TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

DISSERTATION

Test Driven Software Development for Improving the Quality of Control Software for Industrial Automation Systems

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften (Dr.techn.)

unter der Leitung von
Univ.–Prof. Dipl.-Ing. Dr.sc. techn. Georg Schitter
E376
Institut für Automatisierungs– und Regelungstechnik

eingereicht an der
Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von
Dipl.-Ing. Reinhard Hametner
Matrikelnummer: 0125670
Schlossstrasse 13, 3250 Wieselburg, Österreich

Wien, im Oktober 2013

Dekan:	Univ.–Prof. Dipl.-Ing. Dr.techn. Gottfried Magerl
Tag des Kolloquiums:	01. 10. 2013
Vorsitzender:	Univ.–Prof. Mag.rer.nat. Dr.rer.nat. Gottfried Strasser
Erster Gutachter:	Univ.–Prof. Dipl.-Ing. Dr.sc. techn. Georg Schitter
Zweiter Gutachter:	Prof. Dr.-Ing. Georg Frey

Vorwort

Diese Arbeit wurde im Jahr 2009 im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Automatisierungs– und Regelungstechnik an der Technischen Universität Wien begonnen. Im Oktober 2013 wurde die Arbeit in ähnlicher Form als Dissertation an der Fakultät für Elektrotechnik und Informationstechnik an der Technischen Universität Wien eingereicht und erfolgreich verteidigt.

Wien, im Oktober 2013 — Dipl.-Ing. Reinhard Hametner

Acknowledgements

First of all I would like to thank the contributions of a large number of people, who have provided in different ways invaluable support for the completion of this thesis.

Most of all I would like to thank Professor Georg Schitter, who conducted my thesis as supervisor during a long period of time. Under his supervision I learned a lot according to scientific standards at a high level and be precise in formulate scientific statements. I gratefully acknowledge Professor Georg Frey from the Saarland University for co-supervising this work.

My special thanks are dedicated to Dr. Alois Zoitl for his support and fruitful discussions during my work. He spent a lot of time and energy in discussing my results. He teaches me to write scientific papers from the very beginning as well as helping me to write scientific research grants. Furthermore, I want to thank Professor Gerfried Zeichen for the discussions about industry challenges during my work on different research projects.

This dissertation came to standing during my employment as a research assistant at the Automation and Control Institute at Vienna University of Technology. I want to thank all colleagues for providing a highly innovative and productive environment, especially Monika Wenger, Ingo Hegny, Martin Melik-Merkumians, Michael Steinegger, Dr. Gottfried Koppensteiner, Wilfried Lepuschitz, Alexander Prostejovsky, and Dr. Munir Merdan.

Furthermore, I would like to thank Dietmar Winkler and Thomas Östreicher from the Institute of Software Technology and Interactive Systems for the fruitful discussions and the pleasant cooperation. Numerous scientific publications are the results of our cooperation which are presented on international scientific conferences all over the world.

Many warm thanks go to my parents Maria and Franz Hametner and my brother Christoph for their support during the long period of my education. They frequently asking me about the status quo of my thesis with the state-

ment: “Und wie schauts aus?”

Finally and above all, I would like to thank my wonderful girlfriend Melanie for her devoted love and her incessant patience when this thesis was at the forefront of my thoughts. Thank you for supporting me during all these years and for pursuing me to finish this thesis as soon as possible.

Parts of this work were funded by the Austrian Research Funding Agency (FFG) grant logi.DIAG (Bridge7-196929).

REINHARD HAMETNER

Abstract

In today's world the complexity of industrial automation systems increases rapidly. Because of this complexity, the development of such automation systems becomes more time consuming and has to deal with adaptations of the industrial automation code on short notice. The engineering efficiency has to be increased in terms of reusability support of tested and testable software components for upcoming projects in order to reduce the development time and costs.

The intention of this thesis is to make a significant step towards an increase in the quality of industrial automation software by providing a test infrastructure with appropriate testing techniques for industrial automation software. To ensure the high software quality, testing processes are necessary which assist the development engineers during the life-cycle of developing industrial automation systems. Currently there is no systematic test process for testing industrial control software in the industry. First steps are available in academia to adapt testing techniques from the software engineering domain applied on the industrial automation domain.

A new test framework is developed in this thesis which is able to support testing of industrial automation software, i.e. based on IEC 61131 and IEC 61499, on various levels of detail and from different perspectives. New criteria are proposed for selecting Unified Modeling Language (UML) models which are useable for specifying tests of industrial automation software. According to the developed test framework, different testing techniques considering the different test levels are presented which are able to test industrial automation software considering the Test-First Development (TFD) strategy. Finally, a new automation component architecture and a new component design for developing industrial automation applications are introduced to support testability. Therefore, it is now possible to increase the quality of industrial control software by using the newly developed test infrastructure.

Kurzzusammenfassung

Der starke Konkurrenzdruck der Märkte lässt die Anforderungen heutiger industrieller Automatisierungssysteme rapide ansteigen. Dies führt zu einer Erhöhung der Komplexität solcher Systeme. Darum steigt der Aufwand in der Entwicklung, welche mit ständigen Adaptierungen des Software Codes verbunden ist. Durch den Einsatz von wiederverwendbaren und testbaren Komponenten kann die Effizienz für zukünftige Projekte gesteigert werden. Zusätzlich können die Entwicklungszeiten und Entwicklungskosten durch Verwendung solcher Komponenten reduziert werden.

Die Intention dieser Doktorarbeit ist, einen signifikanten Schritt zur Steigerung der Qualität von Steuerungssoftware beizutragen und eine Testinfrastruktur mit geeigneten Techniken zum Testen einer solchen Software bereitzustellt. Um eine hohe Softwarequalität zu erreichen ist ein Testprozess notwendig, welcher dem/der Ingenieur/in während der ganzen Entwicklungszeit unterstützend zur Verfügung stehen soll. Derzeit gibt es keine systematischen Testprozesse für Software im Bereich der industriellen Steuerungstechnik. Erste Ansätze sind im akademischen Bereich erforscht. Hierzu werden Techniken aus dem Bereich der Informatik, im Speziellen der Softwareentwicklung, adaptiert. Diese Doktorarbeit zeigt eine neue Teststruktur, die das Testen von industrieller Steuerungssoftware nach dem Standard IEC 61131 und IEC 61499 an unterschiedlichen Testebenen mit unterschiedlichen Perspektiven unterstützt. Geeignete Unified Modeling Language (UML) Modelle für die Testspezifikation werden vorgestellt. Weiters werden Testmethoden und Techniken diskutiert, die für das Testen von Steuerungssoftware nach der "Test-First Development" Strategie geeignet sind. Abschließend wird eine neue komponentenbasierte Architektur und ein neues Komponentendesign für die Entwicklung testbarer Steuerungsapplikationen präsentiert. Nun ist es möglich, die Qualität der industriellen Steuerungssoftware mithilfe der neu entwickelten Testinfrastruktur zu erhöhen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

A

AC Automation Component. xi, 125–131
AI Artificial Intelligence. 38

B

BFB Basic Function Block. x, 19, 27, 54–56, 59, 102, 116–120

C

CFB Composite Function Block. 19, 54–56, 59, 88, 120
CVS Concurrent Versions System. 22
CWM Common Warehouse Metamodel. 24

D

DCS Distributed Control System. 28
DES Discrete Event Simulator. 21
DSP Digital Signal Processors. 21

E

ECC Execution Control Chart. x, xi, 19, 27, 54, 55, 87, 88, 91, 101, 102, 116–119
ES Embedded System. 39–41

F

FB Function Block. ix, x, xii, 15, 17–20, 27, 28, 42, 46, 48, 54–56, 76–78, 86–89, 91, 92, 94–96, 98, 99, 101–104, 109–115, 118–121, 133
FBD Function Block Diagram. 17

FFA Federal Aviation Administration. 7
FMEA Failure Mode and Effects Analysis. 129
FSM Finite State Machine. 6, 7, 26, 30, 74

H

HAZOP Hazard and Operability. 129
HiL Hardware-in-the-Loop. 21, 41

I

IEC International Electrotechnical Commission. 15, 17, 31
IL Instruction List. 17
IPMCS Industrial Process Measurement and Control Systems. 17
ISO International Organization for Standardization. 12, 29

K

KDT Keyword-driven Testing. x, xiii, 81–84, 103–106, 109, 110, 120–122, 133

L

LD Ladder Diagram. 17
LOC Lines of Code. 135

M

M2C Model-to-Code. 26, 27
M2M Model-to-Model. 27, 99, 122, 123, 134
M2T Model-to-Text. 27
MBT Model-based Testing. 36–39, 41, 43, 95, 96, 115, 119–123, 133, 134
MDA Model-Driven Architecture. 24
MDD Model-Driven Development. 25, 27, 28
MDSD Model-Driven Software Development. 24
MiL Model-in-the-Loop. 41
MOF Meta Object Facility. 24

O

OCL Object Constraint Language. 28, 29
OMG Object Management Group. 24, 29

P

PIM Platform Independent Model. 24
PLC Programmable Logic Controller. 1, 9, 15, 16, 20–23, 27, 28, 39, 67, 75, 79, 107–109, 121–123

POU Program Organization Unit. 15–17
PSM Platform Specific Model. 24

Q

QA Quality Assurance. 12, 61, 62

R

RUP Rational Unified Process. 26

S

SFC Sequential Function Chart. 17
SIFB Service Interface Function Block. 19, 27, 54, 55, 89, 94, 120
SiL Software-in-the-Loop. 41
ST Structured Text. 17, 48, 88, 101
SuT System under Test. 2, 5–8, 12, 13, 22, 25, 29, 33, 34, 37, 50, 74–76, 79–85, 96, 97, 101, 103, 105, 107–111, 119, 120, 122, 133, 135
SysML Systems Modeling Language. 27, 62

T

TDA Test-Diagnosis-Automation. 134, 135
TDD Test-Driven Development. 10, 34, 44, 46, 62, 64, 65, 72
TFD Test-First Development. iv, ix, 8, 30, 34–36, 41, 44, 46, 49, 51, 52, 54, 60, 83, 89, 95, 112, 114, 132–134
TPT Time Partition Testing. 41

U

UML Unified Modeling Language. iv, v, 10, 24–30, 36, 38, 39, 61, 62, 64–66, 71, 72, 74, 78, 80, 81, 88–92, 95, 96, 99, 103, 105, 115, 118–122, 133–135

CHAPTER 1

Introduction

The pressure of the competition in the global economy increases steadily. Ongoing changing markets with decreasing product life cycles are a great challenge for manufacturing industries. To be competitive in the field of automation system development, companies must bring their innovative products into the market faster than their competitors. The biggest challenge is to manage the increasing complexity of the requirements in industrial production systems. A further challenge is to handle the ever-changing requirements of produced goods during the development phase. A change of the mechanical system also entails changes of the electrical, electronic, and the software part of the control application. Therefore there is a need for an efficient development process which handles the immediate reaction of changing requirements during the development phase.

The functional requirements of nowadays' industrial production systems rapidly increases. There is the requirement in the industrial automation domain to shift functional implementations from hardware components to software components to increase the flexibility. Such software components are program parts which are used for the most important type of control devices named Programmable Logic Controllers (PLCs). Software components (i.e., industrial automation code) become more important and allow to cope with the ever increasing complexity of modern applications. The development of such software components for industrial production systems becomes more time-consuming and deals with adaptations of the control code on short notice. Furthermore, the engineering efficiency has to be increased by a reusability support of software components for upcoming projects to reduce the development time.

In the automation systems domain there is an observation of a strong en-

gineering focus on hardware development on the one side and limited experience in software engineering know-how on the other side. Usually tests are only conducted with respect to hardware because software is still considered a "by-product" of hardware development. Additionally, machines are in operation for several decades but the control software is extended and adapted in much shorter periods. However, low software quality and the increased use of untested software in automation systems bears high risks to fulfill the overall quality requirements, the robustness, and the reliability of the system. Furthermore, the reuse of untested software components in multiple automation applications spreads this risk. Thus software tests can help to increase and keep up the high software quality standards over several applications. Many companies associate testing with additional labor costs, e.g. test engineers, and set the test process to a low priority in the development phase to save project costs. That bears a higher risk because failures are more expensive to be fixed the later they are found in the development phase [2].

Future production systems and their control application software must be flexible, adaptable, reusable, and quickly expandable with low financial and temporal resources [3]. Major costs of such flexible production systems are costs incorporated with down-times of production facilities, for instance, in the case of changes or break-downs, e.g. continuous processes with long start-up times. In order to support software quality assurance, the control code has to be tested systematically, thereby the risks of fatal software failures are reduced [4].

In this thesis the term "control code" is used in the context of software for industrial automatic control. The quality of the control code has a significant influence on the reliability and safety of machines as well as the quality of the produced goods. To ensure the quality and reliability of the control code, testing during the development phases is needed. With conventionally used software testing techniques a high degree of human interaction is necessary to check the System under Test (SuT) which is partly unmanageable at present and harder in the future [5]. Therefore a systematic testing process is needed to cope with complex industrial automation applications.

1.1 Problem Statement

Currently there is no systematic testing process and there are no approaches to check the software quality for industrial applications in the field of automation systems. Often new projects and project parts are copied from previous projects and modified based on the new requirements. Each 'copy-and-modify' cycle reduces the software quality which results in numerous untested software parts in new projects [6]. Furthermore, such software systems are difficult to maintain which results in extremely time-consuming and long pro-

cesses. Through the high effort of the industrial applications development, the test effort will be reduced, resulting in a low software quality. In order to solve these issues a systematic test process with appropriate testing methods applicable for the industrial automation domain are required.

1.2 Goal

The goal of this thesis is to develop methods for systematic testing of industrial control code, in order to improve the software quality in industrial automation applications. In order to achieve this, an infrastructure and environment for support testing of industrial automation software is developed.

Current industrial applications are mostly implemented in IEC 61131-3, but industrial implementations based on the IEC 61499 standard increase [7, 8]. Hence, a general testing approach for testing industrial applications in prevalent industrial modeling languages is needed and will be conceptualized. Also a test environment is developed and supports testing of the desired functional behavior of the control code during the development process.

1.3 Software Testing

> "Testing is the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component." [9, p. 76]

Software systems become more important in most aspects of production systems and therefore the demand of software quality increases. This tends to new technologies which support the development of high quality systems in industrial applications. Without testing there is no way of establishing a decent quality of a developed system. Faults can cause minor disturbances, e.g. applications by using a keyboard for the input and a push-button does not work, but others can be potentially life threatening, e.g. a fault in a production system with human interaction [10]. There is the need to ensure the software quality of developed systems. This can be achieved by testing which is a method of quality assurance that verifies the behavior of a system towards a set of requirements that are expressed as tests [11].

In this section fundamental terms from software testing are defined for a clear understanding. Further, commonly used basic software testing techniques are presented. Finally, software testing versus software verification techniques are evaluated.

1.3.1 Fundamental Terms

In the following several fundamental terms that are important for testing software are defined and explained in more detail. These terms are defined based on [9, 12, 13]:

- A *failure* is the incorrect behavior of a program, observable during the occurrence of a fault.
- A *fault*, also known as *error* or *mistake*, is a localized code artifact which can result in a failure.
- A *symptom* is a non-intended program behavior and can result in a failure, e.g. memory leak.
- A *bug* refers to a symptom or a fault. Other used words for a bug are defect, anomaly, problem, or issue.
- *Debugging* is an analyzing method for identifying a bug.
- *Dead code* is a code fragment or part of a code which will never be considered and executed.
- *Verification* is a process to prove if a program complies with its specification by a mathematical prove, i.e. formal verification.
- *Validation* is the process of evaluating software to ensure compliance with the requirements at the end of the development process.

Software faults occur through the following chain of events. The engineer incidentally introduces a fault to the code during the implementation. If this incorrect program is executed, the system will produce wrong results. Note that not all defects will necessarily result in failures. For instance, defects in dead code will never result in failures. A symptom can turn into a fault when the run-time environment is changed.

Edsger W. Dijkstra [14] clearly stated the main drawback of software testing in the early 1970's.

> "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence!" [14]

1.3.2 Test Methods

In the field of software testing a test procedure can be organized in *static testing* and *dynamic testing* [15, 16]. The simplest static testing process is a compiler sweep. The compliance with the formal criteria of the program guidelines would also be a static test. In addition, a plausibility check of the hardware can be done as a static test.

A dynamic test is when the program code is executed and then the test code/process is executed and applied on the SuT in order to check the correct behavior of the test reference.

Black-box testing, *White-box testing*, and *Gray-box testing* are three basic techniques in the field of software testing. In the following the three testing methods are presented.

Black-box Testing is understood as *functional testing* to test the behavior of the code against the specification without knowing the assumptions about the internal structure of the software component.

The term "behavioral testing" is marginal different because the use of internal knowledge is not strictly forbidden. Classical Black-box tests evaluate the pure input-output behavior of the SuT. Therefore the goal is to test the software behavior towards the specification of the software.

In the following several test design techniques are presented and explained in more detail based on [17–20], [13, p. 35ff], and [21, p. 11]:

Equivalence Class Testing is used to reduce the number of test cases. Therefore each input parameter of a function is structured into equivalence classes and only one representative test of the class needs to be tested.

A graph-based testing method is basically a hardware testing technique adapted to software testing[1]. Testing begins with creating a graph which includes relations and nodes. Nodes are objects and can have a direct link which represents a relation between objects. Additionally, node-weights can be defined as attributes of the objects. *Cause-Effect Graphing Technique* is a well-known graph based testing technique. A *'-Cause-'* represents a distinct input condition to act as a change in the system. Further, an *'-Effect-'* represents an output condition which results in a system transformation or a state resulting from a combination of causes.

Tests against the limits like *Boundary Value Analysis* is another common testing technique. Therefore extreme boundary values are chosen which include: minimum value - 1, minimum value, minimum value + 1; maximum value - 1, maximum value, maximum value + 1; typical values; error values.

Pairwise Testing and Orthogonal Testing are popular approaches to combinatorial testing problems [22]. Therefore each pair of input parameters to a SuT

[1] http://www.softwaretestinggenius.com, visited: July 2012

tests all possible discrete combinations of those parameters. This technique is based on the observation that most faults are caused by interactions of at most two factors.

The *Classification Tree Testing* method is an approach to partition testing which uses a descriptive tree-like notation. This testing method supports the systematic design of black-box test cases. The input of the SuT is investigated under various aspects which are relevant for the test. Therefore several classes with different aspects are formed. Stepwise partition of the input domain by means of classifications is represented graphically in the form of a tree and the test cases are combined by different classifications [23, 24].

White-box Testing uses the knowledge of a SuT internal structure for the testing process which is also named glass-box testing. Accurate examples for models to be used for testing purposes are Finite State Machine (FSM), formally defined by Petrenko and Yevtushenko [25].

An FSM is a 7-tuple $(S, s_0, X, Y, D_A, \delta, \lambda)$, where

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- X is a finite set of inputs,
- Y is a finite set of outputs,
- $D_A \subseteq S \times X$ is a specification domain,
- $\delta : D_A \to S$ is a transition function,
- $\lambda : D_A \to Y$ is an output function.

The FSM can be used to generate test cases for achieving test coverage [26]. Therefore, a Reachability Analysis is used which is the process of computing the reachable states from the initial states [27]. There exists a set of desirable properties for the testing of FSMs.

This method is used for checking test coverage of the program code to determine unused code branches, data type errors, or a measurement of the code performance. There exists a large variety of coverage metrics. In the following a summary of fundamental metrics are presented.

Statement Coverage - All statements (e.g. if, for, switch) are executed in the SuT. Implicit statements like 'return' are not subject of the statement coverage. This metric is the weakest criterion in the coverage testing family. Beizer [20, p. 75] stated that testing less than this coverage method for new software is unacceptable.

Decision Coverage - This metric reports the Boolean expressions tested in control structures such as while- and if-statements evaluated both false and true. The Federal Aviation Administration (FFA) [28, p. 13] provides safe and efficient aerospace systems and stated that during the test every point of entry and exit in the SuT has been invoked at least once and every decision in the SuT has taken all possible outcomes at least once. The FFA makes a distinction between branch coverage and decision coverage. Branch coverage is weaker than decision coverage. The organization defined a decision as a Boolean expression composed of conditions and zero or more Boolean operators at which a decision without a Boolean operator is a condition. Furthermore, if a condition appears more than once in a decision, each occurrence is a distinct condition.

Condition Coverage - This metric is similar to the decision coverage metric but it has a better sensitivity to the control flow. Condition coverage reports true or false results of each condition. Condition coverage measures the conditions independently of each other.

Path Coverage - This metric executes all possible control flow paths through the SuT. This is the strongest criterion in the coverage testing family and it is generally impossible to achieve because of relationships of data [20, p. 74]. Fundamental path coverage analysis methods are:

- *Branch Coverage* traverses an FSM in order to visit each branch so that the complexity of all possible paths reaching to infinity at worst can be reduced.
- *Switch Coverage* describes a branch-to-branch tuple meaning that in- and out-branches of a state are covered by test sequences [29].
- *Boundary-Interior Coverage* is described by Chow [30] which characterizes test sequences causing loops to be traversed once without additional iterations .
- *H-Language* is a similar approach for Boundary-Interior Coverage loop testing.

The non-descriptive name for Statement Coverage is C1, for Decision Coverage it is C2, and other non-descriptive name for Path Coverage is C∞, see [20, p. 75]. Those coverage analysis methods are powerful testing tools for checking the correctness of the control structure at the design level of many software systems [30]. Application examples of the coverage metrics are presented in [20, 28] and Bullseye Testing Technology [2].

[2] http://www.bullseye.com/coverage.html, visited: July 2012

Gray-box Testing is a software testing method which combines the Black-box testing method and the White-box testing method [31, p. 27]. In the Black-Box testing the internal structure of the item under test is unknown to the tester and in the White-Box testing the internal structure is known. By using the Gray-Box testing method the internal structure is partially known. This involves having access to the internal data structure and algorithm for designing the test cases.

A similarity of the Black-box and White-box testing is that the developer will share the ignorance about the internal structure of the SuT at the beginning of the testing because the Gray-box tests will be written before the SuT is implemented. This means that Gray-box testing is usable for using the Test-First Development (TFD) method which is explained late in Section 2.4.2, see Figure 2.8.

A similarity of Gray-box and White-box tests is that the tests are written by the same development engineer as the SuT.

Summarized, Gray-box tests are not a full alternative to Black-box tests. It can be seen as a qualitative improvement of White-box tests.

The question is: "Do we need all these different testing techniques [31, p. 27]"? Yes, because the White-box testing checks the code for its correct functionality. The Black-box ensures that the requirements have been realized. The Gray-box test is used for integration testing of a component-based architecture. Therefore the source-codes as well as the monolithic parts are used for the tests [31, p. 27].

Those fundamental terms are their understanding is required for further steps in order for developing testing concepts.

1.3.3 Testing vs. Verification

A different approach for ensuring software quality and its correctness is formal verification, e.g. model checking [32,33]. Model checking is an automatic technique for verifying concurrent finite state systems and was first used for verifying hardware components. Then it has been used for software component verification, e.g. verifying complex sequential circuit designs, communication protocols. Formal verification of industrial automation systems needs three components to verify correctness and to detect a conflict [34]: (1) A model of the controller, (2) a model of the uncontrolled plant, and (3) a specification of the plant behavior.

In this approach control software is represented in form of formal mathematical models. This one analyzes and checks if all requirements are fulfilled at all times. Thus the correct behavior of the verified component can be certified. In contrast to the methods of formal verification, a formal model of the application is not required for testing [35].

A major drawback of the verification process for software components is the state-space-explosion-problem [36]. This problem occurs in systems with many components which interact with each other. This means that the system has a strong coupled behavior, i.e. dense matrix. Furthermore, systems with data structures consist of many different values which are a big effort for the system verification. Many realistic systems are still too large to be handled in a verification process [33, p. 10].

Kormann et al. [5] presented an automatic test case generation approach based on model checking models for control software. Loeis et al. [37] and Pollmächer et al. [38] presented a formal modeling approach for IEC 61131 PLC programs to be used for verification by model checking. At the moment, the test case generation needs too much time which is therefore less applicable for testing of larger control systems.

Hurnaus and Prähofer [39] presented a combination of model checking and artificial intelligence techniques to guide domain experts in building control software. This software can be used to create a state and knowledge deduction process to allow deriving knowledge at a code position of the control code which can be used to verify contracts and constraints. The programming language is based on MONACO, a domain-specific language for building control solutions.

Several approaches for performing formal verification on IEC 61499 applications have been presented in [40–42].

A critical part of executing control applications is the execution environment as well as the control hardware. Gerber and Hanisch [43] showed that the execution environment can change the execution behavior and added it to their verification models. However, Sünder and Vyatkin [44] showed that a full mathematical model of the whole execution environment and the control hardware leads to highly complex models, i.e. inefficient calculation effort. In order to check also the components' behavior on real control devices, using software testing techniques can complement model checking methods for improving software quality. Specified models for the model checking process as presented in [45] can be used for the test specification, i.e. the components' intended behavior. In this respect, software testing can support model checking as test cases which are executed on the real control devices but this thesis will not focus on software verification processes.

Summarizing, the effort for verifying a control application is very high because of the changing requirements on short notice. This influences a change of the verification model. Hussain and Frey [46] state that the models for the verification process will require additional effort for building up the formal model as well as huge computational power while realizing it.

1.4 Summary

In this section an introduction and motivation about software testing for industrial automation systems is given. The problem statement of testing control systems is pointed out and the goal of this thesis is defined.

A basic overview on software testing methods is presented. Fundamental terms are defined and basic testing methods from the software testing domain are introduced. An overview on testing versus verification is presented.

1.5 Contributions and Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2 presents an introduction of the common industrial automation programming languages IEC 61131 and IEC 61499. Related work on software testing methods are investigated, such as testing control software of IEC 61131 and IEC 61499 applications. Common methods and the state of the art of testing in the software engineering domain, software testing techniques in the embedded systems domain are presented, and also a comparison of the different testing domains is discussed. Furthermore, work on reducing the number of test cases as well as the economical aspects of testing in general are discussed.

In Chapter 3 a new test framework for systematic automated testing of industrial automation applications is proposed. This test framework supports testing on various levels in detail and from different perspectives. In the second part of this chapter individual aspects for a testing process is shown and how these aspects address various testing levels is presented.

Model-based specification techniques are useful to be flexible of changing requirements. In Chapter 4 a selection of appropriate Unified Modeling Language (UML) models for Test-Driven Development (TDD) is presented and evaluated for the use in industrial applications. Furthermore a test case generation approach which extracts test case information from UML state chart diagrams is presented.

In Chapter 5 four testing techniques are presented which are able to test industrial automation software. The manual testing method, the keyword-driven testing method, the unit testing method by using service sequence diagrams, and the model-based testing method are presented and explained.

Chapter 6 demonstrates the benefits and drawbacks of the proposed testing techniques from Chapter 5. The testing methods are explained based on the elected prototypical implementations. Finally, a comparison of the testing methods is shown.

Chapter 7 presents a new automation component approach for a more test friendly development of industrial automation applications. The proposed component structure enables a reduction of valuable development time

because the components are designed for reusability in different industrial project applications.

The thesis is concluded with a discussion about the proposed approaches in Chapter 8. Answers to all derived research questions that are defined in Section 2.9 are given. The successful implementations of the proposed testing techniques applied on different laboratory constructions is summarized. Finally, an outlook on future research to ensure the quality of industrial automation software is presented.

CHAPTER 2

Related Work

Testing is defined as the execution of a SuT with the aim to find faults in the system and verifies the behavior of the SuT towards a set of requirements that are expressed as tests [11, p. 1]. Software should be without defects (zero defects) and should guarantee correctness with construction implementations of a complete specification. Complete in this case means that the full functionality is considered and no constraint is left out [47, p. 6]. Nowadays industrial control applications are mostly tested manually which results in limited Quality Assurance (QA) and the effort of such test process is enormous. Control software for industrial plants are not tested within a systematic testing process which results in an unmanageable task.

The history[3] of software testing started in the 1950s. The software engineering domain had the same challenges some decades ago as the industrial automation domain has today. Currently several testing methods and techniques are available. Some testing techniques were adopted from the software engineering domain into the embedded systems domain. But the testing requirements for industrial automation needs new approaches to ensure a high software quality [48].

There is no universal definition of *software quality*, it is highly context dependent [49]. To improve software quality, a definition of aspects of quality that are of interest for the developer or tester has to be done. In the following, six aspects are considered to ensure software quality based on Kitchenham and Pfleeger [49], International Organization for Standardization (ISO) 9126 [50], and [47, p. 81]:

[3] History of software testing, an overview:
http://www.testingreferences.com/testinghistory.php visited: September 2012

- Functionality - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
- Reliability - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
- Usability (Human Engineering) - A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.
- Efficiency - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used under stated conditions.
- Portability - A set of attributes that bear on the ability of software to be transferred from one environment to another.
- Maintainability - A set of attributes that bear on the effort needed to make specified modifications.

The process to measure the software quality starts when the main functionality of the software is achieved. After that the aspects of reliability, usability, efficiency, and portability can be considered. Böhm et al. [51] presents an evaluation of quality metrics to measure the software quality based on those aspects. Indicators for measuring software quality are structuredness, robustness, and self-descriptiveness. The resulting taxonomy of software quality characteristics based on [51] is presented in Figure 2.1. In general, there can be asked three questions which bring the requirements to the point [51]:

- How well can I use the system? - reliability, easily, efficiently
- How easy is the maintenance? - modify, understandability, re-test / testability
- Is it possible to change my environment? - portability

Figure 2.1 shows that portability, utility, and maintainability are necessary conditions for the software quality characteristics. The utility requires a component which is reliable, efficient, and useable but it does not require the user to test the software component. Maintainability requires the user be able to understand, modify, and test the SuT [51].

