel-Schmidt, P., Fay, A., Riediger, W., Schulte, T., Köslin, F. and Diehl, S. (2014) ‘Validierung von Steuerungscode fertigungstechnischer Anlagen mit Hilfe automatisch generierter Simulationsmodelle’, in *Entwurf komplexer Automatisierungssysteme (EKA)*. Magdeburg. doi: 10.1515/auto-2014-1127.

Rösch, S., Ulewicz, S., Provost, J. and Vogel-Heuser, B. (2015) 'Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains—Current Challenges and Research Gaps', *Journal of Software Engineering and Applications*, 8 (9), pp. 499–519. doi: 10.4236/jsea.2015.89048.

Rösch, S. and Vogel-Heuser, B. (2017) 'A light-weight fault injection approach to test automated production system PLC software in industrial practice', *Control Engineering Practice*. Elsevier, 58 (March 2016), pp. 12–23. doi: 10.1016/j.conengprac.2016.09.012.

Rothermel, G. and Harrold, M. J. (1997) 'A safe, efficient regression test selection technique', *ACM Transactions on Software Engineering and Methodology*, 6 (2), pp. 173–210. doi: 10.1145/248233.248262.

Rothermel, G., Untch, R. H., Chengyun Chu and Harrold, M. J. (2001) 'Prioritizing test cases for regression testing', *IEEE Transactions on Software Engineering*, 27 (10), pp. 929–948. doi: 10.1109/32.962562.

RTCA (1992) 'RTCA DO-178B: Software Considerations in Airborne Systems and Equipment Certification'.

Runeson, P., Höst, M., Rainer, A. and Regnell, B. (2012) *Case Study Research in Software Engineering*, *John Wiley & Sons, Inc*. Hoboken, NJ, USA: John Wiley & Sons, Inc. doi: 10.1002/9781118181034.

Sepp.med GmbH (2017) *MBTsuite*. Available at: https://www.seppmed.de/de/portfolio/mbtsuite/ (Accessed: 13 February 2018).

Siegl, S. and Caliebe, P. (2011) 'Improving model-based verification of embedded systems by analyzing component dependences', in *IEEE International Symposium on Industrial and Embedded Systems*. IEEE, pp. 51–54. doi: 10.1109/SIES.2011.5953678.

Siemens (2016) *Polarion*. Available at: https://polarion.plm.automation.siemens.com/ (Accessed: 13 February 2018).

Simon, H., Friedrich, N., Biallas, S., Hauck-Stattelmann, S., Schlich, B. and Kowalewski, S. (2015) 'Automatic test case generation for PLC programs using coverage metrics', in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. doi: 10.1109/ETFA.2015.7301602.

Sinha, R., Pang, C., Martínez, G. S. and Vyatkin, V. (2016) 'Automatic test case generation from requirements for industrial cyber-physical systems', *at - Automatisierungstechnik*, 64 (3), pp. 216–230. doi: 10.1515/auto-2015-0075.

Srikanth, H., Williams, L. and Osborne, J. (2005) 'System test case prioritization of new and regression test cases', in *International Symposium on Empirical Software Engineering*. IEEE, pp. 62–71. doi: 10.1109/ISESE.2005.1541815.

Stattelmann, S., Biallas, S., Schlich, B. and Kowalewski, S. (2014) ‘Applying static code analysis on industrial controller code’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. doi: 10.1109/ETFA.2014.7005254.

Süß, S., Magnus, S., Thron, M., Zipper, H., Odefey, U., Fassler, V., Strahilov, A., Klodowski, A., Bar, T. and Diedrich, C. (2016) ‘Test methodology for virtual commissioning based on behaviour simulation of production systems’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–9. doi: 10.1109/ETFA.2016.7733624.

Thonnessen, D., Reinker, N., Rakel, S. and Kowalewski, S. (2017) ‘A concept for PLC hardware-in-the-loop testing using an extension of structured text’, in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–8. doi: 10.1109/ETFA.2017.8247580.

Ulewicz, S., Feldmann, S., Vogel-Heuser, B. and Diehm, S. (2016) ‘Visualisierung und Analyseunterstützung von Zusammenhängen in SPS-Programmen zur Verbesserung der Modularität und Wiederverwendung’, in *VDI-Kongress Automation*.

Ulewicz, S., Schütz, D. and Vogel-Heuser, B. (2014) ‘Software changes in factory automation: Towards automatic change based regression testing’, in *Annual Conference of the IEEE Industrial Electronics Society (IECON)*. IEEE, pp. 2617–2623. doi: 10.1109/IECON.2014.7048875.

Ulewicz, S., Simon, H., Bohlender, D., Obster, M., Kowalewski, S. and Vogel-Heuser, B. (2017) ‘A priori test coverage estimation for automated production systems: Using generated behavior models for coverage calculation’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, pp. 1–4. doi: 10.1109/ETFA.2017.8247704.