In the following an introduction of the common used modeling languages, the IEC 61131 and the IEC 61499 standard is presented. Further, existing concepts and existing work on software testing methods are investigated. First

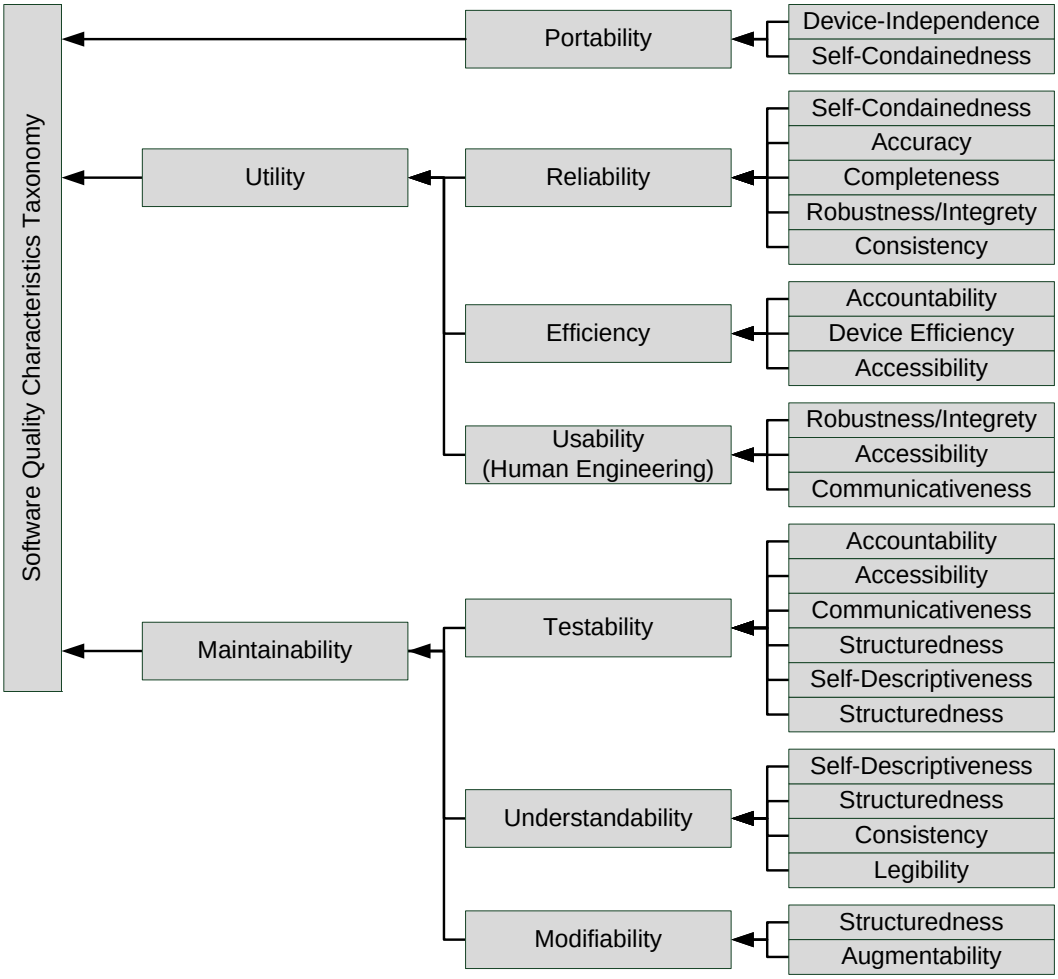


Figure 2.1: Software quality characteristics taxonomy based on Böhm et al. [51].

testing control software of IEC 61131 and IEC 61499 applications are discussed, followed by common methods of testing in the software engineering domain and finally software testing techniques in the embedded systems domain are presented. Then a comparison of the different testing domains is discussed. Furthermore, aspects on reducing the number of test cases is discussed as well as the economical aspects of testing are presented.

2.1 Modeling Languages for Industrial Automation Systems

In this section an introduction on the common industrial automation programming languages IEC 61131 and IEC 61499 is given which are mostly used to model the desired behavior of applications in the field of automatic control. Industrial automation is applied to control and optimize production processes. It provides high quality and reliable products, for example in the following domains [47, p. 35]: Machine and plant control, chemical process control, distributed production control, traffic control. Therefore safety critical behavior and real-time constrains should be observed.

In order to achieve a high software quality of such control applications, the software testing effort must be effective and efficient to be competitive in the ever increasing market situation. For increasing the testing effort of the IEC 61131 applications and the IEC 61499 applications, the fundamental structure of the those modeling languages are presented. An explanation and definition of a model is presented in Section 2.3.

2.1.1 IEC 61131

The International Electrotechnical Commission (IEC) provides the International Standard IEC 61131 [52,53] which consists of a set of eight parts concerning PLC. Part 3 of this standard is the currently most important one as it defines programming languages, data types, and the software architecture used by most PLCs vendors.

The main elements of the IEC 61131-3 software model are shown in Figure 2.2. The top element of the software model is the CONFIGURATION element and represents a PLC. The CONFIGURATION contains one or more resource elements. Each of such RESOURCE contains one or more programs executed under the control of zero or more tasks. A TASK unit is an execution control element to schedule the execution of associated Program Organization Units (POUs), e.g. periodically or event-triggered execution. Further, a PROGRAM may contain zero or more Function Blocks (FBs) or other language elements as defined in this part. Additionally, IEC 61131 supports *functions*. The difference between FBs and functions is that FBs contain internal state information. The association between these elements is described by the execution control path which is pictured by the dashed lines in Figure 2.2.

Data types: The IEC 61131-3 defines a set of elementary (pre-defined) data types, e.g. integer, real, string, Boolean, real numbers. Additionally, generic data types are defined for using such pre-defined data types to overload functions. A mechanism for specifying additional data types is

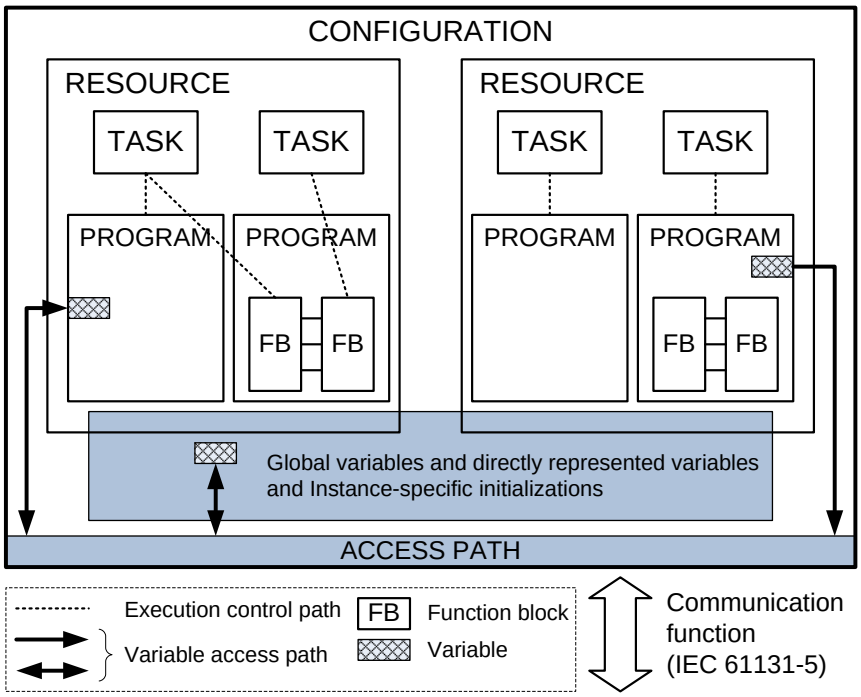


Figure 2.2: Overview of the IEC 61131-3 software model based on [52].

also defined. Elementary data types, a keyword for each data type, the number of bits per data element, and a range of values for each elementary data type are shown in [52, Table 10] and [54].

Variables are data objects whose content may change, e.g. data associated with the inputs, outputs, or memory of the PLC. A variable can be declared to be one of the elementary data types or a derived data type. There are several variants of variables which have an important influence on the PLC program. In the following a full overview of the declared variable keywords is presented which is documented in [52, Table 16a and 16b].

- VAR and VAR_TEMP identify internal variables, the latter can be used as temporary variable.
- VAR_INPUT, VAR_OUTPUT, and VAR_IN_OUT defines the interface of a POU.
- VAR_GLOBAL defines variables which can be used within a whole scope of the configuration. This means that variable values can communicate between different programs which are in the same configuration. Access definition of the variable is allowed by the

use of READ_WRITE or READ_ONLY commands. Variables defined by the CONSTANT command are automatically defined as READ_ONLY.

- VAR_EXTERNAL constructs can be used to access a POU to a globally defined variable.
- VAR_ACCESS defines variables which can be used for remote access via the communication service.
- VAR_CONFIG parameter is used to initial different values for different instances.

Modeling languages: IEC 61131-3 defines five modeling languages:

- Instruction List (IL) is a line-oriented textual language. Each line includes a command and an operator which looks like an assembler language.
- Structured Text (ST) is similar to a high level programming language such as Pascal but with limited functionality.
- Ladder Diagram (LD) is a graphical language and comes from relay ladder logic diagrams and provides a similar visual representation.
- The Function Block Diagram (FBD) language is used to describe a function between input variables and output variables. A function is described as a set of elementary blocks which can be connected to blocks by input and output variables. Such connections are represented by connection lines.
- The Sequential Function Chart (SFC) elements are defined for structuring the internal organization of the programmable controller, i.e. programs and FBs.

Applications based on IEC 61131 for the automation domain are written in one of these defined programming languages, see [52, Chapter 2.6].

A general difference of programming languages from the computer sciences, such as C++, Java, is their execution behavior. The execution element is a task element which can trigger POUs cyclic or event-based. Hence, the POUs are executed in the context of these task executions. A cyclic triggered execution is typically for a IEC 61131-3 application.

2.1.2 IEC 61499 - Distributed Event-based System

The IEC provides the International Standard IEC 61499 [55, 56] for Industrial Process Measurement and Control Systems (IPMCS) defining a FB-oriented

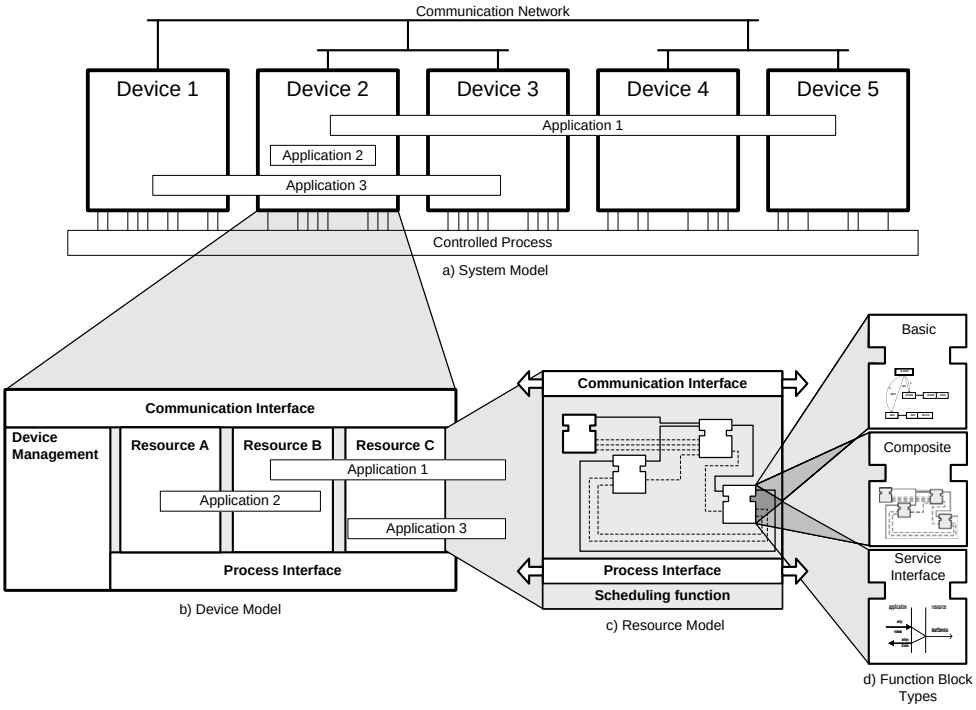


Figure 2.3: Reference models according to IEC 61499 [58].

paradigm for distributed control systems development. The standard includes several models which are presented in Figure 2.3:
System model, device model, resource model, application model, and the FB model. These models allow developing control applications by using graphical development methods. A FB, the basis model of the IEC 61499 family, is a software component which is self-contained and provides its functionality through a determined interface.

There is the possibility to build simple or sophisticated solutions by using FBs or parts of software. Each FB contains particular algorithms to configure a solution without the need of programming and developing the application from scratch [57]. The model interface has been expanded from the IEC 61131-3 interface definition and consists of two parts: The event- and the data-interface which are shown in Figure 2.4. The data-interface consists of data input and outputs which are already presented in the FB interface definition in IEC 61131-3. The interface is extended with an additional event interface. The execution and processing of the internal FB algorithms are started if a trigger event to the event input is sent. During the execution process of the FB, the input data will be processed and the output data will be generated. Additionally, an event output can be triggered.

The IEC 61499 standard is organized hierarchically ranging from the system model (top-most model layer) to the FB model (lowest model layer).

The System model forms the highest abstraction layer and specifies the relation between applications and devices. Applications can be distributed over several devices and are not restricted to a single device, e.g. application 1, 2, and 3 over the devices 1 to 5 in Figure 2.3.

The Device model is composed of one or several resources to contain FBs and enable their independent execution. The device model includes an interface for communication services as well as a process interface to communicate with input or output ports of the physical device.

The Resource model contains a FB network and enables the execution by providing a scheduling function for their correct execution sequence of algorithms. It also provides a communication interface as well as a process interface.

The Application model consists of a FB network including event- and data-connections. An application can be distributed over several devices and resources.

The Function block model is defined as a functional unit of software parts containing data and algorithms. It defines the design and the characteristic of the FB (event- and data I/Os), as seen in Figure 2.4. Multiple instances of a FB can create and be executed independently from each other.

IEC 61499 supports different FB types: The Basic Function Block (BFB) type, the Composite Function Block (CFB) type, and the Service Interface Function Block (SIFB) type. The FB types describe the behavior and interface of instantiated FBs.

- A BFB includes internal algorithms which are executed according to the defined Execution Control Chart (ECC) in the FB. Only the BFB includes an ECC which forms an internal state machine and maps input events to algorithms for processing and creates output events at the occurrence of state changes.
- The CFB encapsulates a network of FBs and their corresponding event- and data- connections.
- SIFBs provide the necessary interface to communicate between FBs of different networks, e.g. remote resource, or are used for direct hardware access, e.g. hardware I/O elements.

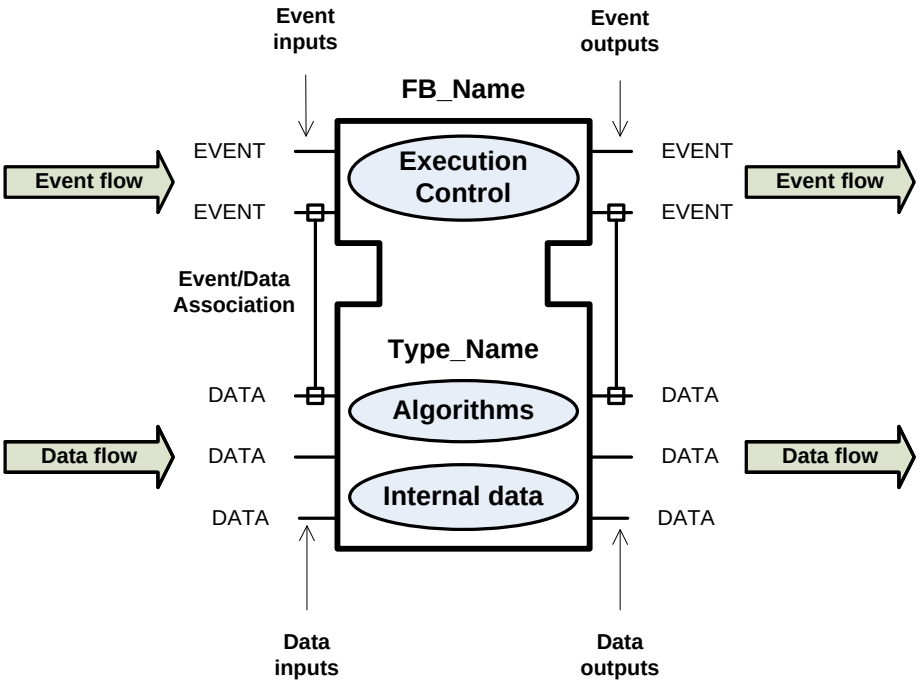


Figure 2.4: Characteristic and interface design of an IEC 61499 FB.

2.2 Testing Control Software

Industrial manufacturing and industrial automation system grow in complexity and there is the need for control engineering support by testing. Several years ago Seitz et al. [59] proposed that a complete testing of automatic control systems including the documentation over the full development life cycle is required.

Software testing is well established in the field of software engineering and computer science (see Section 2.4). But the systematic and automated testing of software modules and components in the automatic control domain is still at an early stage. The reason is that the constraints of the general requirements of industrial systems are much harder than in the field of classical software development. Such general requirements are real-time conditions, interaction with 'the real world' by sensors and actuators or distributed control systems.

Testing of PLCs software is a time consuming and expensive item. In the moment, there is no standard for a test process of industrial control software, but with the introduction of the IEC 61508/61511 [60, 61], a similar approach to the safety-critical processes will be extended in the manufacturing industry. This section gives an overview of the related work and the state-of-the-art in the field of testing control software.

2.2.1 Testing as Commissioning Support

Auinger et al. [62] and Schludermann et al. [63] proposed an approach to test industrial control software by connecting a PLC to a commercial Discrete Event Simulator (DES). This approach is called Soft-Commissioning which is a Hardware-in-the-Loop (HiL)-based approach. The provided system reacts similar to the behavior of real hardware like an industrial manufacturing line. The methodology of HiL is that the inputs and outputs of a controller are connected to a simulator of the controlled part. The interaction of the various parts can be tested easier and a reproduction of the test conditions are given. The biggest difference between Soft-Commissioning and other HiL systems are the essential system reaction times. Soft-Commissioning is provided to work with round trip times smaller than 100ms. While most HiL systems use fast Digital Signal Processorss (DSPs), Soft-Commissioning is executed on standard PC workstations. Furthermore, Soft-Commissioning interacts with DES while other HiL systems are based on continuous real-time simulations [63]. The infrastructure communication between PLC and simulator is a first step for testing the behavior of the control application. This approach is very extensive and the simulator requires an appropriate specified simulation definition. A systematic testing of the control software is not adequate with this approach.

A virtual commissioning process is proposed by Kabitzsch et al. [64]. They used a combination of simulation, testing, and monitoring techniques for commissioning industrial manufacturing systems. For all fields of applications, i.e. simulation, testing, and monitoring, different tools are used. The goal of the project named OMSIS[4] (On-the-fly-Migration und Sofort Inbetriebnahme von automatisierten Systemen) was to develop a consistent integration of the three application fields. In the case of testing, a test system manager named *ECU-Test* by TraceTronic[5] is available. This test manager is used in the automotive domain. There is only a tool adapter between Siemens Simatic-S7 and ECU-Test available to stop and restart the PLC, read and write selected signals [64].

2.2.2 Automatic Testing

Manual tests may find many defects in a software application but it is an exhausting and time consuming process. Automatic test automation is the process of defining an instruction to do testing that would otherwise need to be done manually. The big advantage is that once tests have been automated they can be run quickly and repeatedly. During the lifespan of the application features may break which were working at an earlier point in time. Automatic testing is often the most cost effective method for testing software that have a long maintenance life.

[4] http://iai8292.inf.tu-dresden.de/omsis/de/index.html visited: August 2012
[5] http://www.tracetronic.de/ visited: August 2012

A universal architecture for efficient test automation is proposed by Korotkiy and Bender [65]. The focus of this architecture approach is based on testing mechatronic products. This universal test system architecture can be used as a basis for the realization of test systems and can be extended for an architecture of testing control applications for improving the overall efficiency of test automation. The following definitions are discussed: systematization available tools, requirement analysis, tool selection, concept and realization, and final evaluation. An implementation of this approach is not presented.

An application example of automatic daily-build tests for PLC programs modeled in IEC 61131-3 are presented by Stetter and Erben [16]. A Concurrent Versions System (CVS), a built system, and a target system, i.e. PLC, are necessary. Every day the developers have to check-in the source code into the CVS. Note that the source code must be executable on the target system. The built system compiles the source code into an executable program. Automatic testing in combination with a CVS is most important for testing libraries. That means that the approach from Stetter and Erben [16] is useful for testing units or components but this approach is not applicable for testing system applications.

Hussain and Frey [46] presented a UML-based development process with an automatic test case generation approach. First, several UML diagrams are used for the specification of the functional behavior, i.e. state chart diagram, component diagram, use case diagram, sequence diagram, and activity diagram. For the testing process, the white box testing method is used under the consideration of the state chart diagram and the activity diagram specification. Further, the round-trip path coverage is applied on the SuT. The coverage of at least every defined sequence of specified transitions that begin and end in the same state is considered. This approach is not applicable for testing the SuT against the specification because of using the white-box testing method and the specification for the implementation is the same as the test specification. Hence, only the transformation process is being tested and not the correctness of the control software.

Hussain and Eschbach [66] proposed an automated fault tree generation approach which is used for verification, i.e. model checking, and testing of control systems. Therefore distinct modeling artifacts produced during the earlier stages of the development life-cycle of the system are used to convey results to drive the activities of later phases. After the analysis a fault tree is generated. This is used to drive the design and development decisions as well as the testing activities. A combination of verification and testing methods are used. In order to compose formal models of the system automatically, distinct modeling artifacts and information are used. Embedded hardware and network failures are checked by model checking methods. Through model checking, counterexamples can be found to satisfy particular properties which can be used as test cases, e.g. testing a redundant system. This is an example

that testing and verification methods complement one another.

2.2.3 Simulation-based Testing

Traditionally simulations are used to help understand the behavior and performance of complex systems [67]. Simulation however can also be used to support testing.

Seitz et al. [59] describe a test process in which binary input-output (I/O) signals are connected to a simulator which includes a simulation model (model of the real system). The simulator automatically executes the test sequences to the PLC by generating test signals through a plant simulation. The feedback is controlled by the simulator as well. The test results are automatically recorded and maintained, so that every time the test process can be repeated. This is used to check the wire connection of patch panels and their hardware configuration in a test rig. Further, the behavior of the control software is tested by external I/O test signals. Summarizing, it is useful for testing the hardware configuration of PLCs and further the final implementation of the control software can be tested.

Another simulation-based testing approach is presented by Greifeneder et al. [68]. A simulation tool is presented which is able to generate a simulation model on the basis of process control engineering data. Based on the semi-automatically generated simulation model, the I/O behavior of the relevant sensors and actuators as well as the physical process can be tested earlier in the development phase.

Simulation-based testing is quite close to the commissioning phase and is not applicable for testing in an early stage of the development process, i.e. starting development of control code, but it is a powerful method for checking the system in a later or closed to the final development phase.

2.2.4 Analyzing Methods

An approach for checking the software behavior of control applications is the debugging technique. A comprehensive offline debugging solution for Soft-PLCs based on IEC 61131 is presented by Prähofer et al. [69, 70]. All external influences and sources of non-deterministic behavior in the original execution run are recorded which can later be replayed deterministically without requiring any connection to the original environment and any hard real-time constraints. This capture-replay technology approach is used for recording the reactive behavior as well as for detailed analysis of a program run. A visualization and exploration of the reactive behavior of PLC programs is presented by Wirth et al. [71]. Such analyzing methods for industrial software can be

combined with new testing methods to improve the quality of industrial system solutions.

The methods which are used with classical software engineering are not directly applicable for testing control software. Therefore a special development process for testing automatic control applications is needed.

Systematic testing processes including testing control software is uncommon in the domain of industrial automation. Even in a recent state of the art review [7] no evidence of testing can be found.

2.3 Model-Driven Development

In order to manage development of complex software systems, new development processes for the software development are concerned [72, p. 21]. The purpose of Model-Driven Software Development (MDSD) is to close the gap between:

- The problem domain, i.e. user requirements,
- the solution domain, i.e. system requirements, design specification, and design analysis,
- and the realization domain, i.e. system development, implementation, and their analysis [73].

Nowadays software development processes are based on the Model-Driven Architecture (MDA) standard. MDA is an open standard which is propagated by the Object Management Group (OMG)[6]. The main goals of MDA are interoperability, portability, and reusability through architectural separation of concerns. A Platform Independent Model (PIM) represents functional requirements in a formal notation and is used as the basis for one or more Platform Specific Models (PSMs). Each PSM is an abstract representation of one aspect of the implementation. For creating an implementation one or more transformation steps from the PSM are progressed [11, p. 22].

Additionally, standards are used with MDA, such as Meta Object Facility (MOF), UML, Common Warehouse Metamodel (CWM) [72, p. 14].

2.3.1 What is a Model?

> "A model is a description of something." [74]

[6]Object Management Group (OMG), Model-Driven Architecture (MDA), http://www.omg.org/mda/ visited: August 2012

In the following, the term *model* is understood as *software model*. The software engineering domain traditionally defines a model referred to be an artifact that is formulated in a modeling language, such as UML. There are different kinds of models, but not all model types are suitable for Model-Driven Development (MDD) in the field of software engineering. Further, not every model is a formal model, e.g. semi-formal model. A *formal model* is able to describe an aspect of the software in a complete and unambiguous manner for instance in a mathematical manner. A formal model has a distinct defined grammar which includes its structure, e.g. syntax, as well as its meaning, e.g. semantic. These requirements imply that a single model does not have to and will not be able to describe the whole system. But it has to be clear what is included and what is not included. Note that formal models must not necessarily be defined by UML models [75, p. 11]. UML[7] describes a system through the help of various diagram types. In general, such software model descriptions are graph-based and typically rendered visually. Several definitions of the term model, i.e. software model, are explained by Kühne [74].

According to [76, p. 131-133], a model can be defined by three features:

- Mapping feature: A model is based on an original object and represents a natural or imitation original which can be a model again.
- Reduction feature: A model reflects only a relevant selection of the original's properties.
- Pragmatic feature: A model needs to be usable in place of the original with respect to some purpose, e.g. for special objects (who?), during a special time interval (when?), under special conditions (why?).

2.3.2 Model Specification

An important issue of modeling a system is to define a good level of abstraction. Hence, it has to be decided which aspect of the SuT should be included or omitted in the model [77, p. 60].

The ideal model specification should be easy to understand for testers. Further, a large problem should be described as easily as a small system and the model specification should be understood by a test generation tool. There is no ideal modeling language for all purposes which include several notations. Ideally the model can be generated from some representation of the requirements [78].

If the model is used only for test case generation purposes, it does not have to specify the whole behavior of the system. Because the specification of test

[7] in the further UML and UML 2 is used as a synonym

cases must not be complete such as a behavior description of a system which is used for code generation.

Over the years several development processes have arised, for example UML or the simultaneous process model Rational Unified Process (RUP) for software development. RUP is a commercial product of Rational Software which is a part of the IBM group. RUP uses the notation language of UML. In recent years, UML has been very useful for the description of software, e.g. starting from the technical analysis to the technical design [79,80]. The advantages are clarity, comprehensibility, intensity of expression, the standardization and acceptance, platform and language independence, and independence of process models. They all are essential aids in the presentation of interim results in a development process.

A practical example is to specify the requirements of a model by setting all possible values, i.e. valid and invalid values, of a parameter and specifies a test generation model for the testing process. The model includes the specific constraints among the specified values. These constraints include semantic information about the relationship between parameters, e.g. two parameters which cannot be empty (null) at the same time. An overview of existing specification tools is presented by Micskei[8].

Utting et al. [77, p. 62ff.] present several modeling notations for modeling the functional behavior of systems. Pre/post (or state-based) notations and the transition-based notations are the most popular. Pre/Post notation represents a snapshot of the internal state of the system as a collection of variables. Therefore each operation is defined by a precondition and a postcondition. The transition-based notation describes the transition between different states of the system, such as FSMs. The nodes of a FSM represent the major states of the system and the arcs represent the action of the system. Other notations are: History-based notations, functional notations, operational notations, statistical notations, and data-flow notations. The selection of the most practical notation depends on the application whether it is more data-oriented or control oriented.

After the model specification, the parameter has to be checked towards incorrect data ranges which leads to test failures and therefore must be corrected.

2.3.3 Model Transformation

The transformation is the connector between the model and the execution platform. Hence a formal model or a semi-formal model is transformed into something else. That can be a program code which is called *Model-to-Code (M2C) Transformation* or the source model is transformed into another model which is

[8] http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html visited: August 2012

called *Model-to-Model (M2M) Transformation* or simply Model-Transformation [75, p. 33].

- M2C Transformation / Model-to-Text (M2T) Transformation: M2C transformation produces, based on the model definition, a source code which is called code generation. This kind of transformation is needed for generating dependent artifacts from the model, i.e. source code of a programming language like C++, C#, Java. It is possible to define several different M2C transformations based on the same (semi)-formal model. The results, i.e. generated code, can be used for different target platforms such as different PLC platforms or different implementation standards such as IEC 61131 or IEC 61499.
- M2M Transformation: M2M transformation is used to transform one or several models into a new target model. This is needed if a model is modified, extended, or used in different engineering domains.

Further information about model transformation is presented in [75, p. 33]. Model transformation applied in the field of industrial control software is shown in [81–84]. The use of such model transformation methods are applicable to generate test cases from the (semi)-formal test case specification automatically, see Section 5.4.

2.3.4 Model-Driven Development for Control Applications

MDD is widely used and accepted in the software engineering domain. In the last years this development process has become more important in the field of industrial control software development. The idea of this approach is that models rather than software code are the primary artifacts of the software development. In order to use MDD the complexity of industrial control applications can be handled easier by using an abstract view of the system's issues to solve industrial challenges.

Frey and Thramboulidis [85] state that the FB language can be seen as a MDD approach in the automation domain. The abstraction level of the IEC 61131-3 application modeling is very low and close to the implementation level. The IEC 61499 is used for modeling because the level of the modeling is more abstract from the implementation. This can be seen in the explicit specification of the control execution, i.e. ECC included in the BFB. Instead of the direct hardware access by global variables in IEC 61131-3, the hardware access in IEC 61499 is realized by a SIFB. Further, UML and Systems Modeling Language (SysML) are considered as modeling language for the industrial control domain. Frey and Thramboulidis [85, 86] state that the best condition for an integrated MDD is provided by SysML.

Panjaitan and Frey [73] propose a development process for Distributed Control Systems (DCSs). In the development process UML is used in the first stage and IEC 61499 in the later ones. Several design patterns for implementing controller based functionality of mechatronic objects are presented [73].

A further combination of UML modeling and IEC 61499 FB network concept in the field of DCSs is shown in [87].

The translation of UML artifacts to FB environments based on UML class diagrams is proposed in [88]. This development approach supports the collaboration between designers from automation engineering and software engineers.

Hussain and Frey [89] use an IEC 61499 compliance profile by using UML combination with Object Constraint Language (OCL). Therefore, the OCL is used to add additional constraint information into the modeling elements (e.g. initial values, derived values) which helps with the mapping and analyzing process.

Vogel-Heuser et al. [90] use an object-oriented approach by using UML as modeling notation integrated into an IEC 61131-3 PLC programming environment. The focus lies on automation software development for the machine and plant manufacturing industry. There is actually no support for testing, the derivation of test cases and the automation of tests based on the structural and behavioral description of the UML model in the development phase.

Thramboulidis [91] proposes an architecture for the development of industrial manufacturing systems that promotes model integration not only for the implementation artifacts but also for artifacts of the early analysis and design phase of the development process. The model integrated development process applies a domain specific modeling language for concurrent engineering of mechanical, electronic, and software components of mechatronic systems.

MDD for industrial control applications is the right way to solve future challenges in this domain. Techniques of this development method can also used for testing control software which is presented in Section 5.4.

2.4 Testing in Software Engineering

The rising functionality of software based functions requires a significant increase of test activities. Concurrently, because of the short time of development, new challenges arise for the software developers. The quality of the software has to be constant or has to be increasing by constant expense. This increase in efficiency can be realized by testing. The goal is to reduce the necessary test expense of the test process by reusability, automation, and optimization. Software testing in the software engineering domain has been known to consume about 50% of the development costs and spans about 50% of the development time [92].

Testing is an experimental process with two objectives: a) detect as many errors within the SuT (destructive testing) and b) demonstrate the correctness of the SuT (demonstrative testing)

The ISO 9126 [50] addresses software quality and defines how quality can be measured. Reliability, performance, and security are quality targets which are confirmed by testing.

2.4.1 Test Specification in Software Engineering

The first step of each entire development project is the requirement analysis. The test specification for a component is not only used for the test process. It also acts as a design and development guide to support the developer in better understanding of the implementation problems [93].

There are four methods for system specification which can be used similarly for test specifications [94, p. 22] and [95]:

- *formal*: An artificial language like Z or OCL is used and is propagated by the OMG. This method has their strength in their ability to rigorously define their requirements.
- *semi-formal*: Norm tables, drawing, and diagrams are used for the specification. This method is mostly used nowadays because semi-formal specification is easier to understand and is more human-nature oriented.
- *structured*: Structured analysis has been mostly used in the 1980s. This method is a combination of structured programs, data trees, and data flow diagram, additional to tables and textual descriptions.
- *informal*: The description is only specified in a textual form and is not applicable because it is hard to understand for test engineers.

There are many approaches to specify tests in the field of software engineering. Dalal et al. [78] state that there is no ideal modeling language for all purposes. Several notations are required and ideally the data model can be generated from some representation of the requirements. An actual specification method is semi-formal by using the combination of UML and OCL. The preferred approach for system requirements is a combination of diagrams, tables, list of terms, and a formal description. Further information about combination of (semi)-formal models is presented in Chapter 4.

UML 2.0 [96], as the dominant software modeling language, is a potential language for describing and specifying functional and test behavior of complex software systems. The UML state diagram, which is part of the UML diagram family, is mostly used for structural test processes [97].

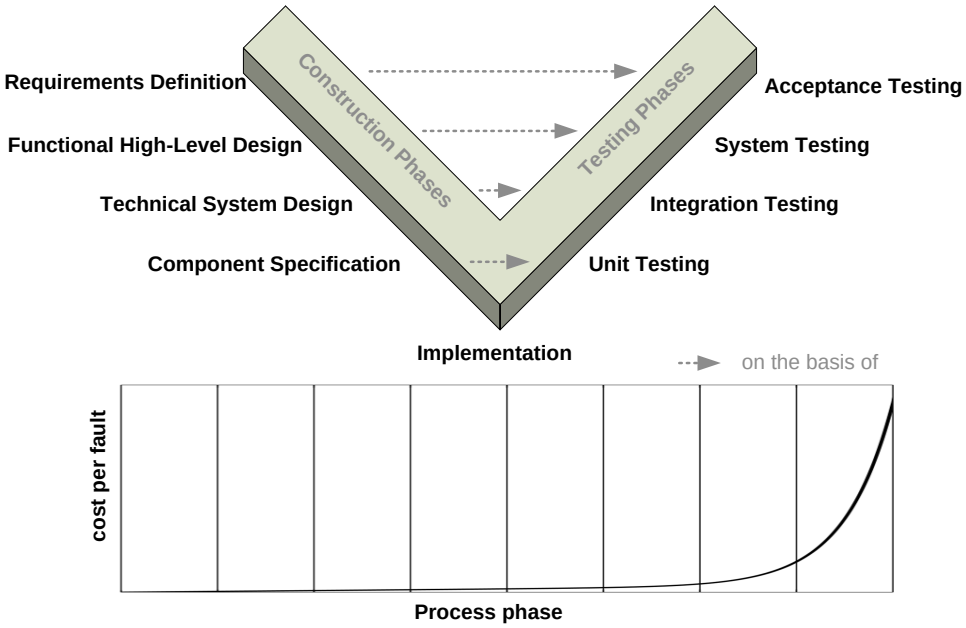


Figure 2.5: Definition of the V-Model XT and dependency between time to find a fault and the resulting costs.

Swain et al. [98] use the combination of UML state chart diagrams and activity models named state-activity-diagram for the test case specification and generation because the combination provides control flow and event-oriented state change information.

A test specification can be derived from business concepts, business documents, and predefined derivation rules to build test diagrams. A combination of several specification parts can be built into a test suite which includes all test cases [99].

The test specification can also consist of failures which can be checked. Chow [30] describes a method for checking the correctness of control structures at the design level by FSMs. Therefore, state chart diagrams and the automata theory are used for the software testing process.

2.4.2 Test Processes, Test Strategies, and Test Levels

Software processes define sequences of steps along the project life-cycle and focus on specific requirements of the application context. Such software processes are necessary to guide development engineers and testing engineers during a project [100].

There is a wide range of flexible software processes, for example: The waterfall model, the spiral model, the unified process, the V-model, the W-model, Scrum, and TFD [31, 101].

The standard IEC 29119 [102] defines vocabulary, processes, documentation, techniques and a process assessment model for software testing which can be used within any software development life-cycle. In the future this standard will replace several existing standards like: IEEE 829 Test Documentation, IEEE 1008 Unit Testing, BS 7925-1 Vocabulary of Terms in Software Testing, and BS 7925-2 Software Component Testing Standard.

The most effective possibility to reduce the development time is to avoid specification and implementation faults. The investment can be reduced if abnormalities of a specification and implementation are found early, which is shown in Figure 2.5. For example, if the specification of a software part is not unambiguous, the development engineer implement the wrong behavior of the system. In order to correct this fault, additional costs and additional time is used to get finally the high quality product. To find such faults, the test procedure must be organized in different test levels. Each test level addresses a specific group of requirements, functional, or technical specifications.

2.4.2.1 Development Processes

V-Model [103] was defined in the year 1992. The last update was in 1997 and currently it is not maintained. Indeed, it has had a defined position in the field of software development for business IT development. A significant benefit of this approach is the definition of the communication between the stakeholders and the development engineers to avoid misunderstandings. It becomes clear that the ordering of activities in time sequence and with abstraction levels between the development and test activities. For example, the acceptance test is carried out on the basis of the results of the requirement definition phase [101]. As an important extension of the V-Model, the V-Model XT[9] was developed, see Figure 2.5, which has been a mandatory process model for IT projects for the development of software intensive systems since 2005.

V-Model XT defines several levels and phases in the development and testing process. The levels of the V-Model XT are named based on [21, S. 8]. The requirements definition, functional high level design, technical system design, component specification, and implementation phase are defined in the descending branch (shown on the left hand side in Figure 2.5) of the V-Model XT. The ascending branch shown on the right hand side in Figure 2.5 in the V-Model XT represents the testing phases in which the testing levels are executed towards the requirements to prove the software to be working comply with the specification. Various tests such as unit testing, integration and regression testing, system testing, and acceptance testing are defined.

[9] http://www.v-modell-xt.de visited: August 2012

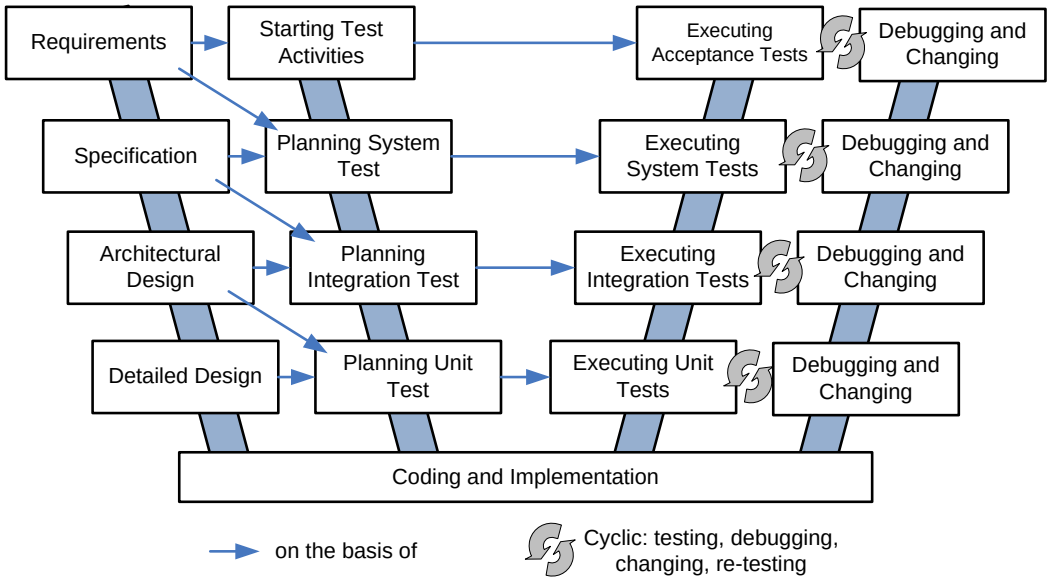


Figure 2.6: Definition of the W-Model based on Spillner [101].

- The *unit testing level* [21, S. 51] is used when component implementations are tested towards their specification.
- The *integration testing level* [21, S. 63] is used to check the harmonized connection of the different components.
- The *system testing level* [21, S. 71] is the first test that checks if all components are integrated and the complete system is ready for testing.
- The *acceptance testing level* is similar to the system testing level but adds the perspective of the customers view.

The disadvantage of the model is the rough division into constructive work including the implementation on the left-hand side of the V-Model and the more destructive tasks on the right-hand side. A planned-in removal of defects and regression test is not given [101]. A detailed explanation for testing levels in software engineering is presented in [4].

W-Model is shown in Figure 2.6 and is an extension and refinement of the V-Model XT. The testing aspect and the development activity are organized parallel which supports an early testing activity. The testing process will start after the implementation is complete. Further, the importance of the tests and the ordering of the individual activities for testing processes are clear.

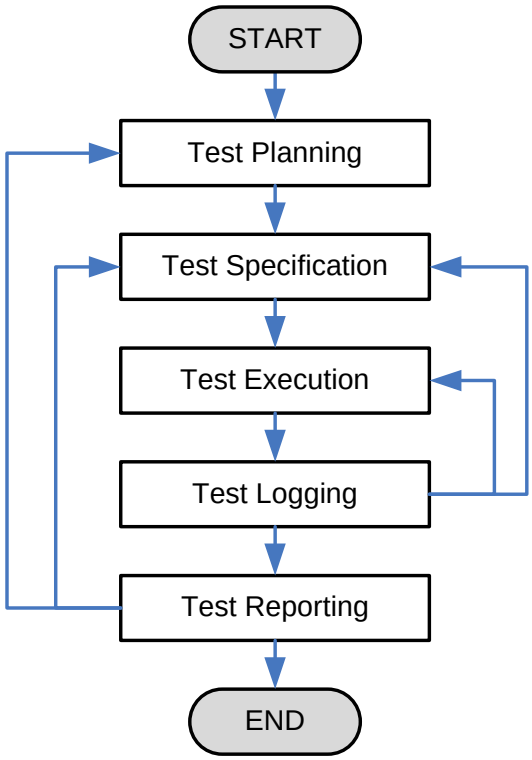


Figure 2.7: Test Process based on Spillner et al. [104].

2.4.2.2 Individual Test Processes and Test Strategies

Sneed et al. [94] presented a compilation of several testing processes based e.g. on Spillner et al. [104], Kaner [13]. Spillner et al. [104] presented a test process which is shown in Figure 2.7.

- The *Test Planning* provides a plan on how and which component of a SuT will be tested.
- The *Test Specification* includes the definition of the test cases.
- After the *Test Execution*, the test results are logged.
- The *Test Logging* logs all actions during the test execution, e.g. trace listings.
- The *Test Reporting* is an analyzing and a decision phase which results in an additional test execution or a final analysis of the executed test case.

A similar definition of a test process from [104] is given by the IEEE Standard 829 [105] but with different namings:

- Test Plan
- Test Design
- Test Specification
- Test Execution
- Test Reporting

There the Test Logging is combined with the Test Execution but an additional phase, the design phase, is included after the test plan and before the test specification. The test design is a refinement of the test planning phase.

Kaner et al. [13] refuse a fixed test process and recommend a flexible test process which will be adapted for the project situation. This basis test process defined by Kaner et al. [13] is also used by Sneed et al. [94, p. 13ff] and is defined as:

- The *Test Requirements Analysis* identifies the requirements of what should be tested and how can the test improve the software quality.
- The *Test Planning* provides a plan for testing the SuT.
- The *Test Case Specification* defines the structure of the test cases and test sequences without knowing the test data parameters.
- The *Test Data Preparation* is used when an extensive range or volume of data is needed for testing. Test data must be created for each and every test cases.
- The *Test Execution* executes the test cases to the SuT.
- The *Test Reporting* visualize the results of the executed tests.

2.4.2.3 Test-First Development

Another common technique to increase the software quality in the field of software engineering is TDD or also called TFD. This test process method breaks the traditional development process work-flow *first coding then testing* and is a widespread development technique for agile development processes. Based on system specifications and customer requirements, test cases are defined prior to the implementation of the SuT. First the test cases will be developed for a small increment of functionality, then the functionality is implemented for a correct test run [106–108]. Early test case definitions enable early defect detection in specification documents during test case construction, e.g. inconsistencies between requirements and missing and/or unclear requirements,

and represent the foundation for frequent test runs [109]. This can be included in a test strategy [110] which includes fast feedback on the current project state. The implementation of the test case is based on the specification and documentation of a program part that was defined by the stakeholders. TFD is used in combination with a continuous integrated testing process, which can and should be carried out in each development step, although the program components are testing towards each other, i.e. integration test [93, 111, 112]. The advantage of this development process is to achieve a better understanding of the specification and definition of requirements and to obtain the knowledge and individual requirements to test the control code. To use test cases in a test process by the above mentioned testing techniques, they must first be defined and implemented manually or generated automatically from the specification. Usually the manual implementation of a test case is very time consuming and very error prone, hence the trend goes towards to model-based test case generation [113].

Figure 2.8 presents the basic work flow of the TFD consisting of five phases: Specification phase, test implementation phase, implementation phase of the functional behavior, refactoring phase, and the final phase.

- In the *specification phase* all requirements for the software component will be defined.
- Further the test will be implemented based on the previously defined specification in the *test implementation phase*. This phase includes the first test run. Because of the missing implementation this first test run has to fail.
- At this stage, the *implementation phase*, the functional behavior will be coded and the test case can be executed.
- If the test passes, the *refactoring phase* can be started. In this step the functionality is already implemented but the quality and structure of the software may not be high enough. Therefore, the implemented behavior has to be optimized without changing the functionality of the code.
- After a successful validation of the refactored code with tests the *final phase* is reached. Finally, if additional functionality is needed, the process can be started all over from the beginning of the five phases.

Shull et al. [106] present a study about the evidence of effective TFD based on a systematic literature review . They analyzed the influence of TFD on the delivered quality, internal quality, and productivity. Madeyski [108] presents an overview of the majority of empirical studies which have investigated the TFD and pair programming versus the test-last and solo programming practice. Mattsson et al. [114] verified that TFD is a well applicable method for

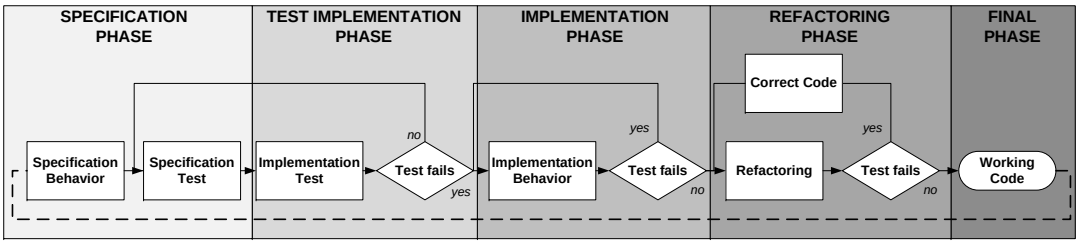


Figure 2.8: Testing and development work-flow of the TFD strategy.

the development of software projects as it provides a good improvement of implementation efficiency with limited additional effort. These works support the attempt for improving the software development process used for control applications.

2.4.3 Model-based Testing

A common testing technique and a well-known approach for an automatic test case generation in the field of software engineering is Model-based Testing (MBT) [77, 113]. This testing method promises higher quality, better coverage, and efficient change management [115]. Defined software models can be derived from the requirement specifications as a foundation for automated test case generation instead of hand-crafted individual tests [77, 78, 116]. Different specification methods, e.g. UML diagrams, are available, but only a subset of such UML diagram types are really suitable. A selection of suitable UML diagrams has been shown in Section 2.3.2 [87, 117]. The UML diagram family supports several diagram types to address the system structure (6 diagrams), the behavior (3 diagrams), and the interaction (4 diagrams) of entities.

- System structure diagrams present a static view of the system based on components, distribution of components, or classes.
- Behavior diagrams include use cases, state machines, and activity diagrams to define work-flows.
- Interaction diagrams focus on communication and collaboration of components.

Software models can be used to illustrate the structure and interaction between components, furthermore it can support automated code and test case generation [118]. During test execution the set-behavior of the model is compared with the actual-behavior of the test application. The complexity of the test problem is abstracted to a comprehensible controllable level, i.e. test engineers build a mental model of the system. This mental model is used to derive test

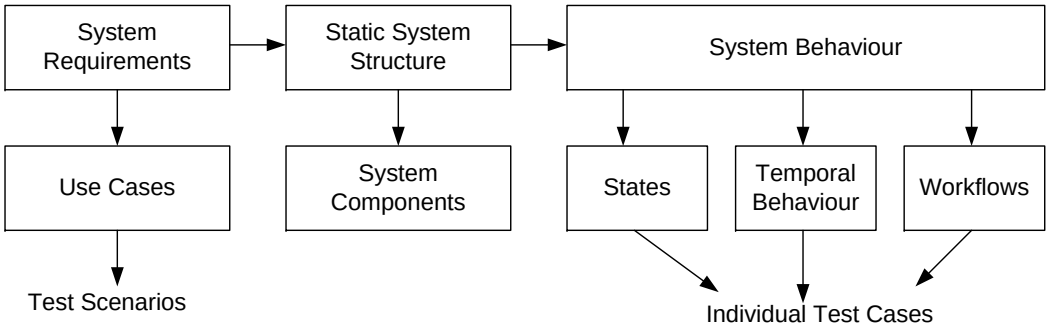


Figure 2.9: Test case generation with model-based testing derived from the UML diagram family [119].

cases for the implementation of the SuT [113, p. 280]. The test specification is the foundation for selecting test cases. Figure 2.9 gives a structured overview for a generic test generation system. This figure represents that the system requirements specification can be used for generating test scenarios which include several test cases. Individual test cases can be derived from the system behavior specification, e.g. states, temporal behavior description, work-flow description. Detail explanation about test case extraction is presented in Chapter 4.

MBT uses abstract models to generate test cases for testing an implementation. Figure 2.10 shows the data flow in a generic test generation system. The inputs of the software can be developed early in the cycle from the requirements information. Individual test cases are summarized in test suites, i.e. bundles of test cases with focus on a set of system requirements. The test suite includes inputs, expected outputs, and necessary infrastructure parameters to run the test automatically. Test generation can be especially effective for systems which are changed frequently. Testers can update the data model at any time and generate a test suite automatically. The suite of editing handcrafted tests can be avoided. For validation, the generated test cases should be checked manually in order to ensure that the model represents the system requirements and their specification correctly. Finally, the model is used to increase confidence in the understanding between customers and developers. Model-based testing depends on three key technologies [78, 120]:

- Notation used for the data model, e.g. UML diagrams,
- test generation algorithm, i.e. deriving test cases from the models,
- tools that support test case generation and may provide a framework for test execution.

There are several approaches for MBT as shown in a survey by Dias Neto [121] that characterizes and analyzes MBT approaches . The result of this study

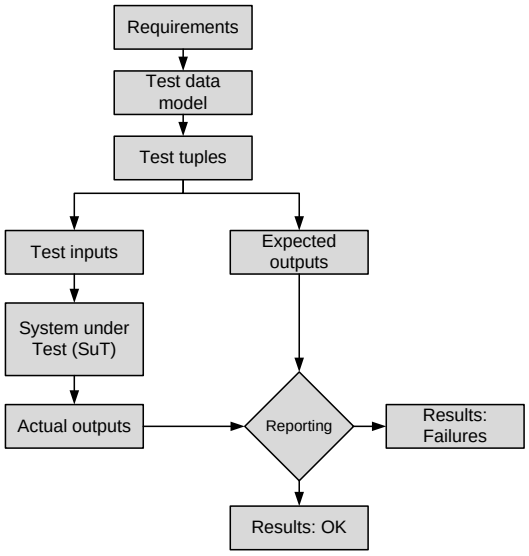


Figure 2.10: Architecture of generic test-generation system based on Dalal et al. [78].

is that in 78 analyzed papers only a few describe approaches in an industrial environment. Usually they have been developed for a specific context and do not get introduced to the industry. Pretschner et al. [122] state that tests that are derived without using a model detect fewer failures than using the MBT technique. The number of detected programming errors is nearly equal but the number of detected requirements errors is higher.

The *test case generation* part is a key technology included in the MBT process. Sneed et al. [94, p. 49ff] present techniques for automatic test case specifications based on defined software models.

In practice there are some combinations which are not valid and therefore constraints must be considered when test tuples are generated. Broy et al. [113, p. 356 ff] and Vigenschow [31, S. 283] give fundamental information about *Model-based Test Case Generation* and test automation.

Chevalley and Thévenod-Fosse [116] describe a technique that adapts a probabilistic method, called statistical functional testing, to the generation of test cases from UML state diagrams, using transition coverage as the testing criterion. Statistical functional testing involves exercising a program with input values that are randomly generated according to a given probability distribution over the input domain. A generic algorithm for the random generation of input values is presented that allows to produce sequences of test cases that trigger several times every transition for testing Java programs on sub-system level.

Fröhlich and Link [123] explain how test cases can be automatically generated with the aid of Artificial Intelligence (AI) methods by testing on system

level. The transformation process is separated into two steps. First the transformation from use case models into UML state diagrams is done. Then the state diagrams are furthermore mapped to a planning language and then a planning tool, which is called graphplan, produces the different test cases as solutions to a planning problem. The testing criterion is the coverage of every transition of the state diagram.

Kim et al. [124] propose a transformation method from UML state chart diagrams into extended finite state machines and flow graphs. They showed that conventional flow analysis techniques can be applied to test cases generation from UML state chart diagrams. Furthermore, using the transformation approach it is possible to flatten the hierarchical and concurrent structure of states to simplify complex UML structures. This testing technique is used for class testing, i.e. unit test level.

Hussain and Frey [46] use the generation of a transition tree (set of test cases) which is generated from each defined UML state chart diagram. Furthermore activity diagrams are used as well to generate test scenarios. Information about the test generation process is not presented.

A synoptic list of MBT tools and projects are presented by Micskei[10].

2.5 Testing of Embedded Systems Software

The field of Embedded Systems (ESs) is wide, and there is no exact definition or description. A definition by Marwedel [125] is: "ESs can be defined as information-processing systems that are integrated into a larger product". It describes a broad field of applications and products in which the structure may be is different [126], e.g. telecommunication industry, automotive domain. Relevant to safety applications are, e.g. control systems in airplanes. Further fields of applications are, e.g. refrigerators, toasters, or mobile phones. Such ESs controller are included in a product of mass production instead of industrial automation controller like a PLC which are used to control production systems. Software of PLC controllers are changed permanently instead of ESs software which are developed once per product.

Traditional software in the field of software engineering usually has no real-time requirements or time influences from the outside. They are systems whose functionality depends not only on the logical results of the calculation but also have a time behavior called *real-time systems*. A real-time system is a system that interacts with the real world and needs to produce its results within a predefined time. Some time constraints are called *hard real time constraints* which are defined as:

[10]http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html visited: August 2012

> "A time constraint is called hard if not meeting that constraint could result in a catastrophe." [127]

All other real time constraints are called *soft real time constraints* [125]. Assuming a hard real-time system, a wrong temporal behavior will result in major system damages or even in injuries to humans. Soft real-time systems tolerate a certain variation of durations and time spans. A wrong timing will only result in a degradation of the quality or performance of the system, depends on the controlled system [128]. A detailed illustration of real-time constraints and real-time systems is given in [113, 127–129].

ESs also react to external events which are called *reactive systems*. A reactive embedded system is a computer system which interfaces the real world with sensors and actuators. It is defined as:

> "A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment." [130]

Testing is a suitable method to ensure the aims of these properties for ES software. Such software tests are different to tests in the field of software engineering and industrial automation software. An ES should have the following properties based on [125]:

- *Reliability* is the probability that a system does not fail.
- *Maintainability* is the probability that a failed system can be repaired within a specified time interval.
- *Availability* is the probability that a system is operating correctly. Both reliability and maintainability must be high in order to achieve a high availability.
- *Safety* describes the property that a failing system avoids causing damage.
- *Integrity* describes the property that confidential information is private and that the authenticity of the communication is guaranteed.

In the following, several testing methods for ES software are presented. Marwedel [125, p. 201] state that ESs are tested by so called test patterns which are input patterns to test the behavior of the hardware system. Such test patterns are generated based on fault models which are defined models of possible faults. These generated test patterns are applied to the real and already manufactured systems.

Yu [131] presented a soft real time ES testing approach by analyzing the data-flow to distinguish points of interaction. The most used testing method

properties	IA	SE	ES
Real-time requirements	yes	no	yes
Reactive control	yes	no	yes
Real world interaction	yes	no	yes
Testing software process available	no	yes	little
Software duration of life	high	low	middle

Table 2.1: Comparison of the properties from the different domains. Industrial Automation (IA), Software Engineering (SE), and Embedded Systems (ES).

in ESs is MBT. Conrad [15] and Bringmann and Krämer [132] presents the MBT method for testing ESs software in the automotive domain. Most of the MBT methods are useable for testing control software such as testing techniques for Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), or HiL environment.

A testing technique for testing and verifying software of ESs in the automotive industry based on MBT is Time Partition Testing (TPT). TPT defines a test procedure for testing continuous behavior of ESs [132, 133]. The most important approach of TPT is the continuity of the test process, which means that the design flow of test case modeling as well as the following steps of test execution, test validation, and test documentation is in a definite structure and pretend a framework for all central test activities. The modeling language enables the definition of test cases for continuous and reactive system behavior and is mostly used for testing timing behavior of ESs. A TPT tool is available at PikeTec[11], other test tools for testing ESs are presented in [47, p. 83].

2.6 Comparison of the Different Domains

Testing is an appropriate method to reduce the risk of failure occurrence in operating systems [31, S. 63]. In the following, a comparison of testing software in different domains such as *Industrial Automation* (IA), *Software Engineering* (SE), and *Embedded System (ES)* are considered. Table 2.1 shows an overview of the required properties in the different domains.

There exists no systematic test process for testing industrial control software. First steps are available in academia to adapt testing techniques from the software engineering testing domain applied to industrial automation testing [111].

In the software engineering domain many testing approaches are available and used in practice. Several testing methods such as TFD combined with MBT are common in the software engineering testing domain [11, 108].

In the field of ESs, the software is developed alongside their hardware com-

[11] http://www.piketec.com/ visited: September 2012

ponents. Embedded software must often compensate the problems of the embedded hardware. Several application and software testing methods are available, most in the field of the automotive industry.

Summarizing, to ensure the software quality of industrial automation software, the different domains could be adapted to be use in the automation domain.

2.7 How Many Test Cases are Necessary?

> "A major activity in software testing research is deriving criteria that aid in selecting the smallest set of test cases that will uncover as many errors as possible" [134].

In the software engineering domain numerous methods for reducing the number of test cases are available. This field of research is completely new in industrial automation. This thesis does not focus on methods for number of test case optimization. This is beyond the framework of this thesis, but a short encouragement is given.