Ulewicz, S., Ulbrich, M., Weigl, A., Kirsten, M., Wiebe, F., Beckert, B. and Vogel-Heuser, B. (2016) ‘A verification-supported evolution approach to assist software application engineers in industrial factory automation’, in *IEEE International Symposium on Assembly and Manufacturing (ISAM)*. IEEE, pp. 19–25. doi: 10.1109/ISAM.2016.7750714.

Ulewicz, S. and Vogel-Heuser, B. (2016a) ‘Guided semi-automatic system testing in factory automation’, in *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, pp. 142–147. doi: 10.1109/INDIN.2016.7819148.

Ulewicz, S. and Vogel-Heuser, B. (2016b) ‘System regression test prioritization in factory automation: Relating functional system tests to the tested code using field data’, in *Annual Conference of the IEEE Industrial Electronics Society (IECON)*. IEEE, pp. 4619–4626. doi: 10.1109/IECON.2016.7792997.

Ulewicz, S. and Vogel-Heuser, B. (2018a) ‘Increasing system test coverage in production automation systems’, *Control Engineering Practice*, 73, pp. 171–185. doi: 10.1016/j.conengprac.2018.01.010.

Ulewicz, S. and Vogel-Heuser, B. (2018b) ‘Industrially Applicable System Regression Test Prioritization in Factory Automation’, *Transactions on Automation Science and Engineering,* accepted publication.

Unicom (2016) *PurifyPlus*. Available at: https://teamblue.unicomsi.com/products/ purifyplus/ (Accessed: 5 May 2017).

Uppsala University (UPP) and Aalborg University (AAL) (2010) *UPPAAL*. Available at: http://www.uppaal.org/ (Accessed: 13 February 2018).

Utting, M., Pretschner, A. and Legeard, B. (2012) ‘A taxonomy of model-based testing approaches’, *Software Testing, Verification and Reliability*, 22 (5), pp. 297–312. doi: 10.1002/stvr.456.

Vogel-Heuser, B., Fay, A., Schaefer, I. and Tichy, M. (2015) ‘Evolution of software in automated production systems: Challenges and research directions’, *Journal of Systems and Software*. Elsevier Ltd., 110, pp. 54–84. doi: 10.1016/j.jss.2015.08.026.

Vogel-Heuser, B., Kormann, B., Tikhonov, D. and Rösch, S. (2013) ‘Automatisierter modellbasierter Applikationstest für SPS Steuerungsprogramme auf der Basis von UML’, *at - Automatisierungstechnik*, 61 (6), pp. 382–392. doi: 10.1524/auto.2013.0033.

Vyatkin, V. (2013) ‘Software Engineering in Industrial Automation: State-of-the-Art Review’, *IEEE Transactions on Industrial Informatics*, 9 (3), pp. 1234–1249. doi: 10.1109/TII.2013.2258165.

Whalen, M. W., Rajan, A., Heimdahl, M. P. E. and Miller, S. P. (2006) ‘Coverage metrics for requirements-based testing’, in *International Symposium on Software Testing and Analysis*. New York, New York, USA: ACM Press, p. 25. doi: 10.1145/1146238.1146242.

Wilhelm, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C. and Heckmann, R. (2008) ‘The worst-case execution-time problem—overview of methods and survey of tools’, *ACM Transactions on Embedded Computing Systems*, 7 (3), pp. 1–53. doi: 10.1145/1347375.1347389.

Wu, X., Li, J. J., Weiss, D. and Lee, Y. (2007) ‘Coverage-based testing on embedded systems’, *IEEE International Conference on Software Engineering*. doi: 10.1109/AST.2007.8.

Yang, M. C. K. and Chao, A. (1995) ‘Reliability-Estimation & Stopping-Rules for Software Testing, Based on Repeated Appearances of Bugs’, *IEEE Transactions on Reliability*, 44 (2), pp. 315–321. doi: 10.1109/24.387388.

Yang, Q., Li, J. J. and Weiss, D. M. (2009) ‘A Survey of Coverage-Based Testing Tools’, *The Computer Journal*, 52 (5), pp. 589–597. doi: 10.1093/comjnl/bxm021.

Yoo, S. and Harman, M. (2012) ‘Regression testing minimization, selection and prioritization: a survey’, *Software Testing, Verification and Reliability*, 22 (2), pp. 67–120. doi: 10.1002/stvr.430.

Zeller, A. and Weyrich, M. (2015) ‘Test case selection for networked production systems’, in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. doi: 10.1109/ETFA.2015.7301604.

Zhu, H., Hall, P. A. V. and May, J. H. R. (1997) ‘Software unit test coverage and adequacy’, *ACM Computing Surveys*, 29 (4), pp. 366–427. doi: 10.1145/267580.267590.

11 Table of Figures

12 Table of Tables