It is nonsense to expect that the testing is measurable by the number of test cases, but there is the need to keep the number of test cases as low as possible. For instance, the `E_CTU` FB defined in the IEC 61499 standard [55, p. 64 Table A.1] is a stateful FB. There exist 2^{17} possible input vectors, i.e. 2 non-concurrent event inputs + 1×16-bit data input. This comprehensible example already has 2^{18} output interface states. Therefore, full testing of the `E_CTU` FB would require 2^{35}, i.e. $2^{17} \times 2^{18}$, test cases.

Bieman and Schultz [134] presented a tool that estimates the number of test cases required to meet the all-du-paths testing criterion. The all-du-paths software testing criterion is the most discriminating of the data flow testing criteria which requires an exponential number of test cases in the worst case. The results of this case study shows that the worst case scenario only occurs, for instance, in one subroutine out of 143. 80% of the subroutines would require ten or fewer test cases.

Sayre and Poore [135] state: "The decision to stop testing can be based on a number of criteria, such as:

1. The confidence in a reliability estimate,
2. the degree to which testing experience has converged to the expected use of the software, and
3. model coverage criteria based on a degree of state, arc, or path coverage during crafted and random testing.

In practice it is best to use multiple stopping criteria."

It absolutely depends on the test approach and testing technique which one is used for the test case generation strategy. There is no method to know the exact number of test cases before implementing it. It depends on the test approach which is used. In a first step, by considering the presented test methods in Section 1.3.2, the number of test cases can be reduced.

2.8 Economical Aspects

> "Economics is the study of of how people make decisions in resource limited situations." [136].

The focus in this section is on the economical aspects in the field of testing and not on the technical aspects.

The potential of improvements in testing methods and tools is enormous. Bouyssounouse and Sifakis [47, p. 84] state that the cost of testing is estimated to take up between 30% and 50% of the development costs of embedded systems.

Myers [92] states that software testing has been known to consume about 50% of the development costs and spans about 50% of the development time. The most effective possibility to reduce the development time is to avoid specification faults and implementation faults. The earlier errors are found, the lower the costs of correcting the errors and the higher the probability of correcting them correctly [92]. Faults that are found early reduce the development costs drastically, as shown in Figure 2.5.

Mlynarski et al. [115] state that software testing can take up to 37% of the overall project costs, i.e. software engineering projects. According to this study, the most cost-intensive activities in testing are the test design and regression testing. Script-based test automation tries to improve the efficiency of regression testing by automated test execution while MBT methods are used especially to improve the efficiency of test design.

Hoffman [137] states that test automation is not always necessary, appropriate, or cost effective. If the benefit exists an expected return on investment analysis has to be done to show the benefit.

Slaughter et al. [138] state that: "Software quality is an investment that should provide a financial return relative to the initial and ongoing expenditures in the software quality improvement initiatives." They introduced three metrics in the software engineering economics context: cost of software quality, return on software quality, and software quality profitability index. An overview of the topic "Cost of Software Quality" is presented in [138] and [139, p. 43ff].

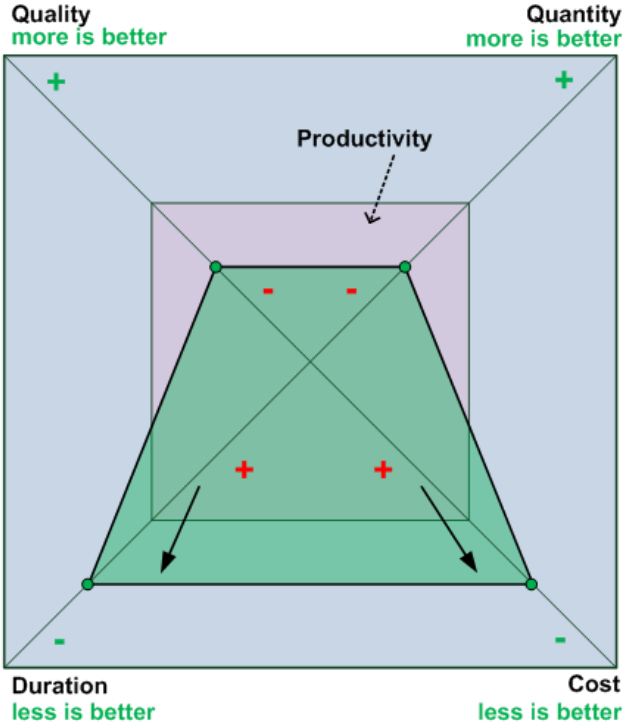


Figure 2.11: Devils Square by Harry M. Sneed [1].

TFD is a significant step forward from the test-last approaches. Fraser et al. [140] state that TDD works in practice because it helps to guide the design process from requirements into an actual running code. Madeyski [108] presented a detailed evaluation of the effects by using the TFD of agile software development practice with respect to the percentage of acceptance tests passed, design complexity metrics, and the number of acceptance tests passed per development hour which is an indicator of development speed and finally in costs. Furthermore, a better modularization, easier reuse and testing of the developed software products due to the TFD is suggested which also reduces costs in the development process.

Harry M. Sneed [1] proposed a concept called "Devils Square" which declares a fixed relation between the main factors of a software project. The main factors of a software project are: Duration, costs, quality, and quantity. He states that the change of one or more factors automatically influences the other factors. For instance, if a project should be implemented more quickly and cheaper, then the quality and the functionality, i.e. quantity are reduced when maintaining a constant productivity, see Figure 2.11.

Within this thesis a further variable is added which is called *innovation*. In order to use new testing methods and new testing techniques the devils square can be increased. That means, by developing new testing methods the quality

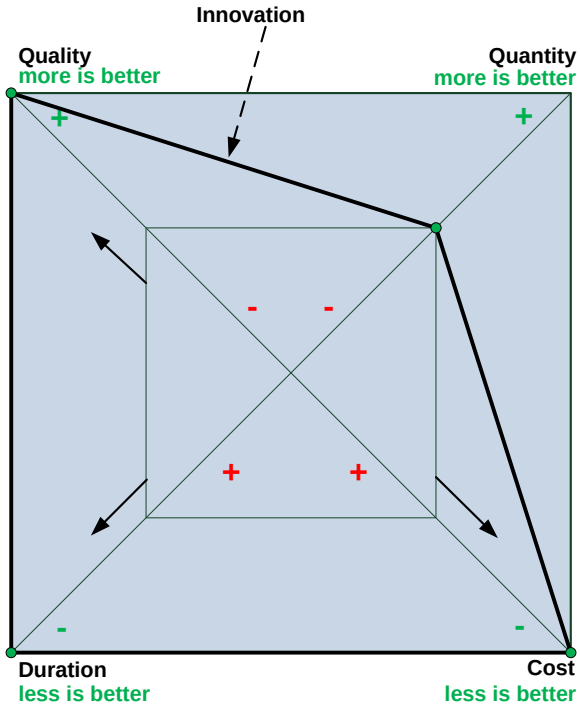


Figure 2.12: Devils Square including the innovation variable.

can be increased while the duration of the project and the required costs can be decreased. Note that the quantity variable cannot increase by testing and is therefore constant. Figure 2.12 shows the enlargement from the original square to the new increased square.

2.9 Research Questions

Most industrial control applications in the field of production systems and batch process systems are realized in the modeling languages IEC 61131-3 and IEC 61499. To ensure high software quality, testing processes are necessary which assist the development engineers during the whole life-cycle of developing industrial automation system. In order to use testing processes and testing methods for control software, changes can be done in a short time, development cycles are reduced, and efficient repeatability of the test process is possible while the quality of the control software is high.

In the following, four research questions which are derived from the related work analysis will considered in this thesis:

1. Systematic testing of control code is not a common practice in the field of industrial automation systems. Therefore it should be considered which test method is useful for which application layer. From this follows that test layers have to be identified first, e.g. system layer, unit layer. The first research question is:
   Which test method is suitable for the different application layers?
   In Section 3.2, the test levels which are suitable for different application layers are shown in Figure 3.5.

2. Developing high quality software requires a development process for industrial automation systems. A common development method from the software engineering domain is the TFD or also called TDD method. TFD is a significant step forward from the test-last approaches [108]. The TFD technique can be the basis for the development of new test-driven automation software. The second research question is:
   Is the test-first development approach from the software engineering domain a suitable method for developing and testing industrial automation code?
   The testing methods presented in Chapter 5 are used the TFD approach which is evaluated in Section 6.1.

3. Current automatic control applications development is characterized by an approach where control applications are individually developed for each application. This results in a high development effort per plant as there is only little reuse of software components between different applications which are hard to test. An important step towards an improved development process is an architecture which defines constraints on the control application structure. The third research question is:
   How should control applications be designed to enable the testing of their functionality in an easier way?
   Resulting design rules for control applications are presented in Chapter 7 which present constraints and structure definitions.

4. Once developed control software components such as FBs are stored in engineering libraries. Such components can be reused in new applications which should have a high software quality check by testing. The fourth research question is:
   Can testing improve the reusability of control software?
   The combination of the TFD approach and a new development structure definition enables the reusability of control software components. Section 7.2 presents an automation model which supports reusable testable control components.

CHAPTER 3

New Test Framework for Industrial Automation Systems

> "Users like Lockheed Martin state that test is being reduced by about 50% or more, while describing how early requirement analysis significantly reduces rework through elimination of requirement defects (i.e. contradiction, inconsistencies, and feature interaction problems." [141]

A test process has to be prepared and planned carefully to increase the quality of industrial control software. There is a need for a new test framework for automated testing of industrial automation applications which is presented in this chapter. Furthermore, a discussion of individual aspects for a testing process is done and how these aspects address various testing levels is described.

The test process for testing control software in an engineering process is presented in Figure 3.1 and is defined by four major aspects. These aspects of testing include:

1. Test Specification, i.e. test planning and organization,
2. Test Case Generation,
3. Test Case Implementation and Execution, and
4. Test Reporting on various levels.

The new test framework address these four aspects. Tests have to be structured hierarchically in order to increase their comprehensibility. A common way for test structuring is to introduce the following layers [26, 77, 142]:

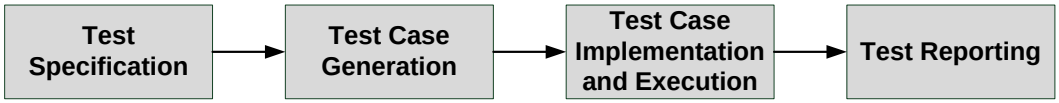


Figure 3.1: Process flow of the test process. It starts with the test specification process and finally shows the results of the test in a test reporting.

- *Test suite* is the topmost layer and consists of multiple test scenarios.
- *Test scenarios* are independent from each other and consist of one or a number of test cases.
- The lowest layer defines individual *test cases*. These are contained in test scenarios in a restricted order, based on dependencies that are defined in the behavior specification.

An overview of the test structure is shown in Figure 3.2. Test cases are separated in three kinds:

- Positive/normal test cases are necessary to test their intended (specified) behavior.
- Special test cases represent a system's behavior in the border area of regular system behavior, e.g. upper or lower limit, see Section 1.3.2.
- Error/negative test cases are used for testing the implementation in case of error states as well as the behavior on wrong input.

3.1 Framework for Automated Testing

In practice, the high effort to run tests manually limits the testing intensity [111]. Test automation encourages a high testing intensity for each reworked software version. Nevertheless, the application of automated testing requires an appropriate infrastructure and needs an additional effort of the setup of the testing framework. Test managers have to find a tradeoff between manual testing, i.e. high effort for every test run, and automated testing which results in a higher effort for the test framework implementation and lower effort for every additional test run. However, in the field of automation systems engineering the current observed practice relies mainly on manual and limited testing in a hierarchical systems design. Additionally, automatic control applications are designed graphically by using FBs and textually by using ST to implement system requirements. Based on these current practices, the test coverage seems to be rather low because of insufficient testing tool support.

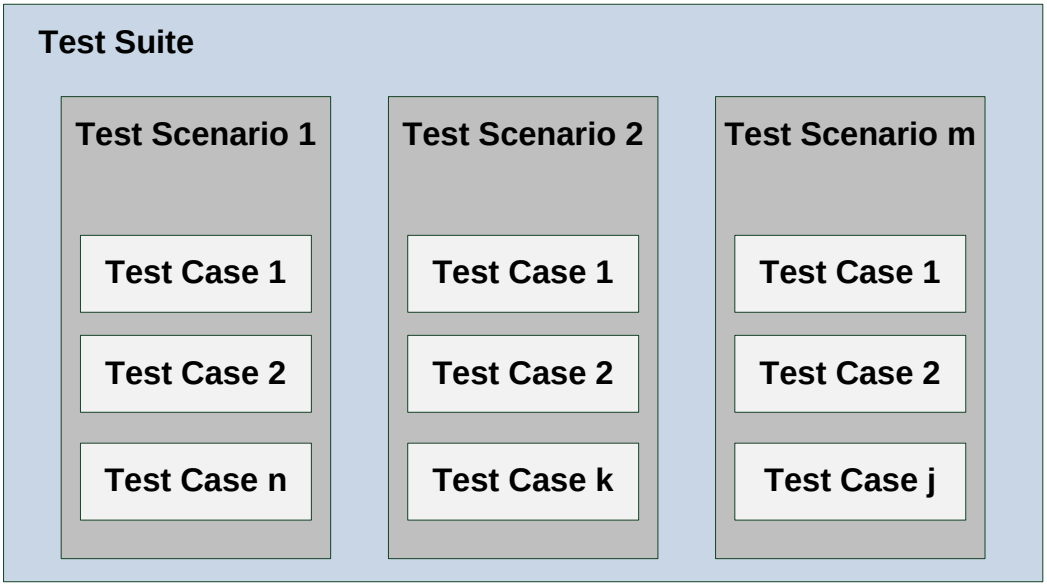


Figure 3.2: Structure of a test suite, including test scenarios and test cases.

Enabling automated testing during automation systems development requires a sound framework that enables early testing approaches such as TFD. Further requirements of the test framework are:

- Automation supported systematic test case generation,
- test execution, and
- test reporting on various levels of detail and from different perspectives.

An important pre-condition is how individual test steps, i.e. definition, execution, and reporting, can be connected to each other.

The aim of an automatic testing process is to create a test environment to bring up previously-specified input parameters, perform the test, and compare the results returned to the expected values. Figure 3.3 presents a schematic overview of the general test framework [143]. This proposed test framework guides the test engineers in the automation systems domain to increase the test intensity in a more efficient way.

This test framework consists of:

(A) The *Test Case Definition* is included in a test suite, which is based on customer specifications. A test suite includes a number of individual test cases and test scenarios. Such a test suite selects the required test case or test scenario for a test run. Test scenarios include a set of test cases to capture and test systems behavior by means of expected work-flows. A detailed explanation is presented in Chapter 5. Based on structured and

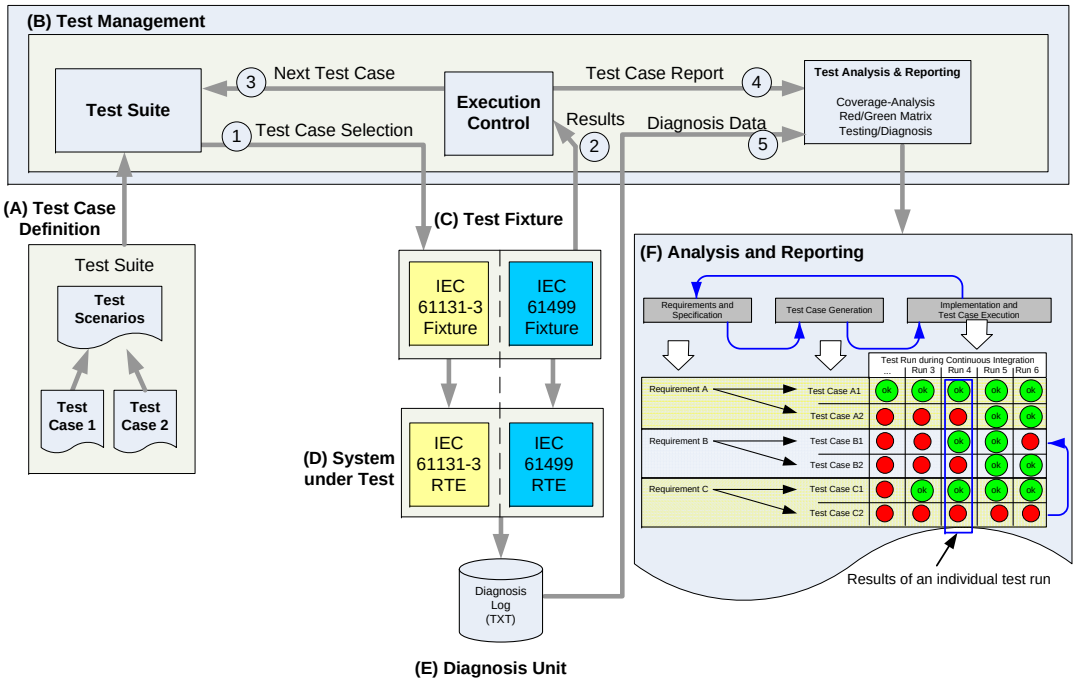


Figure 3.3: General structure of a test framework. The test-flow is represented by numbers starting from number 1 leading up to number 5. These numbers represent the steps how tests are executed, based on [143].

prioritized customer requirements and specification aspects, test cases are defined based on software models (see Chapter 4).

(B) The *Test Management*, also called Test Runner, is the core element of the testing process and is responsible for a controlled test procedure. It ensures that the test case is sent to the SuT and executes the test. The Test Management uses parameters for the test case initialization in order to execute individual test cases based on defined test scenarios. Note that capturing time stamps on the target systems enables measuring the temporal behavior of test case execution. After the test case execution results are captured on the individual test case level and will be aggregated to test scenarios for reporting purposes.

(C) The *Test Fixture* is a required interface between the Test Management system and the SuT. Every test configuration requires an appropriate test fixture related to the implementation standard. Note that the implementation of the IEC 61131-3 fixture and the IEC 61499 fixture are different.

(D) The *SuT* refers to a system that is being tested for correct operation and represents a configuration of the software and system products, e.g. component versions and/or variants.

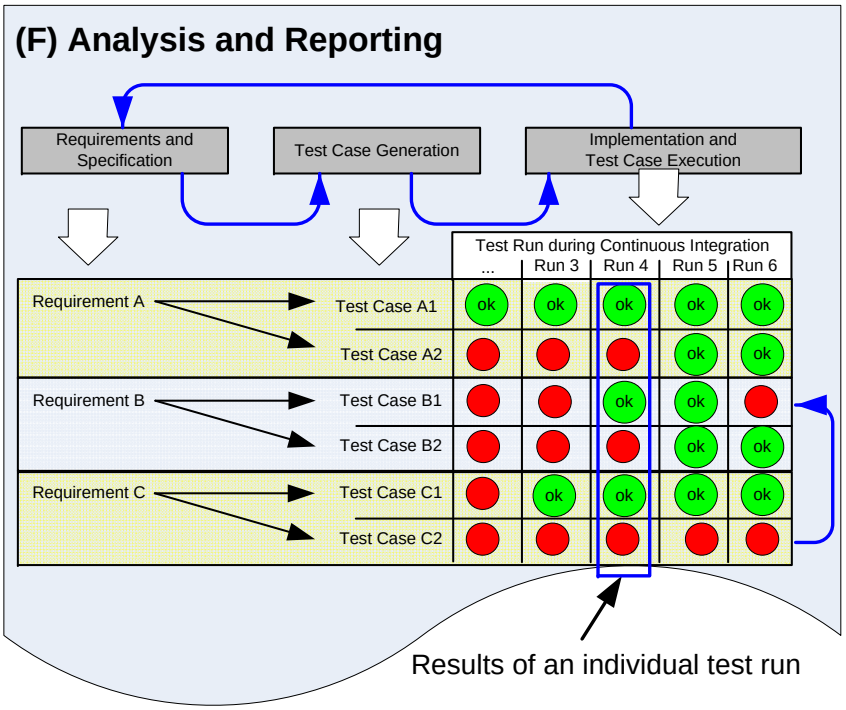


Figure 3.4: Analysis and reporting of the test framework, based on [143].

(E) The *Diagnosis Unit* logs all relevant data of the test environment which can be used for the presentation of the test results. In addition, test diagnosis data derived from the target system are available in order to analyze the system's behavior in more detail based on log-files, e.g. in error cases.

(F) Finally, the *Analysis and Reporting* unit processes the results of the test procedure which is organized by the Test Management system. It enables the analysis of individual tests, e.g. on unit level, and test scenarios, e.g. architecture and system level. The reporting unit evaluates the test results and prepares the values for a clear and representative view, which is shown in the test evaluation unit. The execution results of the test case and the test scenario are linked to the customer and system requirements, so that the project can be monitored as defined in a TFD development process. Figure 3.4 (same as Figure 3.3(F) in detail view) present a sample test report for a set of frequent test runs, e.g. during regression testing, for a set of requirements and derived individual test runs. Note that the Analysis and Reporting unit can include coverage analysis, i.e. share of requirements covered by test cases and test scenarios.

Individual test scenarios and test cases are selected, e.g. based on value contribution of corresponding requirements and are executed by the corre-

sponding test fixture (see item (1) in Figure 3.3). Results of one test run (2) are collected by the Execution Control component of the Test Management system. The Execution Control component included in the Test Management system is responsible for coordinating and controlling individual test runs (3) and also accounts for the preparation of test result analysis (4). Simultaneously additional diagnosis data is provided for the test analysis (5). Based on the results, an analysis and reporting functionality is available for a more detailed investigation, e.g. test coverage measures, quality metrics, and project observations, which can be derived automatically [48].

The application of this test framework leads to an automated frequent testing approach and can be included in a *continuous integration strategy* [110]. The combination of the continuous integration strategy and the proposed test framework enables the TFD approach strategy. Figure 3.4 presents a sample snapshot of the analysis and reporting part of the TFD process. Requirements are split into test cases and implementation tasks which are tested in various and frequent test runs. Because of this continuous integration strategy, the current project state remains transparent along the product development lifecycle. Even side effects, i.e. other test cases and/or requirements which are affected by individual test runs negatively, can be easily identified. This means that, if complex software parts are implemented in a nested way, every small change may have a major impact on other functional requirements or implementation parts . For instance, test execution of test case C2 (Requirement C) has a negative impact on test case B1 (Requirement B) in test run 6. Note that these short iterations allow an early identification and removal of defects.

3.2 Test-Levels / Test-Layer Approach for Industrial Automation Systems

In general, there are three testing levels observed in the automation systems domain [119]:

- Component level tests for individual components, i.e., unit tests,
- architecture and sub-system specification with focus on integration tests,
- systems-level tests at the customer and developer site, i.e. acceptance, factory, and system tests.

Figure 3.5 illustrates those three testing levels of automation systems applying a bottom-up design. It starts with a detailed view on individual components and units on the lowest layer, a more abstract view on the sub-system

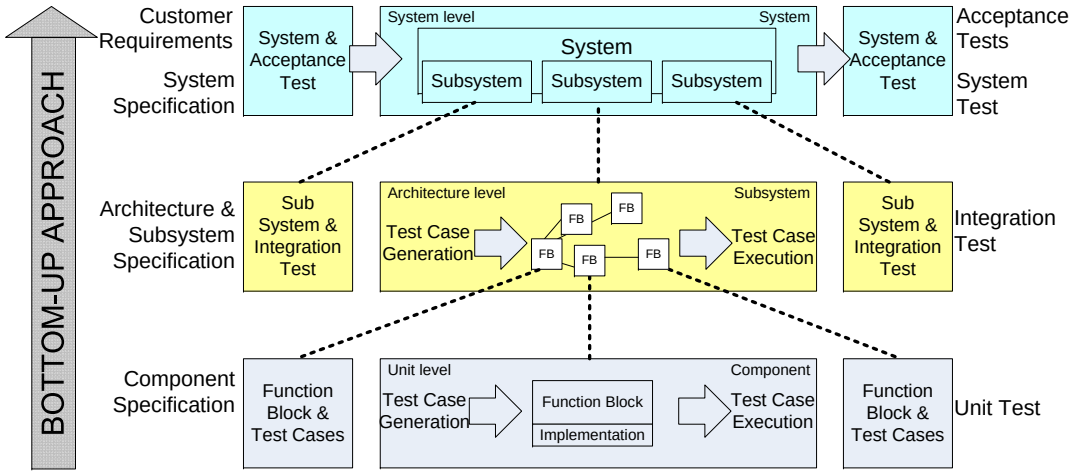


Figure 3.5: Bottom-up implementation design of the test levels in the automation systems domain. Units on the lowest layer, architecture layer with focus on the integration of components on the middle level, and the system level on the highest layer [119].

and architecture level with focus on the interaction of components on the middle layer, and finishes with the system level as well as the business level (requirements and systems) on the highest layer.

Depending on the field of application in automation systems development, there is the need to capture system requirements, modeling the system's structure, and modeling the interaction and communication between units, components, and sub-systems. Program units are arranged in a unit, i.e., smallest unit, a component, i.e. encapsulated functions, a module, i.e. combination and integration of components, and into a system, i.e. complete system.

In the following the test levels, also called test layers, are explained in more detail with the focus on industrial automation.

3.2.1 Unit Tests

The unit test is normally done by the developer of the software component to expose local defects of the algorithm [144, p. 20]. It can be summarized that unit testing can be seen as the smallest testable piece of software.

Software artifacts without external dependencies provide the basis for unit testing. Their fitness for use is assessed with appropriate interface oriented test cases, which check the units' functional behavior. These units (units under test) must have unambiguously testable behavior as well as the test specification must be defined unambiguously.

The tested units form the basis for the application development. Hence, the testing of as many units as possible shall be achieved. Therefore, an automatic

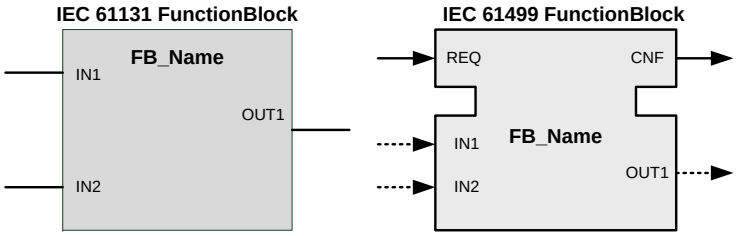


Figure 3.6: Unit component based on an IEC 61131 FB (left hand side). Unit component based on IEC 61499 FB (right hand side) which can be a BFB or a CFB.

test process implemented in a TFD process is desirable.

A case study of unit testing techniques is presented, evaluated, and classified in [145]. Unit testing is extensively used by agile programming methods, e.g. eXtreme Programming, and unit test frameworks, e.g. JUnit, have constantly improved for several years. These unit test frameworks typically require that the tests are implemented in the same language as the code to be tested.

The greatest advantage of unit testing is gained if the units are used in different application contexts without a changed behavior. Only then the tests guarantee that the false behavior and resulting errors only stem from the combination of the units. Software components as defined in [146] feature this characteristic. There a software component is defined as a unit of independent deployment, of third-party composition, and with no (externally) observable state. Another aspect of component orientation is that components shall be usable without any knowledge of its internals. Figure 3.6 shows a unit of an IEC 61131 FB and an IEC 61499 FB. In order to facilitate this aspect, this thesis focuses on black box testing of control software in IEC 61131-3 and IEC 61499 control applications. This means that for the test specification and process no knowledge and assumptions of the internals of the units are used. Solely the public specified interface is used for the specification and execution of tests.

Sünder et al. [147] show that FBs in general are considered as software components when applying the software component concept by [146] to IEC 61499. This is especially valid for BFBs as they perfectly encapsulate the internal data. The contained algorithms as well as the ECC are only allowed to access internal or interface data. For SIFBs the situation is different. SIFBs contain hidden interfaces to the underlying device specific services. This violates the definition of software components. However, these services need to be controlled by and interact with performing tests. This is typically done during the device development by the device vendor and needs special knowledge of the device or the service represented by the SIFB.

Finally, IEC 61499 provides the CFB as the third FB type. CFBs encapsu-

late a set of other FBs (BFB, CFB, or SIFB) and provide this combination as a new unit to the application developer. Nevertheless the type of CFB is indistinguishable from any other FB for the application engineer, i.e. provided FB interface. Therefore, CFBs can also be handled as units for the unit test framework such as BFB. However, the use of FBs within CFBs to BFBs and other CFBs, as SIFBs is restricted because it will break the software component assumption. This is reasonable as CFBs with contained SIFBs will result in device specific functionality which cannot be freely used in any application. The development of reusable application components and I/O access is still an open issue [148].

The software component concept by [146] is also applied to IEC 61131 FBs. These FBs can include other FBs which encapsulate their functionality similar to the proposed approach of IEC 61499.

IEC 61499 FBs are defined as passive elements, which execution can be requested by input events. Associated data inputs will be sampled at the occurrence of the triggering input event. The encapsulated functionality will take the data inputs, internal data as well as data outputs and performs its calculations generating output data and zero or more event outputs, each again with associated data outputs. This behavior is also the starting point for the targeted black box unit tests. With these the observable behavior of the FB's interface are tested. This behavior is formalized to a transformation function FB. It takes an input vector $\vec{I} = (ei_x, di_1 \cdots di_r)$ and produces an output vector set $\boldsymbol{O}$. ei_x represents an input event and $di_1 \cdots di_r$ represents all data inputs [42].

$$FB \quad : \quad \vec{I} \mapsto \boldsymbol{O}. \tag{3.1}$$

$\boldsymbol{O}$ consists of zero or more output vectors $\vec{O}_x$

$$\boldsymbol{O} \quad = \quad (\vec{O}_1 \cdots \vec{O}_y), \qquad \boldsymbol{O} = \emptyset \quad : \quad y = 0, \tag{3.2}$$

$$\vec{O}_x \quad = \quad (eo_z, do_1 \cdots do_s). \tag{3.3}$$

Similar to the input vector, eo_z represents an output event and $do_1 \cdots do_s$ represent all expected data outputs. y represents the number of output vector elements. Equation 3.2 means that if no output event and no output data exist the number of output vectors $\vec{O}_x$ is zero, i.e. $y = 0$.

The formulation for IEC 61131 is similar to the IEC 61499 but the input event ei_x and output event eo_z have to be removed from the equations.

The mapping in Equation 3.1 is only one-to-one and onto if the FB has no internal state, e.g. a simple sine function. However, as FBs may have internal states, i.e. ECC states, internal variables, and the values of data outputs, the history of previously applied input vectors $(\vec{I}_1, \ldots, \vec{I}_t)$ is required to determine the according $\boldsymbol{O}$. Therefore a sequence of $\vec{I}, \boldsymbol{O}$ tuples is needed for describing an FB's observable behavior [149, p. 45ff]:

$$S_{IO} = \{\{\vec{I}_1, \boldsymbol{O_1}\}, \ldots, \{\vec{I}_t, \boldsymbol{O_t}\}\}. \tag{3.4}$$

Typically unit tests are separated into test cases. Each test case checks a certain aspect of the component under test. This separation into test cases has the advantage that tests are easier to develop and to maintain. Furthermore it is easier to identify the reason for a failed test. Applying this to testing IEC 61499 FBs, a test sequence will be a S_{IO} as defined in Equation 3.4. The full test suite is represented by a set of these test sequences:

$$U_T = \{S_{IO,1}, \ldots, S_{IO,u}\}. \tag{3.5}$$

A unit test execution system, i.e. Test Management system, will take this data. For each test sequence entry the input data and input event to the FB under Test (FBuT) is applied. Furthermore the Test Management system will monitor the outputs (events and data) and compare them to the provided output expectations. In order to simplify the development of test cases, each test sequence assumes that the FBuT is in a defined initial state. The unit test execution system has to ensure that this prerequisite is met.

These test sequences can get very complicated and hard to define, especially with more complex FBs with many inputs and outputs, i.e. events and data. Therefore means are necessary that allow the description of the behavior of FBs in a way more intuitive for humans and which than can be transformed into the unit test representation according to Equation 3.5. The presented equations are used to define the structure of the test cases and test sequences in a formal way without having the guarantee of completeness.

3.2.2 Integration Tests

Integration testing is the phase in the testing process after the unit testing, in which several units are combined and tested as a small network group, i.e. as sub-system. The main focus of integration tests is to expose the defects of interactions between different units. Therefore the execution behavior of several units, i.e. FBs in a network, is tested. A main advantage of integration tests is that the FB units can be developed by different engineers and the interaction of the developed unit components can be tested for their correctness.

Figure 3.7 as well as Figure 3.8 present a small example part of an industrial automation network designed in IEC 61131 and IEC 61499, respectively FBs are connected to a part of the system implementation which is tested. Figure 3.7 shows five IEC 61131 FBs (FB1-FB5) and it connections. The FBs can include several FB units. Similar to Figure 3.8 the FB network consists of five IEC 61499 FBs (i.e., FB1 and FB4 are BFBs; FB2, FB3, and FB5 are CFBs that include other FBs too). There exist four approaches in integration testing[12], [13]:

[12] http://softwaretestingfundamentals.com/integration-testing/ visited: March 2013

[13] http://www.iai.uni-bonn.de/III/lehre/vorlesungen/SWT/OOSC05/slides/15%20-%20Testing%20%282-2%29.pdf visited: July 2013

Big Bang is an integration testing approach where all or most of the units are combined together and tested at one go. This approach is taken when the testing team receives the entire software in a bundle. The advantage of this approach is that everything is finished before integration testing starts but the major disadvantage is that this approach is time consuming, because it is difficult to trace the cause of failures due to the late integration. For example, see Figure 3.7, Test (FB1, FB2, FB3, FB4, FB5).

Bottom Up is an integration testing approach where bottom level units are tested first and upper level units are tested step by step after that. This approach is used when a bottom up development approach is followed. For example, see Figure 3.7:

1. Test (FB5),
2. Test (FB4, FB5) and Test (FB3, FB5),
3. Test (FB3, FB4, FB5), Test (FB2, FB4, FB5), and Test (FB1, FB4, FB5)
4. Test (FB1, FB3, FB4, FB5) and Test (FB2, FB3, FB4, FB5)
5. Test (FB1, FB2, FB3, FB4, FB5).

Top Down is an integration testing approach where top level units are tested first and lower level units are tested step by step after that. This approach is used when a top down development approach is followed. Mock-up objects are needed to simulate lower level units (i.e., not implemented systems behavior) which are not available during the initial phases. For example, see Figure 3.8:

1. Test (FB1),
2. Test (FB1, FB2, FB3),
3. Test (FB1, FB2, FB3, FB4),
4. Test (FB1, FB2, FB3, FB4, FB5).

Sandwich/Hybrid is an integration testing approach which is a combination of the top down and bottom up approach. The system is viewed as having three layers: The target layer in the middle, another layer above the target layer (top layer), and the last layer below the target layer (bottom layer). The testing converges in this approach towards the target layer. For example, see Figure 3.8, Bottom Layer Test:

1. Test (FB5),
2. Test (FB3, FB4, FB5),
3. Test (FB2, FB3, FB4, FB5),

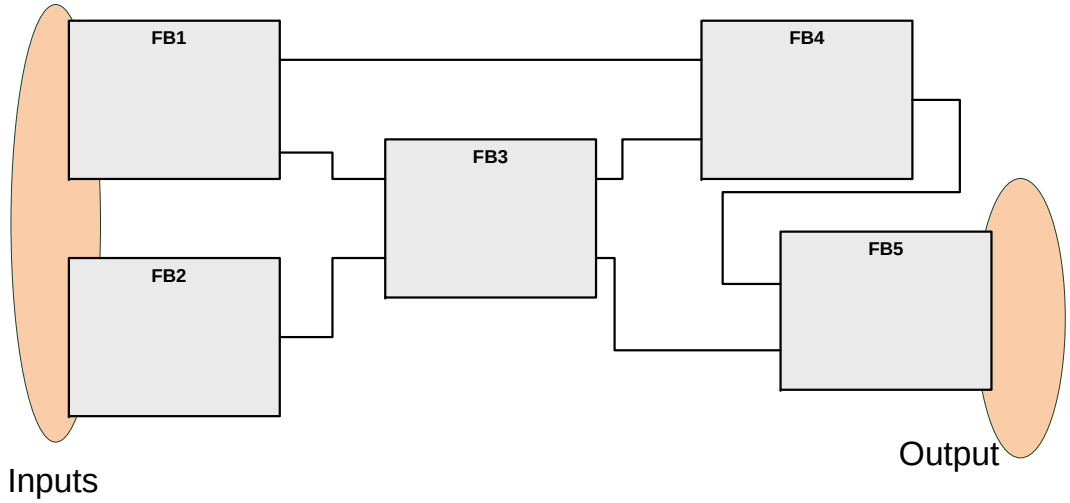


Figure 3.7: Part of an IEC 61131-3 implementation used for integration tests.

and Top Layer Test:

1. Test (FB1),
2. Test (FB1, FB2, FB3),
3. Test (FB1, FB2, FB3, FB4), and

finally, Test (FB1, FB2, FB3, FB4, FB5).

Summarized, the Big Bang approach is used when all or most of the developed units are connected together for a complete system or major part of the system and then used for integration testing. This approach is a nonincremental integration testing approach compared to the three other integration testing approaches. The main advantage of the Bottom Up approach is that bugs are more easily found because of the step by step integration testing. The test conditions of this approach is easy to create and the observation of the test data is well-arranged. With the Top Down testing approach, it is easier to find a missing branch link but the need for Mock-up objects are essential which is a major disadvantage of this approach.

3.2.3 System Tests

Following the individual levels of automation systems engineering (see Figure 3.5), the focus of system testing is on

- systems requirements as foundation for product development and
- technical design concepts, which can be refined successively on every level.

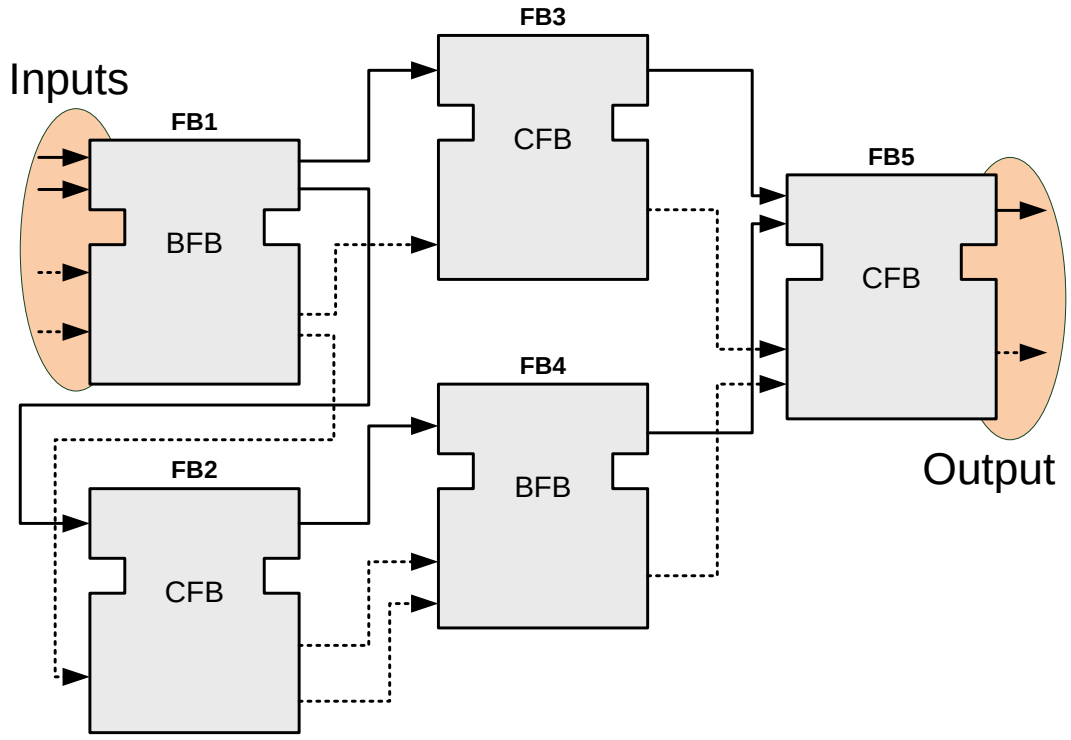


Figure 3.8: Part of an IEC 61499 system (sub-system) implementation consists of BFBs and CFBs used for integration tests.

The systems test level focuses on basic customer requirements and on system, acceptance, and factory tests. The main purpose of system tests is to expose defects which are connected to the whole software code. System testing of automatic control software is testing conducted on a complete, integrated system in order to evaluate the system's compliance with its specified requirements. Therefore, the scope lies on black box testing which requires no knowledge of the inner design of the code or logic.

Testing the whole system, means that all of the "integrated" software components have successfully passed integration testing. There is the need for input- and output- parameters which are available to the tester based on the requirements specification. In most cases, the system test can be done automatically by using a simulated environment [144, p. 21].

Special types of system testing are acceptance testing and regression testing. The *acceptance testing* is related to the system test and demonstrates that all customers' requirements are fulfilled [150]. *Regression testing* is applied to modified software to detect new defects. The simplest regression test is to re-run all test cases in the initial test suite which may need an unacceptable amount of time. An alternative testing technique is the regression test selection whereas only a subset of the initial test suite will be executed. This method

has disadvantages but a trade-off between the time to select and run the test cases and the fault detection ability of the test cases has to be done [151].

3.3 Summary

This section presents a test framework, which is applicable for automation supported systematic test case selection, execution, and reporting on various levels of detail. Further, the proposed test framework environment is suitable for industrial automation and it is a basis for a TFD approach in this domain.

In general, three testing levels are defined which are appropriate in the automation systems domain. Unit and component level, architecture and subsystem level, and system-level are directly mapped to unit test level, integration test level and system test level.

By using the test framework as a basis with the knowledge of the different test levels, a test specification is required which is usable for the industrial automation domain. The next step is a selection of test specification models for the industrial automation domain.

CHAPTER 4

Selecting UML Models for Test-Driven Development Along the Automation Systems Engineering Process

An important issue is the identification of the most valuable test cases to focus on key components of the system under development. There is the need for a model-based specification approach for industrial automation systems specification to be flexible to changing requirements. In the following, a selection of appropriate UML models is presented and evaluated for the use in industrial applications. For testing software components the test case definition as well as the test case generation have to be specified.

4.1 Model Specification

Models can support engineers during design, construction, verification, and validation of systems by systematically capturing static system structures such as components and interaction between components, as well as systems dynamics, e.g. work-flow and behavior [113].

The application of models increases the capability of QA activities, readability, and understandability of the requirement and behavior specification. Because of a graphical representation of static system structures and system dynamics, models aim at supporting engineers from various disciplines in conducting such activities. In addition, models can support (semi-automated) code and test case generation [113]. Typically, system engineers have to comprise a set of skills in a heterogeneous development environment. For instance, automation systems engineering routinely includes mechanical, electric, and software components. Engineers from different disciplines have to

interact and collaborate during the development process. However, various disciplines apply different sets of models within the individual scope of the discipline, which may be unfamiliar and hard to understand for engineers coming from another discipline.

Models such as UML [152] and SysML [153] promise to support the communication and the collaboration between various stakeholders in the automation engineering process. Common models are able to support QA by doing reviews and testing in order to increase systems quality. In addition, models are the foundation for TDD because test cases can be systematically derived from models in most instances automatically [21, 113]. There is a wide range of models. In this thesis the focus is on the UML diagram family because it is a well-known and well-investigated approach in software engineering development. Using UML in the automation engineering domain is a promising approach to increase the handling of the development processes and product quality [154].

4.2 UML in the Software Engineering Domain

The UML diagrams [152], which are embedded within a software engineering process, are widely used in the classical software engineering domain and help to:

(a) Identify and define various aspects of the systems,

(b) enhance communication and collaboration between various stakeholders, and

(c) enable automated code and test-case generation.

Depending on systems, component complexity and risk, various models help to identify and model the system's structure and the expected behavior [119]. Because of the benefits in the software engineering development, UML diagrams are promising approaches for an application in the industrial automation domain. UML 2.0 consists of an overall number of 13 diagrams, organized in structure diagrams and behavior diagrams. Behavior diagrams include subsets of diagrams, i.e. interaction diagrams, focusing on interaction and collaboration of system components. UML diagrams describe the system under construction from various perspectives.

In the following the three main diagram types are presented:

- Structure diagrams,
- behavior diagrams, and
- interaction diagrams.

Structure Diagrams describe the static structure of a (distributed) system which provides 6 diagrams:

1. *Deployment Diagrams* present the run-time configuration of a (distributed) system including hardware and software components.
2. *Component Diagrams* help identifying the underlying architecture of systems with focus on interfaces on various levels, i.e. system, subsystem, and individual components.
3. *Class Diagrams* describe data structures of systems on a detailed level close to the implementation.
4. *Object Diagrams* are used to present instances and the static relationships of instances at run-time.
5. *Composite Structure Diagrams* define the individual architectural components and their interactions.
6. *Package Diagrams* enable structuring the models and related artifacts of the UML diagram family.

Behavior Diagrams describe the dynamic behavior of the system and include three diagram types:

1. *Use Case Diagrams* describe the functionality of a system from the user perspective including the functional behavior without considering order and sequence of actions.
2. *State Chart Diagrams* describe individual states and transitions of systems and are applicable if system states change depending on system inputs, e.g. emergency stop.
3. *Activity Diagrams* are based on business modeling and provide work-flows (sequences of steps).

Interaction diagrams are a subset of behavior diagrams with focus on collaboration and communication between system components.

1. *Sequence diagrams* model the temporal behavior of work-flows and related objects (call and response) including timing information.
2. *Timing diagrams* describe the individual system states over time. Timing diagrams are widely used in electronics to specify digital circuit design.
3. *Interaction overview diagrams* model sequences and interactions.

4. *Communication diagrams* are based on messages and illustrate the communication work-flow between components.

The construction of models requires additional effort. Thus it is necessary to focus on the most valuable model diagrams and identify a subset of models which are sufficient to support TDD in the context of industrial automation software engineering. For such projects the key question is, which set of models, in this case UML diagrams, is suitable for industrial automation applications [154].

4.3 Criteria for Model Selection in the Industrial Automation Domain

Lüder et al. [155] identify 6 information sets for model selection (architecture, process of control decisions, signals and states, communication, control-relevant data structures, and interaction and communication processes). These are summarized into *architecture and structural aspects (SA)* and *temporal behavior (BE)* criteria. Nevertheless, *requirements (RE)* and *risks (RI)* are additional critical aspects from the business perspective. A detailed requirements analysis and requirements specification are key aspects to reduce risks of complex applications. A set of four criteria for model selection and application are identified:

1. *Requirements (RE)* systematically defined to derive test scenarios from the user perspective. Such test scenarios can be related to integration tests, systems tests, and factory tests.

2. *Architecture and Structural Aspects (SA)* describing the basic system structure to focus on individual parts of the system for construction and testing purposes.

3. *Temporal Behavior (BE)* and functional behavior are defined by considering, i.e. states, activities, and timing aspects. Note that work-flows represent sequences of steps of operations and can be used for test case generation especially for generating test scenarios automatically.

4. *Risk (RI)* and Complexity. Depending on the risk and complexity of components, e.g. failures and incorrect specification definition, specifically focused models become reasonable to capture individual systems characteristics, e.g. by using refined sequence charts, that help to understand and mitigate risks.

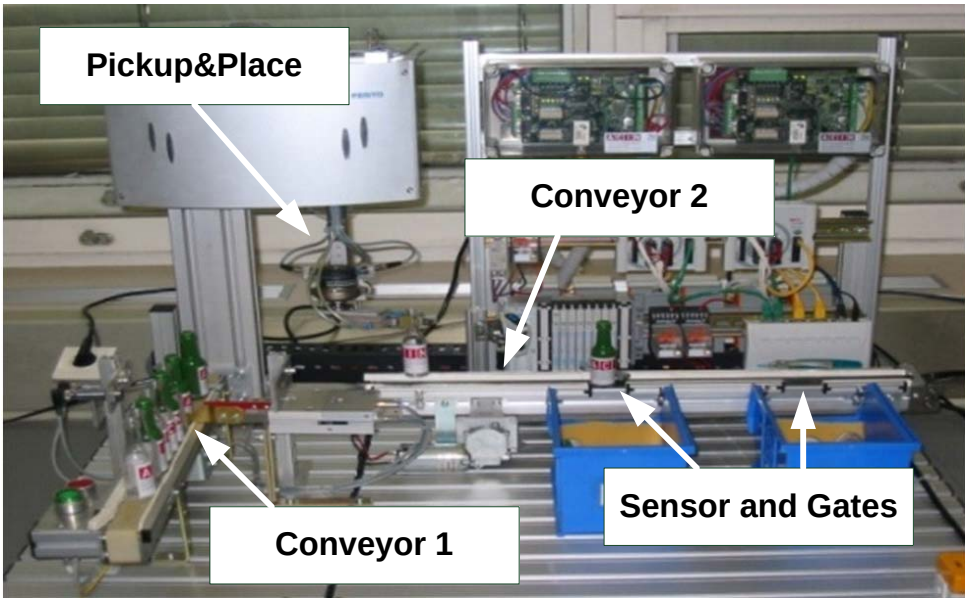


Figure 4.1: Bottle sorting machine used as an application for showing the specification method by using UML model diagrams.

4.4 UML Diagrams Applied on a Sample Application

To evaluate the strengths and limitations of the model selection from Section 4.3, an application of a bottle sorting machine by using the UML models for TDD is presented. The main goal of the bottle sorting machine is to sort a sequence of incoming bottles into two different boxes, depending to the color of the bottles. The bottles arrive on Conveyor 1 and are transferred, one piece at a time, to Conveyor 2 via a Pickup&Place unit. A color sensor, attached to Conveyor 2, determines the color of the bottle and activates one gate for sorting purposes according to the color. The bottle sorting machine is simple enough to show the benefits of models and TDD application but is still relevant to demonstrate the industrial automation engineering process approach for research purposes. Figure 4.1 shows a picture of the bottle sorting machine.

Thus, this application represents a promising prototype application for introducing UML models and the TDD approach. The bottle sorting machine is simple enough to allow a detailed investigation and is still relevant to demonstrate the industrial automation engineering process for research. The proposed approach will also work with more complex systems.

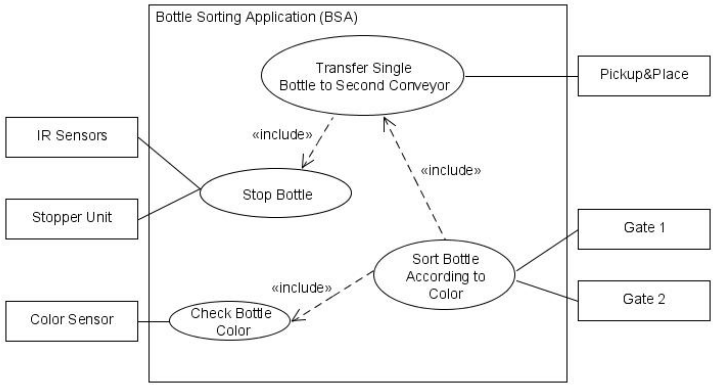


Figure 4.2: Model of the Use Case Diagram of the bottle sorting machine. Solid lines represent the connection with actors, e.g. sensors and actuators, dashed lines represent the internal dependencies.

4.4.1 Systematic Definition of Requirements

Use Case Diagrams focus on user requirements to identify the related actors and use cases according to the customer needs. Use case diagrams are used primarily to capture functional requirements of a system or sub-system and identify related parts of the system or external resources required by the system to fulfill its task. Use cases also enable decomposing functional requirements into smaller and manageable tasks. Because of the graphical representation of requirements, use cases are easy to understand by individual stakeholders and support interaction with stakeholders who may not be very familiar with technical details. Use cases support the identification of scenarios for testing purposes. Scenarios are sequences of individual actions, e.g. moving the arm from the start position to an arriving bottle at Conveyor 1. Note that scenarios can consist of individual test cases, e.g. testing the starting position of the arm, the movement, and the duration of a movement. Figure 4.2 presents a use case diagram for the bottle sorting machine.

An *Activity Diagram* is a representative of a UML behavior diagram and displays the work-flow behavior of a system by depicting basic activities and dependencies between activities a system has performed. They describe data and control flows within the system. Based on activity diagrams, test scenarios and test cases can be derived directly from the model. Figure 4.3 presents an activity diagram of the overall behavior of the bottle sorting machine.

A test scenario based on the activity diagram covers the overall bottle sorting machine:

1. Starting the bottle sorting machine.
2. Waiting for a bottle.

3. Moving the bottle from Conveyor 1 to Conveyor 2.
4. Determining the color of the bottle.
5. Activates the related gate to sort the bottle according to the color in Box 1 or Box 2.

Note that individual test cases can be derived directly, e.g. testing the transfer of the bottle from Conveyor 1 to Conveyor 2, by testing the Pickup&Place unit.

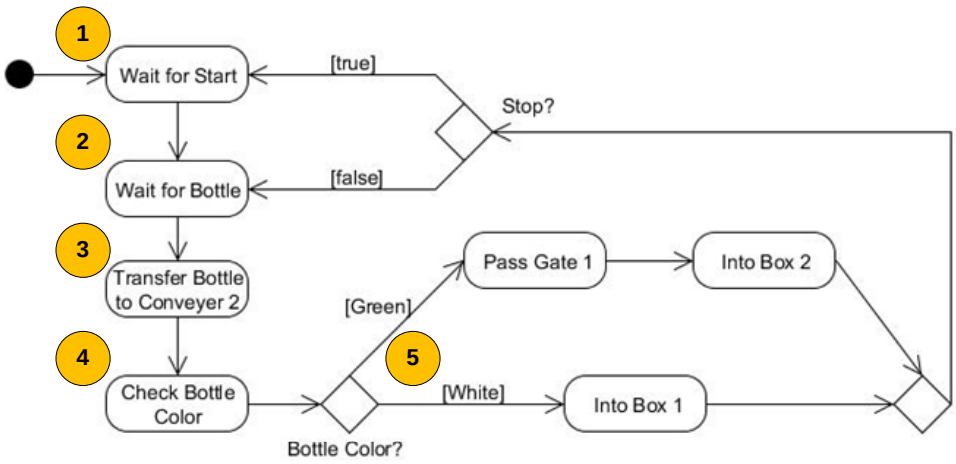


Figure 4.3: Activity Diagram of the bottle sorting machine.

4.4.2 Architecture and Structural Aspects

In industrial automation systems there are several (distributed) components involved. Thus, it is necessary to identify related components and the interaction between components from an architectural point of view. Structural diagrams enable the visualization of the static system structure as well as the structuring of the system. Note that test cases cannot be derived directly because no behavioral information is available to test the sequences of steps. Nevertheless, structural diagrams are required to keep an overview of the system, determine test coverage, and prioritize test cases. Structural aspects help to identify the components involved in test scenarios and individual test cases.

Deployment Diagrams provide an overview of the physical layout of the system and identify communication paths between components of the system. While the strongest benefit of deployment diagrams is observed in distributed systems, where communication between (software-) components is essential. It is applicable from small systems up to complex systems by providing an overview of the hardware components. Figure 4.4 depicts a PLC connected via a bus system with the various actuators and sensors comprising the bottle

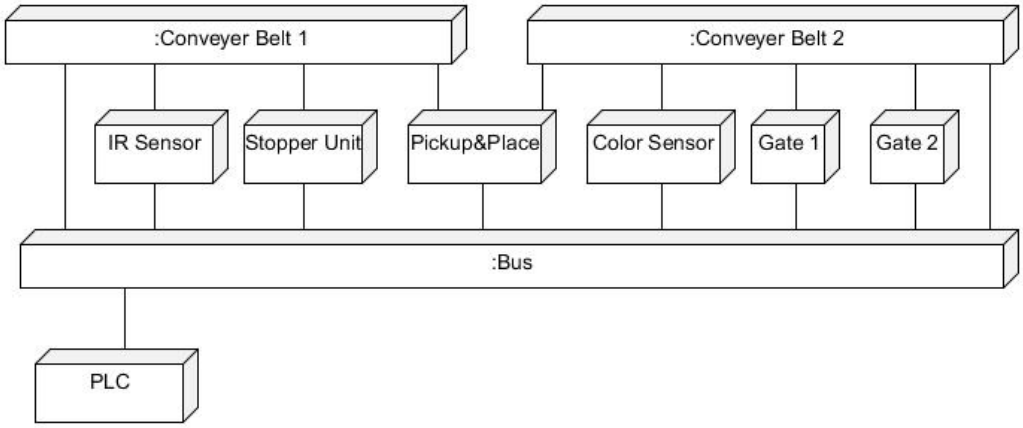


Figure 4.4: Deployment Diagram of the bottle sorting machine.

sorting machine. Note that the deployment diagram enables the identification of components and required mock-up functions, i.e. simplified simulation of non-existing components, related to individual test cases early in the industrial automation engineering process.

Component Diagrams provide a more detailed view of individual components and the interaction to related components. They focus on interfaces for a collaboration between different parts of the system. Such interfaces can be software, mechanical, or electrical interfaces. Figure 4.5 presents the components connected to a single conveyor and the logic component for controlling purposes. Component diagrams support the definition of test cases for a manageable part of the system in order to test interface definitions.

4.4.3 Definition of Functional and Temporal Behavior

The dynamic behavior of the system can be modeled and tested based on its requirements and the static structure of the system. Note that use case diagrams and activity diagrams are assigned to high-level behavior diagrams. Nevertheless, a more detailed view of the system (and individual parts of the system) is required to test these components in an effective and efficient way.

State Chart Diagrams describe the behavior of a system regarding the states and transitions in state-based systems. State chart diagrams depict the individual states of systems, sub-systems, and components and their relationship (transitions). Every state in a state chart diagram and their related state transition can be seen as an individual test case within a test scenario defined by activity diagrams and use case diagrams. Figure 4.6 depicts the behavior of the system as a whole. This diagram is used for generating test cases on system level. Testing on unit level or sub-system level, an emergency stop causes

all activities within the system to stop immediately independent of the individual states of all other components. To address highly complex and critical systems, individual sub-systems can be identified, based on high-level models, for a more detailed analysis and design and/or test. For instance, the stopper unit may be identified by stakeholders as a highly critical component of the system. In this case, the stopper unit is modeled in more detail. This state chart diagram is used to generate test cases on unit level or sub-system level, see Figure 4.7. Further information about test case generation from state chart diagrams is presented in Section 4.6.

Sequence Chart Diagrams capture the temporal sequence of events and communication between different parts of the system. Additionally, they give a detailed view on an activity within a system. Sequence chart diagrams are applicable to illustrate complex processes and critical (sub-)systems. Figure 4.8 presents a sequence chart diagram of the Pickup&Place unit behavior. It describes the transfer process of a bottle from Conveyor 1 to Conveyor 2. Note that every sequence of an activity can be used as a foundation for a (automated) test case generation, see Section 5.3.3. Based on the requirements and the static structure of the system all related components can be identified and tested.

Timing Diagrams depict the behavior of a system regarding time or a sequence of events. They are used to explore an exemplary run of a scenario at a lower level, i.e. more detail view, rather than sequence diagrams or state chart diagrams. Figure 4.9 presents the timing behavior of the Pickup&Place unit while transporting a bottle from Conveyor 1 to Conveyor 2. After finishing the task, the unit returns to the starting position (Ready State). The

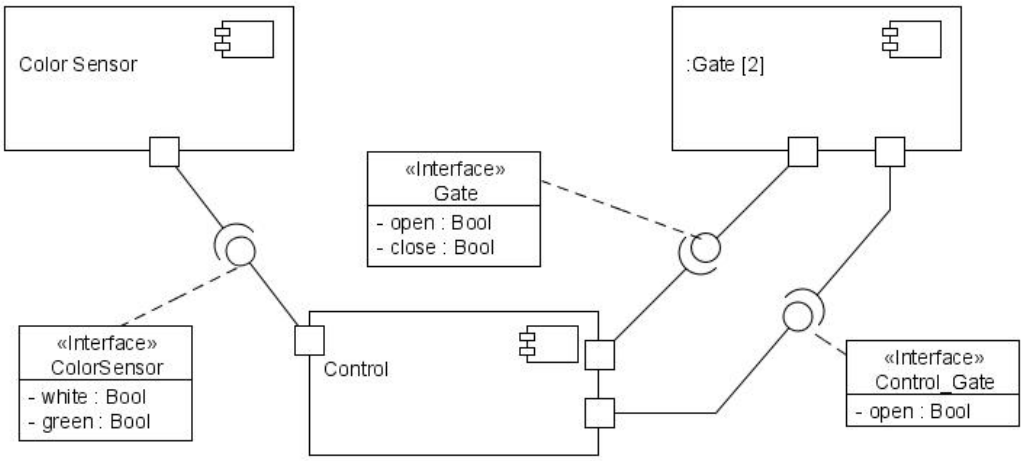


Figure 4.5: Component Diagram of the Conveyor 2 part excluding the Pickup&Place unit of the bottle sorting machine.

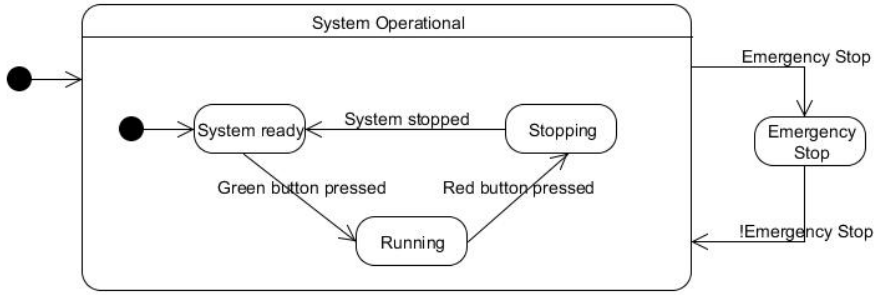


Figure 4.6: System level State Chart Diagram of the bottle sorting machine.

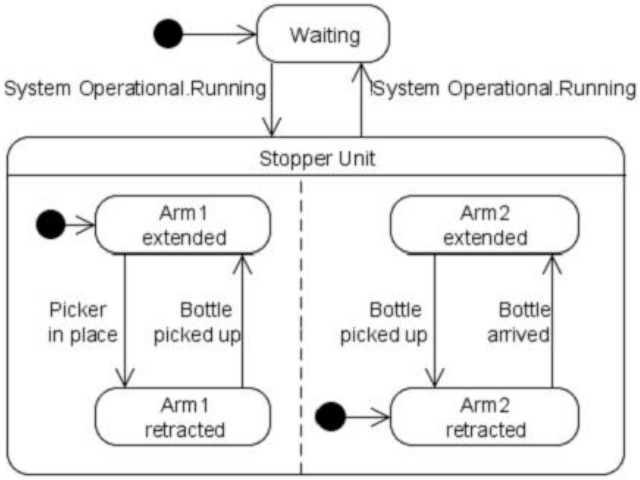


Figure 4.7: State Chart Diagram of the stopper unit which is part of the bottle sorting machine.

timing diagram addresses specific complex and critical aspects of the system to be modeled on a highly detailed level. For instance, the movement from `move left` to `move right` is measured in order to get the timing information to model the timing diagram. The main difference from timing diagrams to sequence chart diagrams is the absolute timing information in the diagram. Therefore it is easier to detect time overlapping from different components which can be considered during the testing. Nevertheless, modeling the overall system at the highest level of detail requires a considerable additional effort for modeling.

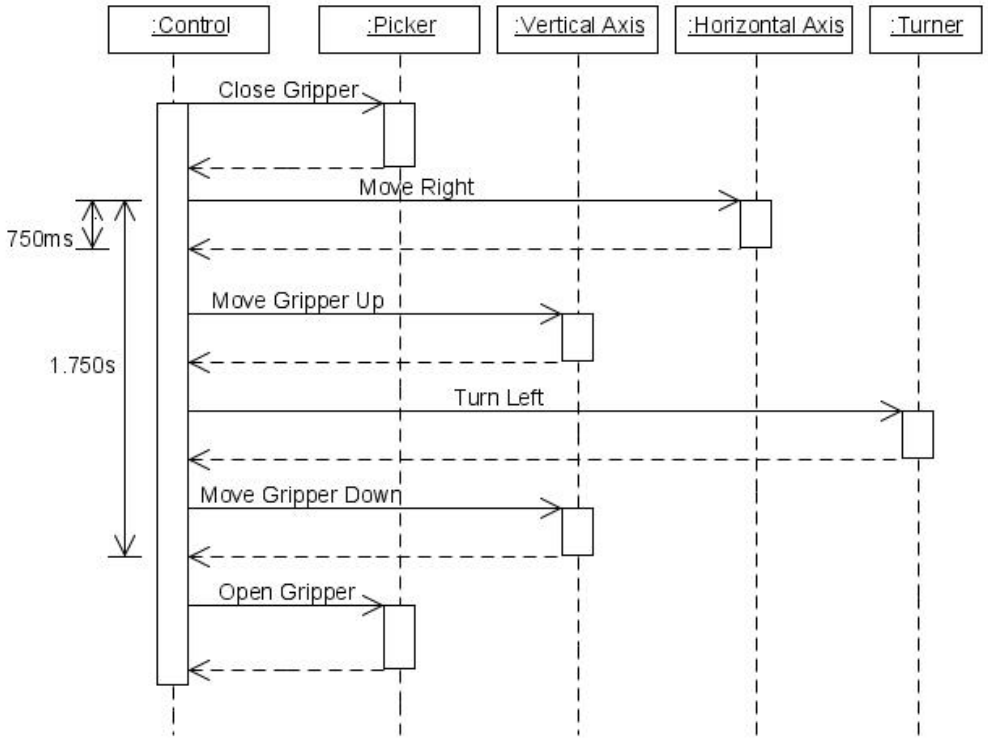


Figure 4.8: Sequence Chart Diagram of the Pickup&Place unit of the bottle sorting machine.

4.5 Selecting UML Diagrams

This section summarizes the results of the findings regarding the applicability of UML diagrams with respect to development phases and various test levels, see Section 3.2, applied for the industrial automation domain.

UML diagrams are analyzed based on their definition and the selection criteria with focus on the different phases of the industrial automation development process, based on [154]:

- (req) requirements definition phase,
- (sys) system definition,
- (arch) sub-systems architecture definition, and
- (unit) component and unit design which includes detailed design.

Table 4.1 presents a summary and highlights the most promising UML models based on the findings from the bottle sorting machine application. This

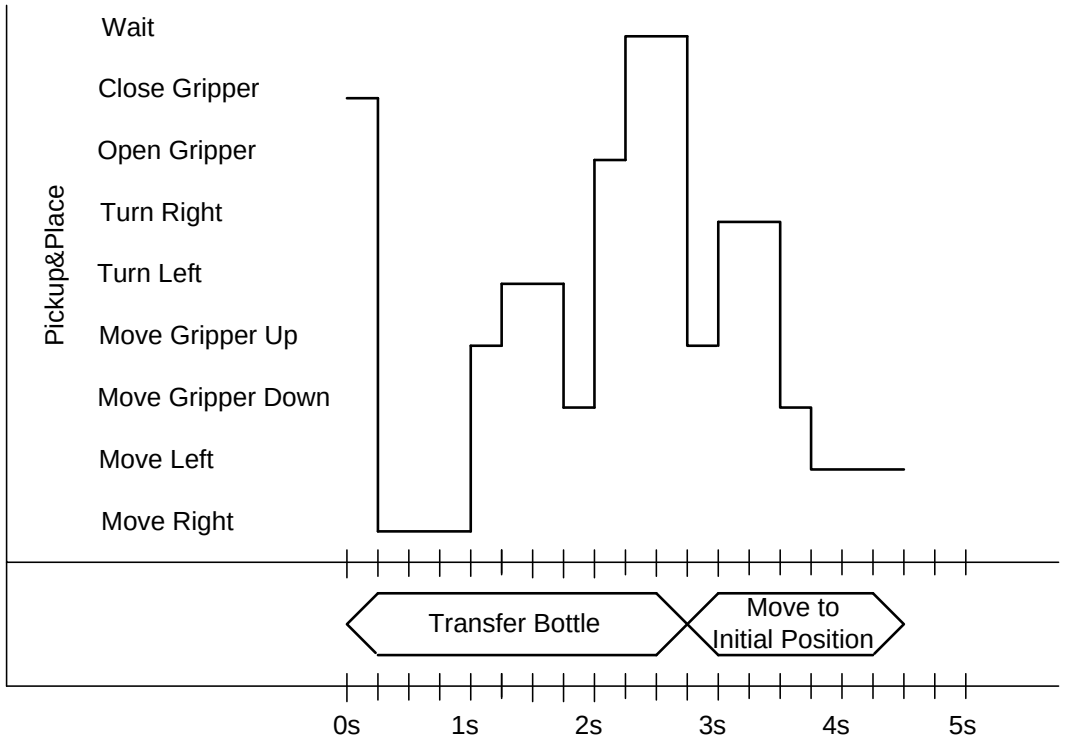


Figure 4.9: Timing Diagram of the Pickup&Place unit from the bottle sorting machine.

table includes UML diagram types, selection criteria, and individual development phases, i.e. (req),(sys),(arch), and (unit).

Experience gained during the development of the bottle sorting machine results that:

(a) Requirements modeling using use case diagrams and high level activity diagrams.

(b) Structure diagrams, such as deployment diagrams and component diagrams, are most reasonable to capture systems requirements and determine the static structure of the system.

Models for requirements (RE) enable the definition of test scenarios prior to more detailed design and implementation for TDD applications. Structural models (SA) enable the consideration of individual components and their interaction to implement TDD. Behavior models (BE) can be applied to directly derive test cases, e.g. in TDD, for implementation. Note that risks (RI) drive the level of detail and the test case generation process. Table 4.2 presents an overview of development phases, selection criteria, diagram types, and test levels. This table shows which diagram type is adequate for the different test levels.

Diagram	Criteria	(req)	(sys)	(arch)	(unit)
Use Case	(RE),(BE)	x	-	-	-
Activity	(RE),(BE)	x	x	x	-
Deployment	(SA)	-	x	-	-
Component	(SA)	-	x	-	-
Timing	(BE)	-	x	x	-
Class	(SA)	-	-	x	x
Composite Structure	(SA)	-	-	x	-
State Machine	(BE)	-	-	x	x
Sequence	(BE)	-	-	-	x
Communication	(BE)	-	-	-	x
Interaction overview	(BE)	-	-	x	x
Object	(SA)	-	-	-	x
Package	(SA)	-	-	-	-

Table 4.1: UML diagrams in the industrial automation software engineering process, based on [154]; The highlighted diagram types are most applicable for the industrial automation domain.

Phase	Criteria	Diagram	Test Level
Requirements Definition (req)	(BE),(RE) (BE),(RE)	Use Cases, Activity	System Testing / Acceptance Testing
System Design (sys)	(SA) (SA) (BE)	Deployment, Component, State Chart	System Testing / Integration Testing
Sub-System Architecture Specification (arch)	(BE) (BE) (BE)	State Chart, Sequence Chart, Timing	Integration Testing
Unit and Component Specification (unit)	(BE) (BE)	State Chart, Sequence Chart	Component Testing / Unit Testing

Table 4.2: Overview of development phases, UML model selection criteria, relevant diagram types for industrial automation, and defined test levels.

While investigating the application of the models in context of testing, three test levels are considered which have been presented in Section 3.2. Additionally, risks (RI) in all phases of the development and results in defining risk-specific views with focus on more critical aspects of the system are addressed. Depending on the risk, a more detailed view on parts of the system is a candidate for more detailed models and in-depth testing.

Regarding the systems dynamics, state charts and sequence charts are the most reasonable option. State chart diagrams illustrate the relationship between individual system states, also including possible parallel states, e.g. emergency stops, which are common scenarios in industry settings. Additionally, sequence chart diagrams enable a focus on a specific (most critical) part of the system to identify individual sequences of steps of the system and/or critical and more complex sub-systems [154]. The use case diagrams and activity diagrams focus on requirements (RE). Note that the package diagrams organize the models and are not connected to the system structure or behavior [154]. Diagram types such as class diagram, composite structure diagram, communication diagram, interaction overview diagram, object diagram, and the package diagram are not suitable for the test specification of SuT in the industrial automation domain.

In contrast to traditional, late and isolated testing approaches, the individual selected models enable early test case generation on defined levels of detail based on models [119]. Major benefits are:

- An increased understanding of the system under development and
- early test cases applicable in various phases of the development within heterogeneous engineering environments.

4.6 Test Case Extraction From State Chart Diagrams

Semi-formal models such as state chart UML diagrams, which are selected for industrial application specification from the UML diagram family, are suitable for the test case specification, see Section 4.5. This diagram type enables behavior specification of the SuT which is most important for testing industrial systems. One of the most time consuming and most critical part in the testing process is the test case extraction and generation. The term “extraction” describes the process of extracting the information from the test specification which is needed for generating test cases.

State charts are commonly used in the field of software engineering for specifying the software behavior and testing code. Chow [30] describes a method for checking the correctness of control structures at the design level by FSM. Swain et al. [98] use the combination of UML state chart diagrams

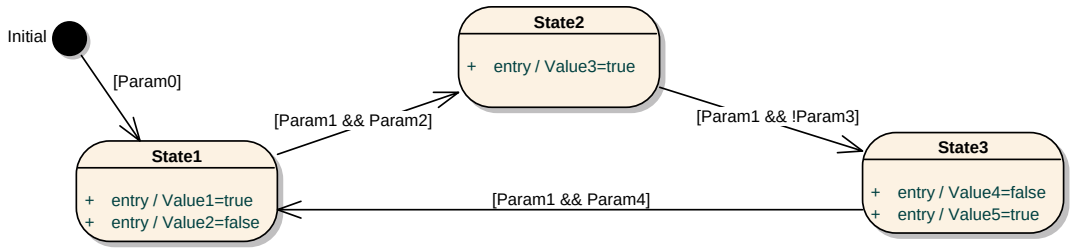


Figure 4.10: Example of a State Chart Diagram. Information of the state parameters and condition parameters of the transitions are used for test case extraction.

and activity models named state-activity-diagram for the test case specification and generation. The combination of state chart diagrams and activity models provides both control flow and event-oriented state change information.

4.6.1 Test Specification with State Chart Diagram

The state-based way of thinking for automation engineers for developing PLC programs have been common over several decades. Frey et al. [156] state that state chart diagrams are also called automata are a useful method for specifying non-complex behavior and implementing automatic control, do testing, and verification for safety applications.

A *Moore-Automata* [3, 156] is used for the test specification and represents a test case or/and a test sequence. The test specification can be done in an early stage of the development phase by the development or test engineer and include the information for testing the SuT. Therefore the test information, i.e. pre-condition, action, and post-condition, is included in the state chart diagram.

The state chart diagram consists of finite *states* with only one start state. Figure 4.10 shows an overview of a simple state chart diagram definition. The graphical representation of a state is a circle or rectangle with rounded edges, see Figure 4.10: State1, State2, and State3. The initial state is a special state and is in most cases represented by a double circle or a black filled circle. *Transitions* are represented by arrows from the source state to the target state. A transition consists of one or several *conditions*, see Figure 4.10: Param0, Param1, Param2, Param3, and Param4. Such a condition is a Boolean expression or a continuous value expression of the input variable. Each state in automata defines the output variable. In safety applications only Boolean input and output variables are allowed. Continuous values can be represented as Boolean values by checking the boundary value [156].

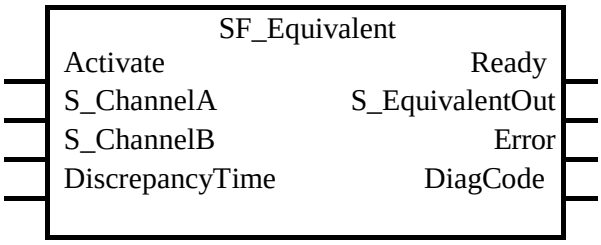


Figure 4.11: Interface design of the SF_Equivalent FB from PLCopen [157].

4.6.2 Test Case Extraction

Based on the state chart diagram in Figure 4.10, a test case is extracted with the following information:

1. *pre-condition:* Includes all the information, i.e. initial parameter values, at which the test case starts. Typically the parameters are included in the source state, e.g. `State1: Value1=true, Value2=false`. If a pre-condition is violated during the test execution, the test becomes undefined and results in failures.

2. *action:* These are the parameters which are used for the execution of a test case to the SuT. Typically the parameters are included in a transition and the parameter values are the conditions, e.g. `Param1 && Param2` from State1 to State2.

3. *post-condition:* These are the expected result values of a test case. Typically this information is included in the target state,
   e.g. `State2: Value3=true`.

An application to evaluate the demand of the test case generation for industrial software is used. A prototypical implementation of a safety FB from PLCopen [157] is used to show the example application of the test case extraction. Figure 4.11 shows the FB interface design of the PLCopen Safety SF_Equivalent FB. This interface of the FB, i.e. input parameter and output parameter are used for the test definition. The input parameters of the FB represent the condition parameters in a transition of the test specification in the state chart diagram, i.e. the action. The output parameters of the FB represent the state parameters of the test specification diagram, i.e. pre-condition and post-condition.

The SF_Equivalent FB converts two equivalent SAFEBOOL inputs to one SAFEBOOL output with Discrepancy Time monitoring. The main goal of this FB is to check both input parameters `S_ChannelA` and `S_ChannelB`. If one channel signal changes from TRUE to FALSE, the output immediately switches off

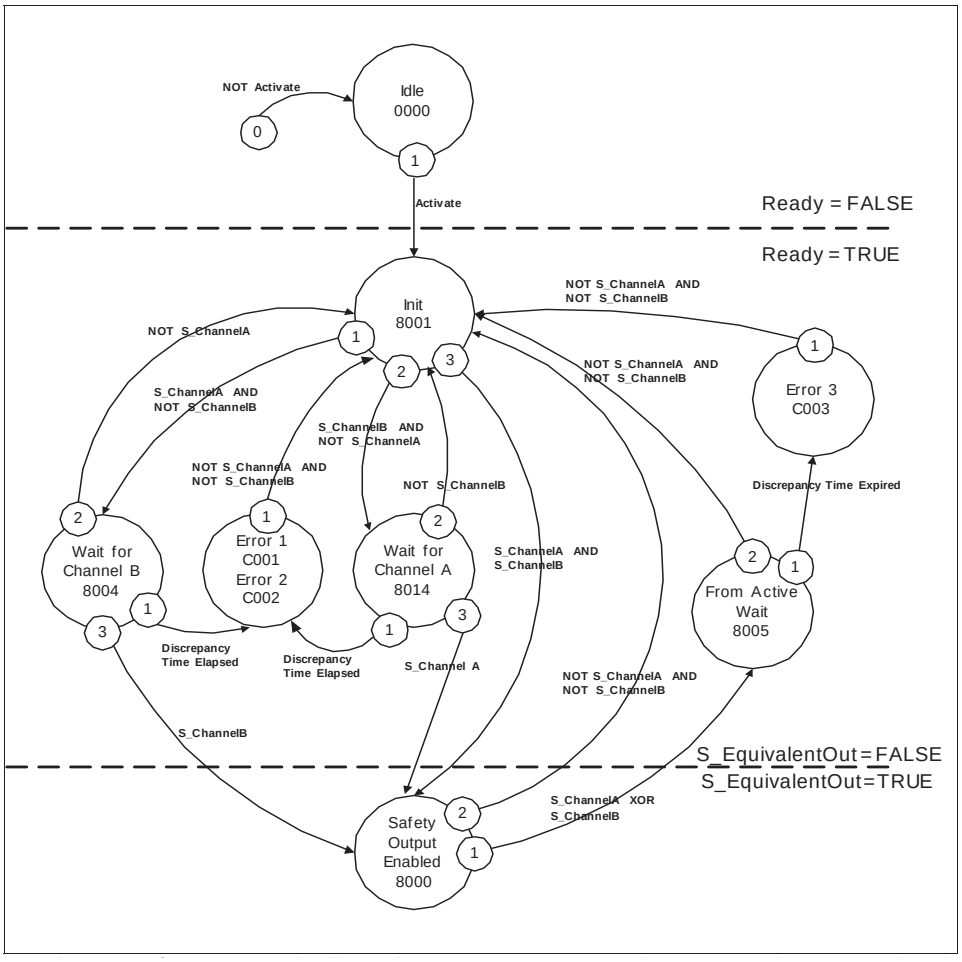


Figure 4.12: State chart diagram specification of the SF_Equivalent FB from PLCopen [157].

for safety reasons which is `S_EquivalentOut=FALSE`. The Discrepancy Time is the maximum period while both inputs may have different states without the FB detecting an error. The monitoring of the Discrepancy Time starts when the status of an input changes. The FB detects an error when both inputs do not have the same status once the Discrepancy Time has elapsed. The FB output shows the result of the evaluation of both channels. Figure 4.12 shows the state chart diagram from the PLCopen documentation [157].

The PLCopen documentation includes additional text information which is not included formally in the state chart model. Therefore, the test case generation of the PLCopen Safety SF_Equivalent FB is modeled in a new state chart diagram in the Enterprise Architect[14] tool which is shown in Figure 4.13. A similar implementation of the state chart is presented in [156], at which the

[14]http://www.sparxsystems.de visited: July 2013

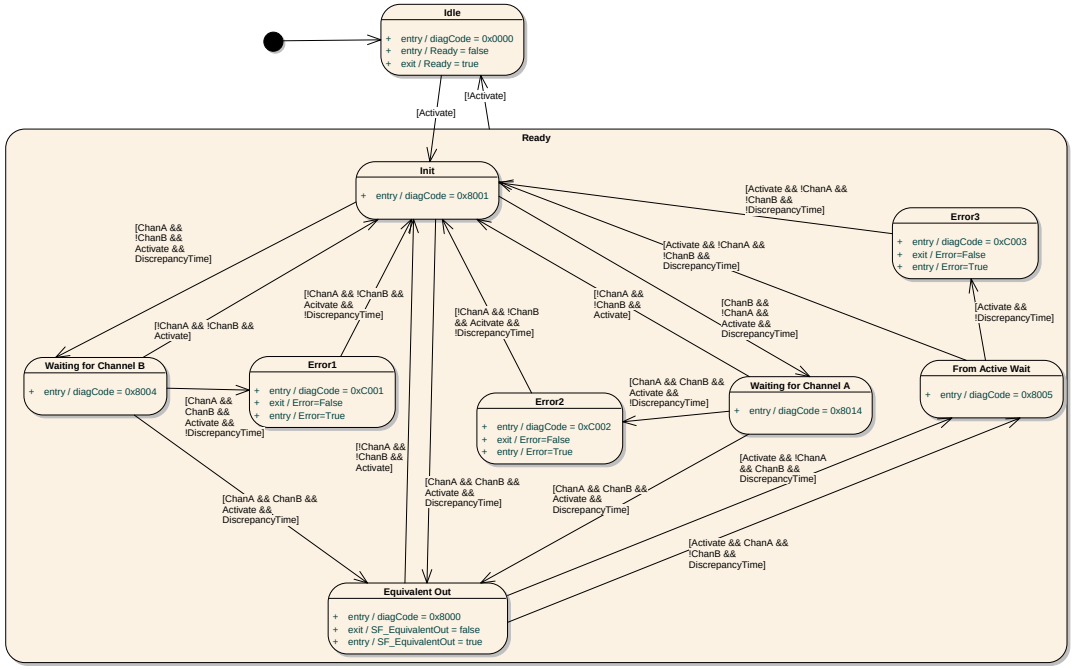


Figure 4.13: New state chart diagram specification of the SF_Equivalent FB.

implementation has additional priority parameters of the conditions and focuses on the Discrepancy Time in more detail. This model implementation is different to the PLCopen implementation. That means, that the state chart specification in Figure 4.13 includes only explicit transition definition. The Discrepancy Time is defined as Boolean because of the safety requirements. The `DiscrepancyTime=TRUE` means that the measured time is below the boundary limit otherwise the measured time is above the limit which means that the `DiscrepancyTime=FALSE`. Furthermore, the Error1 state and Error2 state are separated into two new states. Through these adaptions of the state chart diagram from the PLCopen specification, the test specification is unambiguous and an automatic test case extraction is possible.

In a first step the UML state chart specification has to be exported as a *.uml file. In a next step the file is read by the developed Java-based XML parser implementation. There all information of states, transitions, and conditions are extracted from the UML state chart specification. Based on this information the test cases can be generated automatically. The output of the test case extraction are single test cases which can be implemented in the control environment. An example of the output from an extracted single test case is presented in Listing 4.1. Further, an example of a test scenario from the state “Idle” to the state “Init” to the state “Waiting for Channel A” and then back to to the state “Init” is presented in Listing 4.2.

```
Pre:  State = From Active Wait
Action: Activate = true; ChanA = false; ChanB = false;
   DiscrepancyTime = false;
Post: State = Error3; diagCode = 0xC003; Error=True;
```

Listing 4.1: Output from a test case of the PLCopen SF_Equivalent.

```
Pre:  State = Idle
Action: Activate = true;
Post: State = Init; Ready = true; diagCode = 0x8001;

Pre:  State = Init
Action: ChanB = true; ChanA = false; Activate = true;
   DiscrepancyTime = true;
Post: State = Waiting for Channel A; diagCode = 0x8014;

Pre:  State = Waiting for Channel A
Action: ChanA = false; ChanB = false; Activate = true;
Post: State = Init; diagCode = 0x8001;
```

Listing 4.2: Output from a test scenario of the PLCopen SF_Equivalent.

Condition variables which are not used in every transition based on the source state are called *"free variables"*. There is the need to find such free variables for creating additional test cases, e.g. negative test cases. Test specifications with no free variables are the best case. Therefore the test specification is unambiguous for the test case extraction.

The path coverage testing criteria is used for the test case extraction from state chart diagram, see Section 1.3.2. Therefore the white-box testing criteria is used for the test case extraction and testing the behavior of a SuT as black-box testing, because the information of the internal implementation of the SuT is not used for the test case extraction but the separate test specification diagram.

After extraction and generation of the test information, the test cases can be implemented and executed on the target system, e.g. SoftPLC, Hardware PLC, etc. If the modeling of the test specification include failures the test cases will also include these failures which results in errors and the test fails during the execution on the target system. Therefore the test specification modeling should be done carefully to avoid test specification failures and avoid not executable test cases.

In further research the results and findings are used for the testing techniques of control software, see Chapter 5.

4.7 Summary

In this chapter a UML model selection for specifying industrial automation systems is presented. These selected model diagrams are applicable for specifying test cases for automatic control systems. During the implementation phase use cases and high level activity diagrams are identified which are most applicable for modeling requirements. Deployment diagrams and component diagrams are most applicable for capture systems requirements and determine the static structure of the system. Regarding the systems dynamics, state chart diagrams and sequence chart diagrams are well suitable. In order to specify tests for dynamic control systems, state chart diagrams and sequence chart diagrams are used for the further test case specification.

A Java-based XML parser implementation is realized to automatically extract, based on the path coverage criteria, test case information from UML state chart diagrams. Based on this criteria all possible control flow paths through the SuT specification is considered. By using the knowledge of the selected model diagrams for test specification as well as the results of the test case extraction from the test specification, appropriate testing techniques for testing control software are investigated. In the next chapter such testing techniques for industrial automation systems are introduced.

CHAPTER 5

Testing Techniques for Industrial Automation Systems

Due to the potentially complex behavior of software components, complete test specifications also can be extensive. In the following, the testing techniques are separated into test case specification without a (semi)-formal model and test case specification with (semi)-formal model specification. First, the manual testing and the Keyword-driven Testing (KDT) are testing methods without a (semi)-formal model specification for testing industrial automation software. This means that the test case specification is done in textual form. Then testing with semi-formal model test specification is presented. Here a testing technique based on UML diagrams for the industrial automation domain is shown which is required and suitable for fully-automated test execution.

5.1 Manual Testing

The manual test case extraction is completely operated manually and was the earlierst style of testing, but it is still widely used [77, p. 20-22]. The test case specification has to be done manually by textual description or in a more formal way like UML diagrams. Furthermore, by using this approach the extraction of test case information from test specification and requirements specification has to be done manually. The manually constructed test cases for further processing have to be included in the SuT, for instance in the control application [26, 142]. The test result analysis have to be done manually as well. Manual test case construction, implementation, and modification require appropriate programming skills. Additional programming skills are also needed if the test requirements are changed. Furthermore, changing the test require-

ments will result in modifying the test case implementation which results in a high maintenance effort.

The manual testing technique is time-consuming and does not ensure systematic coverage of the SuT functionality. In fact, the cost of manual test execution is so high that it is often necessary to cut corners by reducing the number of test cases that are executed after each evolution of the SuT. This can result in software which is incomplete tested, has significant risk regarding product maturity, stability, and robustness [77, p. 20-22].

5.2 Keyword-driven Testing

Keyword-driven Testing (KDT) [158], also named as Table-Driven Testing or Action-Word Testing, is a software testing method to support manual and automated testing processes in the field of software engineering. This technique is an extended form of the Data-Driven Testing method.

KDT can be seen as a test language [158] and improves the communication between testers, avoids inconsistencies in test documents, and makes an infrastructure for test automation available. Zylberman and Shotten [158] divided the KDT process in two main layers–Infrastructure Layer (KDT Engine) and the Logical Layer (KDT Test Case). The KDT framework in this thesis is based on such layer design, see Figure 5.1. KDT is used as a testing technique in the field of traditional software engineering [159] but it is a novel approach to use this method in the field of testing industrial control software [48].

The main concept of this testing technique is to express each test case as abstractly as possible while making it precise enough to interpret and execute the test case via a test tool [48, 77].

KDT focuses on a higher abstraction level of individual test cases, i.e. based on customer specifications. Also it is decoupled from traditional software testing approaches, i.e. implementing unit tests on code level. The KDT approach aims at supporting engineers from different domains or in defining test cases more efficiently and effective from the user perspective without having highly sophisticated skills in software testing. Applying a small set of keywords enables non-programmers to read and write test cases without knowing details of the implementation behavior of the SuT [13, p. 115]. Additional benefits come from applying high-level tests, e.g. on systems, acceptance tests, or integration tests, which focus on customer and systems requirement without considering concrete implementation approaches. Automation support of test case execution enables regression tests, i.e. test runs after changes that become applicable during systems maintenance and evolution and can be adjusted to change requirements.

The KDT technique separates the task of the test case implementation from the software development, i.e. separation of code implementation work and

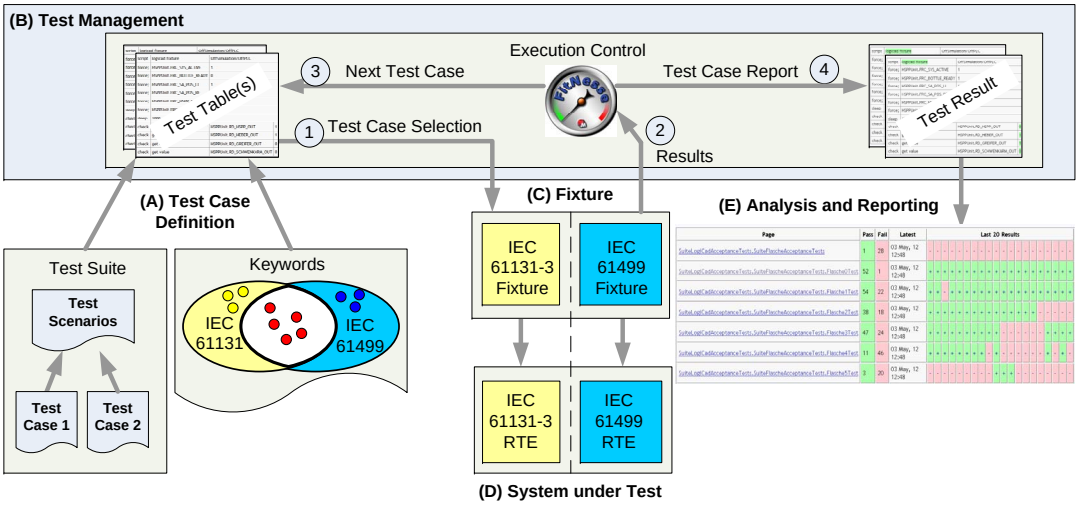


Figure 5.1: Overview of the test framework for Keyword-driven Testing [48].

test case definition and generation. Note that test cases can be defined early during a TFD approach. As the name KDT implies, "keywords", which control the processing, are in the focus of this approach. These keywords are dedicated in functions and in group of functions which include the executed actions of the test cases. Therefore, one keyword is able to include one or more test steps [160]. Once defined keywords can be used by the test engineer who write the test description for the SuT.

5.2.1 Test Framework for Keyword-driven Testing

Figure 5.1 presents an overview of the test framework based on the framework in Figure 3.3. The framework in Figure 5.1 is extended for KDT and consists of keyword-driven test case definition, execution, and reporting. The main difference between the general test framework in Figure 3.3 and the KDT framework is the additional keyword definition which is used for the test case definition. Furthermore, the Diagnosis Unit for the monitoring of all watched values is not used in this case. The framework includes five different building blocks based on [48]:

The test suite (see the defined structure of a test in Figure 3.2) is based on customer specifications. Individual test cases and test scenarios are constructed manually by applying the defined keywords (see Section 5.2.2) and test data. Each test suite consists of defined test cases and test scenarios which are easy to configure and edit in the Test Management system. Note that models, e.g. as stated in [113], assist automation-supported generation of test suites including test scenarios.

Individual test scenarios and test cases are selected based on value contri-

bution of corresponding requirements and will be executed by the corresponding test fixture, see item (1) at Figure 5.1. Every test configuration requires an appropriate test fixture related to the implementation standard. The fixtures of IEC 61131-3 and IEC 61499 are different at which the interface connection to the SuT is implemented in detail. Item (2) presents the results of one test run, which are collected by the *Execution Control* component of the Test Management system.

The Execution Control component included in the Test Management System is responsible for coordinating and controlling individual test runs (see item (3)) as well as for preparation of test result analysis, see item (4). Based on the results, an analysis and reporting functionality is available for more detailed investigations, e.g. test coverage measures, quality metrics, and project observations.

5.2.2 Keyword Specification

The KDT method supports readable test cases by using specially defined keywords. These keyword-commands are used for executing a test case, e.g. set value or get value. The test case description by the KDT technique should be defined as formal as necessary so that test engineers can easily understand and comprehend the test specification. Furthermore, the test cases should be executed automatically.

An example keyword-command definition with two parameters is formulated as:

```
|KEYWORD;|Parameter1|Parameter2|
```

There is the need for a number of pre-defined keywords at which the keyword vocabulary should be framework independent. This means that the keyword definition and specification should not be related to or included in a part of the Test Management system. A separate interface between Test Management system and SuT has to be defined. This interface is called “Fixture”. The fixture consists of all keyword implementations related to the IEC implementation. In this case two different fixture implementations are needed for testing applications based on the industrial standards IEC 61131 and IEC 61499.

General required functionality of the keywords are:

- Build up and close the connection between the Test Management system and the SuT.
- Set values and send events, i.e. IEC 61499 events, from the Test Management system to the SuT.
- Compare resulting output values and events, i.e. IEC 61499 events, with the expected values and events.

Based on these general requirements, keywords for the testing process have to be defined. First, the keywords are defined for testing IEC 61131 applications which are used as basis keyword-set. This basis keyword-set is the minimum required keyword-set which is used for testing industrial automation systems [48]. In a next step the keyword-set is extended for testing IEC 61499 implementations. Most of the defined keywords are applicable for both standards, but testing IEC 61499 applications additional keywords are used because of the different kind of execution of IEC 61131 and IEC 61499 applications.

The following keywords are the minimum required keyword-set, i.e. basis keyword-set, for testing industrial automation systems based on [48]:

- *startConn*: establishes a connection to the target system - SuT (IEC 61131-3 or IEC 61499)
- *force*: sets a new value to the variable with force-flag. The force command is used in industrial automation to overwrite the value of a variable in a control application. This forced value cannot be overwritten by the application itself but rather by the Test Management system. The force command is used in order to avoid influences from the control application during an integration test or unit test without disconnecting data connection in the SuT environment.
- *set*: sets a new value to the variable without force-flag
- *sleep*: pauses the test for a number of milliseconds
- *get value*: reads the value from a specified variable
- *stopConn*: closes the connection to the target system and stops the application in the SuT
- *create resource*: creates an IEC 61499 resource, resource name is defined by a parameter in the test case specification (available only in IEC 61499)
- *create function block*: creates a function block with a specified instance name (available only in IEC 61499)
- *start*: starts the specified resource (available only in IEC 61499)
- *create watch*: creates a watch property for monitoring on an input/output variable, on a specified resource (available only in IEC 61499)
- *trigger event*: triggers an event of a function block (only in IEC 61499 available)

Keywords	*IEC 61131-3*	*IEC 61499*
startConn	X	X
force	X	X
set	X	X
sleep	X	
get value	X	X
stopConn	X	X
create resource		X
create function block		X
start		X
create watch		X
trigger event		X
script*	X	X
check*	X	X

Table 5.1: Overview of the used keywords for testing industrial automation systems, defined in [48] (*... internal keyword function of the Test Management).

- *script*: internal keyword function of the Test Management system in order to select the accurate fixture and additional target system parametrization for testing IEC 61131-3 and IEC 61499 applications
- *check*: internal keyword function of the Test Management system in order to compare the resultant value with the stated value to present the result with true or false

A combination of the basis keywords into grouped keywords is allowed which can be used as a new keyword. For instance, the grouped keyword `SystemStartUp` includes the following keywords:
`startConn`, `create resource`, `create function block`, and `start`.
Currently only a basis set of keywords is defined and used in Table 5.1, based on [48], which is applicable for testing IEC 61131-3 and IEC 61499 implementations.

5.3 Unit Testing Technique

The unit test specification methods strongly depend on the developer of the unit tests. A background of unit testing is explained in Section 3.2.1. Two main target audiences are distinguished. On the one hand there are the control application developers. They will develop and maintain specific FBs for their applications. On the other hand there are FB library developers who provide

general FBs to the control application developer. Their work is mainly focused on maintenance, refactoring, and functional improvement of FBs, which is an elaborate task [161]. As both groups will do their development work with IEC FB models. Taking this further, Baker et al. [21] state that the test specification should be done with the same means as the target application, in order to be effectively usable.

In the following, the unit testing concept is shown based on IEC 61499 and can also be used in IEC 61131. The test specifications should be, in this case, IEC 61499 models or derived models of IEC 61499 at best. Based on this, existing unit test frameworks and methods, e.g. JUnit, cannot be used because of in this case the programming language is restricted to Java. However, they can serve as a basis for the required features and functionality.

Based on the identified requirements, three potential methods for a unit test process for IEC 61131 and IEC 61499 are identified and explained in the following:

- Separated Test Component,
- Separated Test Specification, and
- Integrated Test Specification.

The main difference between these methods is the location of the test specification and test implementation relatively to the FBuT.

5.3.1 Separated Test Component

The first natural approach for a unit test framework is to apply unit test framework approaches from the software engineering domain like JUnit. There the tests for Java classes are specified in form of Java classes again which containing the test in the form of Java code. It can be shown that this is the way how software developers in the software engineering domain specify their unit test cases. When applying this approach to test cases for IEC 61499 FBs or IEC 61131 FBs, unit test cases should be implemented as separated and related test FBs. These test FBs would feature the mirrored interface of the FBuT, i.e. inputs will get outputs and vice versa. When performing the tests, both FBs would be connected and then the tests are executed. The whole development and test process can be done in the original IEC 61499 development environment, e.g. 4DIAC [162], or IEC 61131 development environment, e.g. CodeSys or logi.CAD, which would be natural for the automation control engineer.

With this approach all IEC 61499 / IEC 61131 FB language elements are available and can be used for the test implementation. In case of IEC 61499 applications, the ECC provides a powerful means for defining the test cases for

the stateful behavior of the FBuT. The combination of the ECC with several algorithms included in a FB can be used for specifying complex test procedures. Furthermore, commonly required test procedures can be provided in separate FBs. These can be reused in different test FBs in form of CFBs.

Well established testing frameworks, e.g. JUnit, automatically build test applications. In contrast to the software engineering domain, test applications in the automation domain need to be developed manually and independently from the targeted automation applications. This test application needs an IEC 61499 / IEC 61131 execution container, e.g. a resource, for deployment and test execution.

A disadvantage of this testing method is that the test is not directly linked to the FBuT. If the implementation of the FBuT is used for other applications and will be passed on, the test implementation may be lost. Because of the separated components, i.e. test FB and FBuT, it is difficult and time-consuming to maintain the changes in both FBs, e.g. interface changes in FBuT.

A further disadvantage is that IEC 61499 currently has no means for fulfilling all necessary requirements for test specification, test result interpretation, and test result reporting.

5.3.2 Separated Test Specification

The second approach which is identified is based on the model-based testing method commonly used in the field of software engineering [77]. The model-based testing method can also be adapted for testing industrial control components like IEC 61499 and IEC 61131 FBs. In this approach a separated test specification is used in the form of a software model which is derived from the system requirements specifications, and enables automated code and test case generation [77]. The aim of this method is to automatically implement executable test cases for IEC 61131 FBs or IEC 61499 FBs from their test specification.

Several UML diagram types are applicable for the test specification, see Section 4.5 for the selection of UML diagrams suitable for industrial automation domain. It is a powerful specification language and is widely used and accepted in the software engineering domain. However, it is a challenge to extract the information from the test specification to automatically generate executable test cases. A restricting UML profile could provide the needed unambiguity for the test case implementation. Similar to the proposed approach above (Separated Test Component), this FB includes all executable test cases. The ECC and algorithms in the ST language are used for the test implementation.

An advantage compared to the Separated Test Component approach is that interface changes of the FBuT can be handled more easily as the executable test

cases are generated automatically. This approach supports regular changes of the test specification, i.e. UML models. Hence, the effort for changing and maintaining the test specification in parallel to the behavior specification is reduced. Furthermore no programming skills are necessary for specifying and maintaining the unit tests which may increase the acceptance of TFD in the industrial automation domain.

One of the shortcomings of the Separated Test Component approach, the separated maintenance for FBuT and the related test specification, is also a drawback of this approach because the test specification is not directly linked to the FBuT. This approach is presented and explained in more detail in Section 5.4.

5.3.3 Integrated Test Specification

IEC 61499-1 defines a behavior specification method for SIFBs as well as for Adapters [55, p. 39] by so-called service sequence diagrams. Although this method is provided in IEC 61499-1 only for SIFBs and Adapters, IEC 61499-2 extends it to all FB types. In order to resolve this inconsistency, the second edition of both standards (IEC 61499-1 and IEC 61499-2) allows that service sequence diagrams may be specified for all FB types and sub-applications. Service sequence diagrams are well suited to specify the behavior of IEC 61499 FBs.

The sequence chart diagram is a powerful tool for specifying tests, e.g. UML sequence chart diagram. Service sequence diagrams allow the specification of sequences of input events and the resulting output events. Additionally, input data values can be used for FB test specification. Furthermore, IEC 61499 allows the specification of multiple service sequences, each describing a certain execution aspect of the FB. This can be utilized for defining several test sequences which will test different aspects of the interface behavior of the FBuT in a predefined order. With service sequences a mean is available that is rather similar to the general FB unit test description as defined in Equation 3.5. Furthermore, positive, special, and negative test cases can be specified and used for the testing process.

The main advantage of this approach is that the test specification is directly included in the type definition of the FBuT. Therefore the dedicated test specification is always part of the FB definition and thereby no inconsistencies between different specification versions can occur. Furthermore the test specification method is a native IEC 61499 method and so no additional training effort is required.

For FBs with complex interfaces, the service sequence diagrams can become extensive. On the one hand, service sequences help to focus on specific aspects of the interface behavior, on the other hand, the potentially large

number of test cases makes the assessment of the test coverage hard. Furthermore, similar tasks needed in multiple test sequences, e.g. bringing the FBuT in a predefined state, have to be included in each test sequence. In particular, IEC 61499 does not allow the invocation of a sequence diagram from another sequence diagram. Here extensions like sequence calls, loops, or conditions, which have been introduced to UML sequence diagrams could also be helpful for IEC 61499 service sequence diagrams in a future edition of the IEC standard.

5.3.4 Evaluation of the Three Listed Unit Testing Approaches

The identified requirements are fulfilled by the three listed approaches to a different extent. For the comparison of the presented approaches, seven criteria, based on Böhm et al. [51], have been identified to be relevant for the industrial automation domain:

- Flexibility in test specification,
- portability,
- readability,
- integrity,
- expressiveness,
- maintenance, and
- suitability for the IEC 61131-3 and the IEC 61499 standards implementation.

Table 5.2 presents the evaluation results for the seven selected properties. In the following, the results will be discussed in more detail.

Comparing the three presented approaches, it can clearly identify the Separated Test Component (S.T.C) approach as the one with the greatest *flexibility* regarding the test specification. This approach resembles current unit test approaches applied in the software engineering domain at best. UML diagrams are a powerful language but the needed restriction, i.e. UML Profile, for an unambiguous test case execution also limits the flexibility. With respect to flexibility, the service sequences in the Integrated Test Specification (I.T.S) approach are definitely the most limited and only suitable for IEC 61499 due to the definition in the IEC standard.

An advantage of the (S.T.C) and (I.T.S) approach is that the artifacts needed for testing, i.e. test FB and test service sequences, can be accessed by all

Table 5.2: Assessment of the presented specification and implementation methods for tests: Separated Test Component (S.T.C), Separated Test Specification (S.T.S), Integrated Test Specification (I.T.S).

Evaluated Property	S.T.C	S.T.S	I.T.S
flexibility in test specification	+	o	-
portability	o	-	+
readability	-	+	o
integrity	-	o	+
expressiveness	+	o	-
test maintenance	-	o	+
IEC 61131 suitable	+	+	-
IEC 61499 suitable	+	+	+

IEC 61499 compliant engineering tools. *Portability*, the exchange of engineering data, e.g. FBs, between tools, is an important property defined in the standard. However, exchanging UML diagrams between different engineering tools is problematic.

UML has been specified for the implementation of software. Therefore *readability* is an important aspect to efficiently gain a good insight in the components' behavior specification. Service sequences are easy to grasp. However, an overview of all test cases can hardly be gained for complex FBs. Test relevant properties, e.g. ECC, algorithms, and FB interface, are distributed within the test FB. Therefore, neither a single test case nor the full test suite can easily be grasped at first glance.

In the (I.T.S) approach the test related information is part of the type definition of the IEC 61499 FBuT. Therefore it cannot get lost or forgotten upon transfer. Limited *integrity* is offered by the Separated Test Specification (S.T.S) approach if the same UML model is also used for the specification of the FBs' behavior like proposed in [163]. A major drawback of the (S.T.C) approach is that the test specification and the implementation are completely separated from the FBuTs which may lead to inconsistencies.

All means for the specification and implementation of FBs can be used in the (S.T.C) approach. The (S.T.S) and (I.T.S) approach need a generic interpreter for the test specification because the test specification language is different from the FBuT implementation language. In order to allow an unambiguous interpretation of the test case specifications, the *expressiveness* has to be limited. IEC 61499 service sequence diagrams, as used in the (I.T.S) approach, only provide limited expressiveness. The limited expressiveness of service sequences can be overcome by potential improvements in IEC 61499, e.g. loops, sequence calls, and conditions. Hence, the Test Runner / Test Management system can easily reach an unambiguous interpretation of the specified test

cases.

Integrated modeling methods as proposed in the (I.T.S) and partly in the (S.T.S) approach, if the FB behavior is also specified in UML, enable an easy checking for inconsistencies between FBuT and the test specification, e.g. interface changes. This *test maintenance* tasks are harder to fulfill with complete separation of tests and FBuT as proposed in the (S.T.C) and partly in the (S.T.S) approach.

The (S.T.S) approach is presented in Section 5.4 in more detail to show the applicability for integration tests and system tests. The benefits of the (I.T.S) approach - portability, integrity, and the easy test maintenance - outweigh its disadvantages in unit testing. After assessing the evaluation criteria, the (I.T.S) approach has been chosen as a basis for the IEC 61499 unit test framework.

5.3.5 Used Unit Test Framework

Based on the integrated test specification the test framework is implemented as depicted in Figure 5.2. The tests in the form of service sequences are part of the FB-Type file, like all other FB type specific definitions.

The Test Management / Test Runner is implemented and included in the Eclipse based engineering environment 4DIAC-IDE [162, 164] as an additional open-source plug-in. The service sequences which specify the test cases and the FB interface are parsed by the Test Runner. The gathered information is interpreted as a set of test cases, each with an input vector and a set of expected output vectors.

Based on the interface of the FBuT, a type specific test application is automatically created and instantiated at the (remote) test device for the test procedure. The IEC 61499 compliant run-time environment FORTE [162,164] is used on the test device. The test application consists of the FBuT, FBs for the communication with the Test Runner (Receiver, Sender) as well as Multiplexing and De-Multiplexing FBs (MuX, DeMuX). MuX and DeMuX FBs are generic FBs, this means that the run-time environment is able to instantiate FBs with a variable number of event inputs and outputs according to the interface of the FBuT. The encoding of the events as data allows a synchronous transmission of the input set on a unidirectional connection from the Test Runner to the test device. Also the synchronous transmission of the output vector which is sent back from the test device to the Test Runner is facilitated [55].

For the communication between the Test Runner and the test device the connection-based TCP/IP protocol is chosen. The numbers of inputs and outputs of the sender and receiver functionality are derived from the interface of the FBuT. For the transmission of the encoded events an additional data input is added for the sender and an additional data output for the receiver respectively. The necessary interfaces are schematically shown in Figure 5.2c. In the

following the execution of the test procedure is shown:

1. The type specific test application is instantiated on the test device.
2. The Test Runner starts the evaluation of the specified test cases.
3. The Test Runner transfers the given input set of the test case to the test device.
4. The execution of the FBuT is triggered by the specified input event.
5. The resulting output events and data are transferred back to the Test Runner.
6. Finally, the Test Runner removes the test application from the device, i.e. clean up phase, analyzes the replies from the test application and evaluates the test results of the unit test.

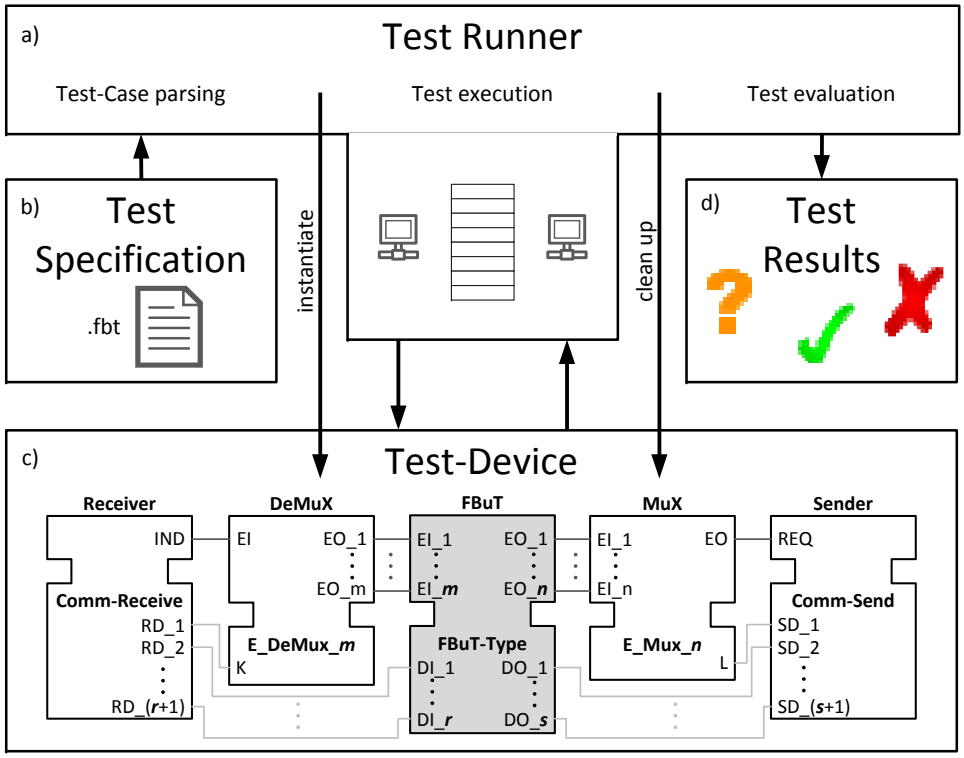


Figure 5.2: Overview of the function block test framework for unit testing. a) The Test Runner is the core element in the test framework concept. It handles the test case parsing, selection, execution, and evaluation. b) The tests are specified in the FB type file together with the interface and behavior. c) FB type specific test application is instantiated and deployed to a target control system. d) Test results are reported and visualized.

A specific test sequence passes the test if the set of the received output vectors $\boldsymbol{O}_r$ equals the expected output set $\boldsymbol{O}_e$

$$\forall S_{IO,s} : \vec{I}_{s,x} \mapsto \boldsymbol{O}_{r,s,x} = \boldsymbol{O}_{e,s,x}, \tag{5.1}$$

which is extracted from the test specification. The variable s represents the number of test sequences, see Equation 3.5, and the vector x represents the input-output set which is defined in Equation 3.4.

As soon as one of the following criteria is met, the given test sequence is evaluated as failure:

- The number of received output vectors is not equal to the specified number,
- a wrong event is received in an output vector, or
- the output data does not match the expected values.

Upcoming tests within the same test sequence are not evaluated any more, since pre-conditions might not have been fulfilled and therefore it cannot be sure that the test results for the next test cases are correct. After the execution of all tests, the test application is removed from the test device, i.e. clean up phase, and the test results are reported and visualized by the engineering tool, i.e. 4DIAC-IDE.

The separation of Test Runner and test device bears the following advantages:

1. The tests are executed on the deployed run-time environment. Errors and changes during the porting to new devices is tested.
2. Besides the FBuT, only standard FBs are needed for the instantiation of the test application on the test device.
3. Any kind of FBs (including SIFB) are able to be tested on devices that an implementation exists for.
4. The network based test execution enables also tests on platforms that do not provide visualization capabilities or file system access (for report saving). The test evaluation on the engineering system further reduces the need to provide memory to gather the test results on the test device. Thus tests can even be executed on small devices.
5. As long as standard compliant communication interfaces are provided by the devices, also tests on devices from various vendors are possible. The proposed test framework also consider the influence of different execution semantics on different target platforms which arise from different interpretations of the standard [42, 165].

6. The TFD approach is applicable with this testing technique because of the early test case definition and test execution as well as during the development phase.

5.4 Model-based Testing of Industrial Control Applications

Automation supported test case generation based on UML models supports flexibility of systems product development and enables immediate response to changing requirements. Furthermore, such models enable automated code generation of functional behavior and test cases. UML models represent a real world setting and are used as a foundation for automated test case and code generation [113].

A more comfortable way of getting a set of executable test cases for applications in automation systems development projects is the test case generation by using model-transformation methods [75, p. 33]. The difference to the test case generation approaches, which are presented in Section 4.6, is that this method uses a meta-model definition instead of the parser functionality. With the help of the meta-model definition, the test cases will be extracted from their specification by model transformation rules.

MBT is a well-known approach for test case generation in the field of software engineering [77]. Panjaitan and Frey [87] not only propose an approach for modeling IEC 61499 applications by UML but also for the entire development process of distributed control systems. In contrast, the approach in this thesis deals with the extraction of information for the automated test case generation with their execution environment and does not concern the generation of the system behavior itself from the UML specification.

In the following an automated derivation as well as an automated test case generation for industrial automation applications from UML diagram models is presented.

5.4.1 Test Specification Modeling

FBs are a common practice for modeling system behavior to encapsulate software components for reusable applications in the industrial automation domain. Based on the selection of suitable UML diagrams in Section 4.5 and in [87, 117, 154], state chart diagram definitions are used for the test case specification. Two different UML state chart diagram models are identified per control component [26, 142]: The first model is the Plant Behavior Model and the second model is the Control Behavior Model for the test case specification. Hegny et al. [166] use this approach for the generation of IEC 61499 applica-

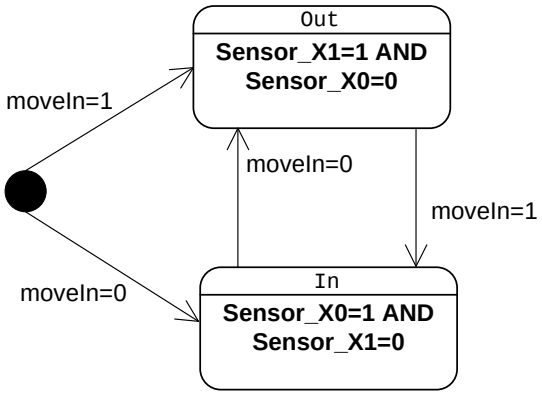


Figure 5.3: State Chart definition/specification by UML State Charts of a control axis behavior which moves IN and OUT [26, 142].

tions and simulate the plants behavior. Their concept of the simulation framework is based on the coupling of control application and the plant simulation. In contrast, here these UML state chart descriptions are used for the test specification of the industrial software .

The *Plant Behavior Model* describes the physical and mechanical behavior of a component or a part of a system, e.g. a mechanical axis. Additional timing information about the movements is included, such as the movement of axis from one position to the other may take 500 ms. The UML state chart model is adequately realistic like the real plant model. This specification is used for I/O tests such as mechanical, electrical, and pneumatic equipment. This allows to also test the basic control functionality.

The *Control Behavior Model* describes the logical behavior and defines sequences of a component and/or control system which have to be tested, see Figure 5.3. This model definition is used to test the logical implementation of the application.

5.4.2 Model-based Test Case Generation Process

The starting point of the MBT process is the UML specification of the SuT, and finally the output of this process is an executable FB network which is generated automatically. The FB network is based on the IEC 61131-3 or the IEC 61499 standard depending on the individual SuT implementations. At the beginning of the test process, the specification has to be done by using UML diagrams. UML state chart diagram models for the test specification are used. This UML specification model is the only one which has to be specified manually. The other models in this process are generated automatically. Figure 5.4a shows the overall process flow for the test case generation.

The test generation aims at covering all branches of the specified state chart

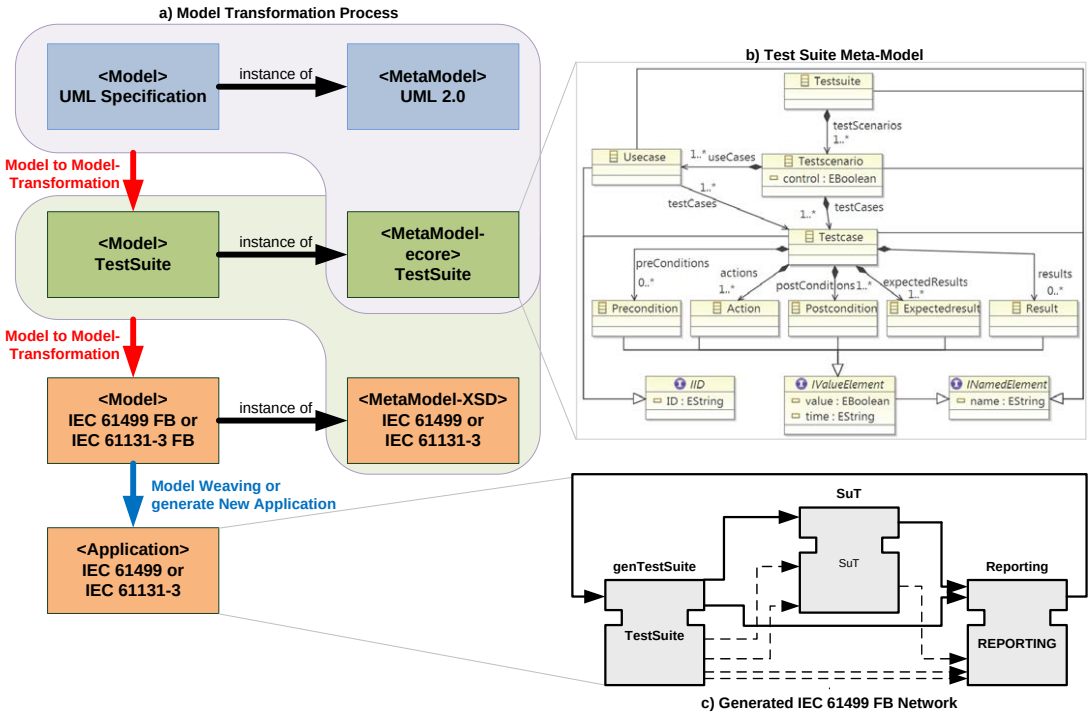


Figure 5.4: Overview of the transformation process to generate an automation control application including all test cases for the SuT a) Work-flow for automated test case generation; b) Test Suite meta-model definition, see Figure 5.5; c) Function Block network based on IEC 61499 standard [26, 142].

diagram. In order to reduce complexity, the path coverage testing criteria is used for the test case extraction from state chart diagram, see Section 1.3.2, so that each branch is visited once by evaluating each transition. The resulting test cases are formed into test sequences for testing the SuT.

A test case consists of pre-conditions, actions, post-conditions, expected results, and actual results. The schematic overview of the defined test suite meta-model is shown in Figure 5.5 which represents the structure of a test, see also Figure 3.2.

- The pre-conditions are parameters which are used for the source test case state.
- Then the parameters, which are defined as action parameters, will be applied on the SuT to execute the test case.
- The post-conditions define parameters so that test cases end in a defined state.
- The expected results are defined for the reporting process so that an actual theoretical comparison is possible.

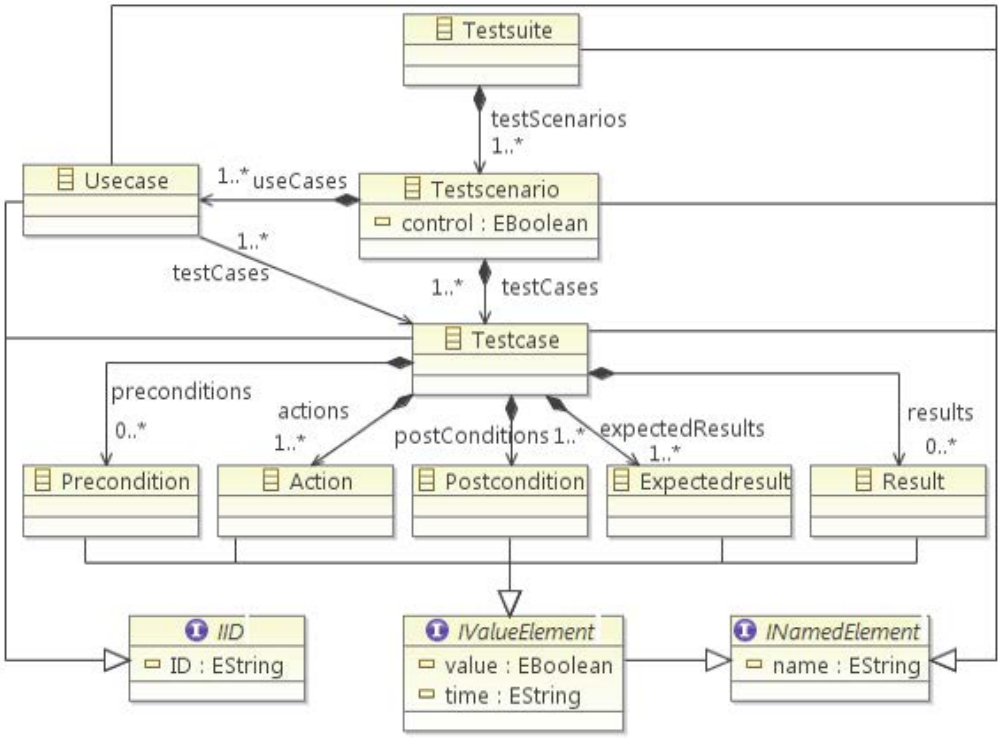


Figure 5.5: The defined <Meta-Model-ecore> TestSuite definition [26, 142].

In most cases, post-conditions and expected results have the same parameters. However, for safety-critical applications the expected results are not representing a safe state of the system. For such cases the post-conditions, which are different from the expected results, ensure a transition from the test result state into a safe system state where the test case is finished. This safe system state may be the starting point for other or/and next test cases. The post-conditions may have additional parameters compared to the expected results. Finally, these parameters will be used for the reporting process, for instance using online feedback [26, 142].

In the following the model transformation process is split in three main parts which is visualized in Figure 5.4a.

1. Generate <Model> TestSuite
2. Generate <Model> IEC 61499 FB or IEC 61131 FB
3. Generate <Application> IEC 61499 or IEC 61131

The transformation process generates the FB automatically, includes all test cases extracted from the UML specification also automatically which is used for the testing process.

Generate <Model> TestSuite: The defined <Meta-Model-ecore> TestSuite, the specified UML Specification, and the <Meta-Model> UML 2.0 [15] are required for the generation of the <Model> TestSuite which represents the basic structure of the test suite. The <Model> TestSuite is automatically generated through a M2M transformation and includes all test cases. A number of transformation rules for the model transformation process have to be defined first. These rules are defined in Xtend [167], a M2M transformation language provided by the Eclipse Modeling Project. Table 5.3 shows a detail overview of the defined rules for the M2M transformation from the UML diagram into the <Model> TestSuite.

The transformation rules for creating a <Model> TestSuite are split into three parts:

1. *Rules for creating a test case:* Based on the <Meta-Model-ecore> TestSuite definition the test case information is extracted and test cases are generated, see Table 5.3.

2. *Rules for creating a test scenario as use case:* The generated test cases are linked by using a predefined order to create test sequences based on the UML test specification model.

3. *Rules for creating a test suite:* The test cases and test sequences are ordered and included in the <Model> TestSuite.

Generate <Model> IEC 61499 FB or IEC 61131 FB: Based on the generated <Model> TestSuite, the <Model> IEC FB (<Model> IEC 61499 FB or <Model> IEC 1131-3 FB) is also generated automatically by using the M2M transformation process. The goal of this transformation is to generate the FB which includes the generated <Model> TestSuite. The same transformation language for generating the <Model> FB as the <Model> TestSuite generation is used, i.e. Xtend. The <Model> TestSuite, the <Meta-Model-ecore> TestSuite, and the <Meta-Model-XSD> IEC are required for the transformation process to generate the test suite included in the <Model> IEC FB.

The <Meta-Model-XSD> IEC are required for generating the industrial automation application network. IEC 61131-3 does not provide a general Meta-Model definition. Each IEC 61131 development tool uses its own Meta-Model definition. The used Meta-Model definition for the IEC 61131-3 implementation is based on the PLCopen specification [168]. The IEC 61499 standard organization provides a general Meta-Model definition. These Meta-Models include the structure definition of the automation model which is necessary for the transformation process.

[15] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML - free available, visited: March 2013

<Model> UML Specification	<Model> TestSuite
Rules for creating a test case	
name of initial state (Pseudostate)	Name of initial state, i.e. "plant" or "control", if plant is true then control is defined as false.
name of the state	used for naming the test case
source state (parameters)	parameters for the pre-condition
target state (parameter1)	parameters for the post-condition
target state (parameter2)	parameters for the expected results; if parameter2 is not defined, the expected results are equal to the post-condition parameters
transition (parameters)	action for the test case
-	automated generated identifier by a numbered consecutively, i.e. ID
Rules for creating a test scenario as use case	
source state and target state	name of the use case
all names of the test cases	name of test cases listed in a defined order
initial state name, i.e. control	name of initial state, i.e. "plant" or "control", if plant is true then control is defined as false.
target state parameters is equal to source state parameters from a different test case	such test cases can combined for a test scenario
Rules for creating a test suite	
name of the state chart	defined as name of the test suite

Table 5.3: Transformation rules for <Model> TestSuite. *-* ... no information is used for this transformation rule from the UML specification diagram.

The rules for generating the <Model> IEC 61131 are similar to the <Model> IEC 61499 but the <Model> IEC 61131 does not have an ECC. Therefore the <Model> IEC 61131 solves this challenge by using simple commands defined in the ST language, e.g. switch/case. Table 5.4 shows as example the transformation rules for generating the <Model> IEC 61499 FB.

The transformation rules for creating a <Model> IEC 61499 FB are split into two parts:

1. *Creating interface of the FB:* The interface of the FB is created including data/event inputs and outputs, based on the parameter from the UML specification. Additional data/event inputs and outputs are included for the FB parametrization, e.g. select test case or test sequence.

2. *Create State Machine (ECC) in the FB:* The internal ECC of the FB is created which is used to select the appropriate test case or test sequence during the test process.

Generate <Application> IEC 61499 or IEC 61131: The last step of the automated test generation is the creation of a test application. The generated FB, the SuT, and the reporting component are connected by its data- and event connections. If the test FB network is included in an existing application, the generated test network will also be connected with the SuT automatically. This process is called "model weaving". If this is not the case, a new application will be created which is done by a model transformation rule. The generated FB network is ready for the testing which is presented in the schematic overview in Figure 5.4c.

<Model> TestSuite	<Model> IEC 61499 FB
Creating interface of the FB	
name of the test suite	name of the BFB "genTestSuite_" + test suite name
-	create event inputs and event outputs, i.e. INITgen, INITOgen, REQ, CNF_TC, CNF_TS
-	create input data, i.e. QI, TC_Select, TS_Select, Mode_TC
-	create output data, i.e. QO, Status
action parameters	name of all action parameter are output data parameters
pre-conditions	one data output for pre-condition values, i.e. required for checking pre-condition state
post-conditions	one data output for post-condition values, i.e. required for reporting analysis
Create State Machine (ECC) in the FB	
-	create initialization part of the ECC
test case name	create an algorithm in the ECC with the test case name
test case	fill in the test case parameters, i.e. action, pre-condition, post-condition, into the algorithm
test case ID and use case ID and test case name	create state in the ECC and include ID in the state name, i.e. for a single test case: "TC" + TCID + "_" + test case name; for a test scenario: "TS"+ TSID + "TC" + TCID + "_" + test case name
-	generate the transitions in the ECC, input parameters of the FB TC_Mode, TC_Select, and TS_Select are used as condition parameters

Table 5.4: Transformation rules for the <Model> IEC 61499 FB. *-* ... no information is used for this transformation rule from the <Model> TestSuite.

5.5 Summary

In this chapter four testing techniques are presented which are classified in test specification with semi-formal model specification and without (semi)-formal model specification. Methods such as the manual testing and the KDT method consist of no (semi)-formal model test case specification. The tests have to be specified with a textual description. KDT uses the defined keywords for the test case specification in a formal way. This testing method is suitable for non-programmers in reading and writing test cases without knowing details of the implementation behavior of the SuT.

A semi-formal model test specification is used in the unit testing technique with sequence chart diagrams and in the model-based testing technique. Three potential methods, i.e. Separated Test Component, Separated Test Specification, and Integrated Test Specification, are identified for a unit test process. After a concise evaluation the Integrated Test Specification is presented in more detail for testing IEC 61499 FBs. Service sequence diagrams are defined by IEC 61499 and allow the specification of sequences of input events and the resulting output events. The sequence chart diagram is a powerful tool for specifying tests and the main advantage of this approach is that the test specification is directly included in the type definition of the FBuT. Thereby no inconsistencies between different specification versions can occur.

Model-based Testing is a well-known approach for test case generation in the field of software engineering. An automated derivation as well as an automated test case generation for industrial automation applications from UML diagram models is presented as the last testing technique in this chapter. By using Meta-Model definitions, the test cases will be extracted from their specification by model transformation rules. These transformation rules are presented and discussed to complete the model-based testing process for testing industrial control software.

In order to show the properties of the presented testing techniques, experiments, an evaluation and a discussion of results of these testing techniques is presented in the next chapter.

CHAPTER 6

Experiments, Evaluation, and Discussion of Results

A scientific evaluation of the proposed testing techniques from Chapter 5 require individual validation and analysis for the selection of the proper one. Therefore three application implementations are presented and an evaluation of these testing methods is discussed in order to show the benefits of the proposed testing techniques. A resulting comparison of the testing techniques is also presented.

6.1 Implementation and Evaluation of the Testing Techniques

In this section the implementation and evaluation aspects of the presented conceptual approaches from Chapter 5 are discussed. In order to show the benefits of the KDT method, the Pickup&Place unit is used, which is a part of the bottle sorting machine. The second part of this section presents the unit testing technique by using FBs from the IEC 61499 standard library. The last part of the implementation and evaluation section presents the implementation of an industrial sorting machine. The used applications are typical components from the field of discrete manufacturing and discrete production systems.

6.1.1 Keyword-driven Testing

A part of the bottle sorting machine is used in order to validate the presented KDT approach for industrial automation systems. A detailed explanation of the used machine is presented in [169] as well as in Section 4.4 for defining the

UML model selection. The selected part of the machine is the Pickup&Place unit which is depicted in Figure 4.1. The Pickup&Place unit is used to transport bottles from one conveyor belt to another conveyor belt.

The Pickup&Place unit is implemented in both industrial standards, with the logi.CAD tool[16] based on IEC 61131-3 and with the 4DIAC tool [162] which is based on IEC 61499. An appropriate comparison of the two different test case descriptions by KDT is carried out.

The tool called "FitNesse" as Test Management system infrastructure is used. FitNesse[17] is an open source testing tool based on FIT[18] (Framework for Integration Testing). It enables test case definition for acceptance tests and integration tests, i.e. system test level, as well as unit tests, based on keywords without requiring sophisticated skills in test case implementation [170]. FitNesse is a standalone WIKI[19] based on Java and is appropriate for communication and collaboration of engineers and non-technical users. Each page of this WIKI is a static WIKI page which is combined to test scenarios through a predefined order for executing the tests/pages. The properties of the specified tests can easily be changed by editing the page through using the included 'Properties' link (define a test scenario by several pages), see Figure 6.1. A test case itself is changed by the 'Edit' link. The test case specification itself is presented in Listing 6.1 for an IEC 61131 test specification and Listing 6.2 represents an IEC 61499 test specification.

These test case definitions are used for the communication between the Test Management system and the SuT by using the developed "Fixture" implementation. The SuT is able to run on two environments, once on the SoftPLC "logi.RTS" for IEC 61131-3 applications and once on the run-time environment "4DIAC-RTE" also called "FORTE" for IEC 61499 applications.

[16] http://www.logicals.com/, visited January 2013

[17] FitNesse: http://fitnesse.org/, visited: December 2012

[18] FIT: http://fit.c2.com/, visited: December 2012

[19] WIKI is a hypertext-system for a website which allows its users to add, modify, or delete its content via a web browser.

```
!|script|logicad fixture|${Testpath}|
|force;|HSPPUnit.FRC_SYS_ACTIVE|1|
|force;|HSPPUnit.FRC_BOTTLE_READY|1|
|force;|HSPPUnit.FRC_SA_POS_LI|1|
|force;|HSPPUnit.FRC_SA_POS_RE|0|
|force;|HSPPUnit.FRC_HSPP_POS_LI|1|
|force;|HSPPUnit.FRC_HSPP_POS_RE|0|
|sleep|1000|
|check|get value|HSPPUnit.RD_HSPP_OUT|0|
|check|get value|HSPPUnit.RD_HEBER_OUT|1|
|check|get value|HSPPUnit.RD_GREIFER_OUT|1|
|check|get value|HSPPUnit.RD_SCHWENKARM_OUT|0|
```

Listing 6.1: Example of a test case definition in FitNesse. Test case specification for the IEC 61131-3 KDT.

```
|script|forte fixture|${host}|${port}|
|startConn|
|create watch;|HSPPUnit|gruppe2_logik_0.HSPP_POS|
|create watch;|HSPPUnit|gruppe2_logik_0.BLOCKER_POS|
|create watch;|HSPPUnit|gruppe2_logik_0.GREIFER_POS|
|create watch;|HSPPUnit|gruppe2_logik_0.SCHWENKARM_POS|
|force;|HSPPUnit|gruppe2_logik_0.SYS_AKT|1|
|force;|HSPPUnit|HSPP_LOGIK.SEN_LI|1|
|force;|HSPPUnit|HSPP_LOGIK.SEN_RE|0|
|force;|HSPPUnit|SCHWENKARM_LOGIK.SENS_LI|1|
|force;|HSPPUnit|SCHWENKARM_LOGIK.SENS_RE|0|
|trigger event;|HSPPUnit|HSPP_LOGIK.INIT|
|trigger event;|HSPPUnit|gruppe2_logik.REQ_SYS_AKT|
|trigger event;|HSPPUnit|gruppe2_logik.REQ_FL_BEREIT|
|trigger event;|HSPPUnit|HSPP_LOGIK.REQ_SENS|
|trigger event;|HSPPUnit|SCHWENKARM_LOGIK.REQ_SENS|
|check|get value;|HSPPUnit|gruppe2_logik_0.HSPP_POS|TRUE|
|check|get value;|HSPPUnit|gruppe2_logik_0.BLOCKER_POS|
   TRUE|
|check|get value;|HSPPUnit|gruppe2_logik_0.GREIFER_POS|
   FALSE|
|check|get value;|HSPPUnit|gruppe2_logik_0.SCHWENKARM_POS|
   FALSE|
|stopConn|
```

Listing 6.2: Example of a test case definition in FitNesse. Test case specification for the IEC 61499 KDT.

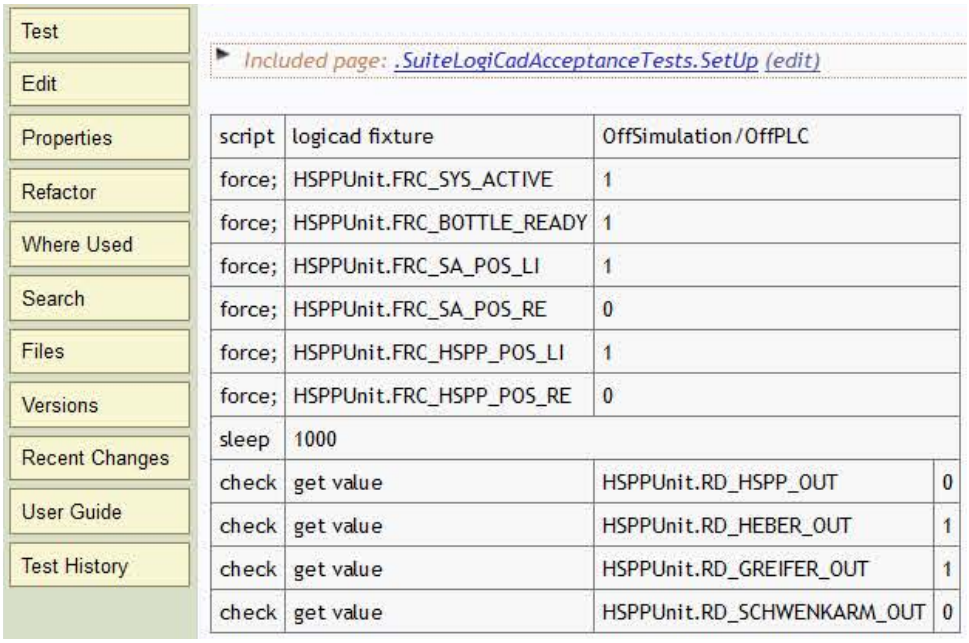


Figure 6.1: Graphical user interface from the FitNesse test case specification for IEC 61131-3 application based on [48]. Menu on the left side allows to start the test, edit the test specification, and support additional configurations of the Test Management.

Keyword-driven Testing of IEC 61131 applications: The communication between the Test Management system FitNesse and the IEC 61131-3 run-time environment logi.RTS PLC is implemented by using COM[20]-libraries on the local workstation, i.e. SoftPLC. This means that the testing process is executed not on a hardware PLC, i.e. in the case of IEC 61131-3. However, a separate hardware PLC is not necessary because SoftPLCs have the same execution behavior [48].

First of all, the SuT has to be run on the PLC or SoftPLC to test the implementation behavior. Figure 6.1 presents one test case specification of an IEC 61131-3 implementation. The test case specification is divided into four parts:

1. The initialization starts with the keyword `script`. Therefore the 'logicad fixture' is selected which is connected to the PLC-Resource named 'OffSimulation/OffPLC'.

2. The input parameters are set, which means the variables are forced with defined values. This is done by using the `force` keyword. Variables can

[20]Component Object Model (COM) is a binary-interface standard. It is used to enable interprocess communication, mostly used in Microsoft frameworks.

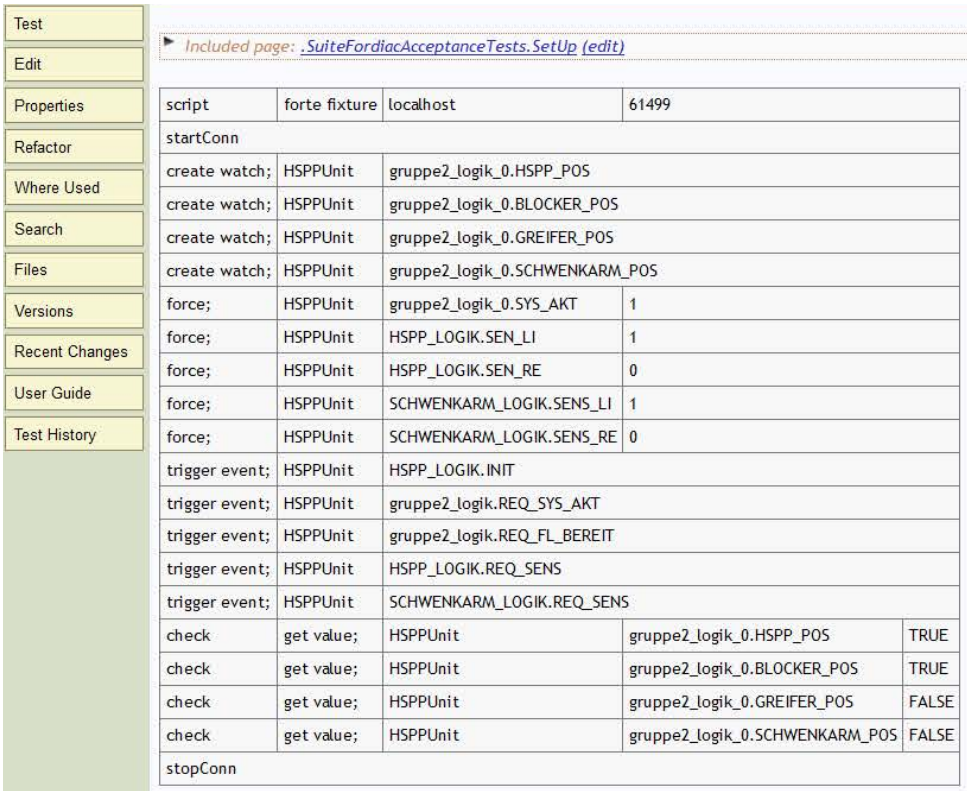


Figure 6.2: Graphical user interface from the FitNesse test case specification for IEC 61499 application based on [48]. Menu on the left side allows to start the test, edit the test specification, and support additional configurations of the Test Management.

only be forced if the IEC 61131-3 application has been extended by a so called "force marker", see Figure 6.3. Force markers are a logi.CAD specific feature to write values from external, i.e. outside from the PLC system, into the application. A main drawback is that these force marker objects have to be included in the SuT manually.

3. The `sleep` command has to be set. The Test Management system stops the execution by using this keyword comment, in this case the waiting time is set to 1000ms. This is necessary in IEC 61131-3 based testing processes because of the cyclic execution. It guarantees that a state is reached and will not be skipped by the execution machine of the run-time.

4. In the checking phase the result value is read by executing the `get value` keyword and is than compared with the expected value by executing the `check` keyword. For instance, `HSPPUnit.RD_HSPP_OUT=0` is compared with the `get value` output.

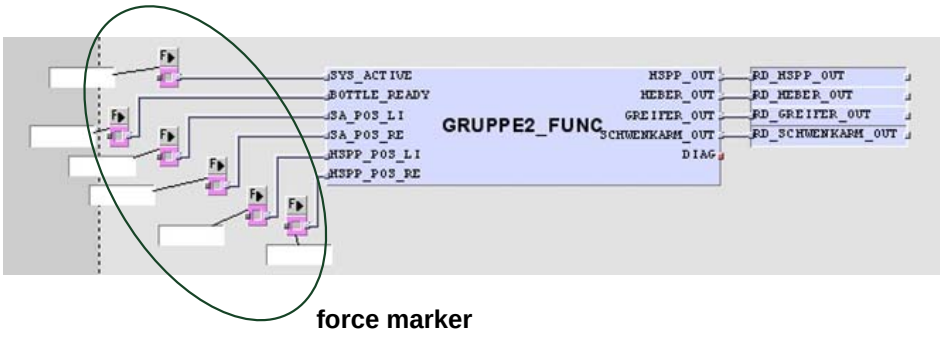


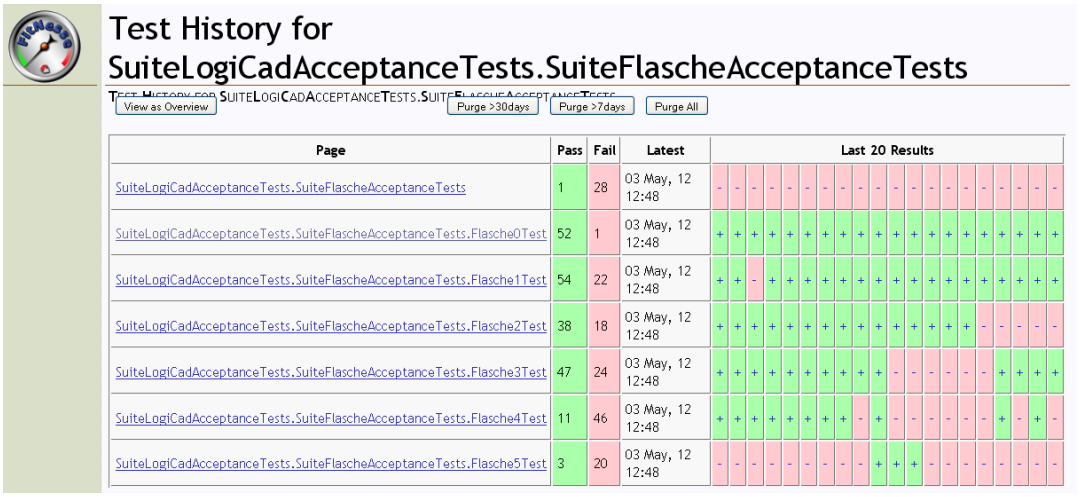
Figure 6.3: Part of an IEC 61131-3 FB network implemented in the logi.CAD development tool, which shows the required 'force marker' for writing set-values in the KDT process. In this case 6 force marker are required.

Keyword-driven Testing of IEC 61499 applications: The communication between the IEC 61499 fixture implementation and the IEC 61499 SuT applications utilizes TCP/IP communication, which directly enables remote testing. This enables testing of the SuT on several PLCs. That means, the KDT process is applicable for testing on real hardware PLCs. In contrast to the IEC 61131-3 test process, the IEC 61499 fixture has additional keywords defined, see Figure 6.2 and Listing 6.2. The `create watch` keyword enables to log variables of the IEC 61499 application which are used for checking the results. This feature is used for reading several variables which are watched at the same time. IEC 61499 is characterized by an event-based execution model. An additional `trigger event` keyword is implemented to trigger event inputs of FBs, i.e. SuT, which enable a separate triggering of each event input. Finally the `check` and `get value` keywords are used to compare expected and result variables, e.g. `gruppe2_logik_0.HSPP_POS=TRUE`, in the defined IEC 61499 resource named 'HSPPUnit'. The resource name has to be defined because IEC 61499 enables distributed control over several resources. This enables testing of distributed FB networks over several resources.

Before the test gets started, the SuT has to be downloaded on the runtime environment and must be in running mode. The implementation of the IEC 61499 fixture enables an automatic download of the SuT. This means that the IEC 61499 implementation is created on the 4DIAC-RTE, i.e. the application is running on the PLC, before the test process will be started. Three additional keywords are defined for this additional feature: `create resource`, `create function block`, and `start`.

As an additional help, a wrapper[21] has been developed which converts an existing IEC 61499 boot-file into a FitNesse WIKI-syntax. This boot-file con-

[21] Wrapper is an adapter implementation of a software part which encapsulates a small functionality of a software program.



Page	Pass	Fail	Latest	Last 20 Results
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests	1	28	03 May, 12 12:48	- - - - - - - - - - - - - - - - - - - -
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche0Test	52	1	03 May, 12 12:48	+ + + + + + + + + + + + + + + + + + + +
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche1Test	54	22	03 May, 12 12:48	+ + - + + + + + + + + + + + + + + + +
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche2Test	38	18	03 May, 12 12:48	+ + + + + + + + + + + + + + + - - - - -
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche3Test	47	24	03 May, 12 12:48	+ + + + + + + + + + - - - - - - + + + +
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche4Test	11	46	03 May, 12 12:48	+ + + + + + + + - + - - - - - - + - + -
SuiteLogiCadAcceptanceTests.SuiteFlascheAcceptanceTests.Flasche5Test	3	20	03 May, 12 12:48	- - - - - - - - - + + + - - - - - - - -

Figure 6.4: KDT - Test history visualized in the Test Management system FitNesse. Last 20 results are presented at which the results leftmost are the latest.

sists of IEC 61499 XML compliant commands. However, this wrapper only generates the test stub. An automatic conversion from control applications which are modeled in an IEC 61499 development tool, e.g. 4DIAC-IDE, into a WIKI-syntax, e.g. FitNesse Syntax, is possible by this wrapper. Only the actual writing and reading values for the test have to be inserted by the user. After that, the generated FitNesse syntax is used as a part for the KDT process, i.e. SuT. This additional feature enables a fast execution flow of the testing process [48].

KDT supports the continuous integration test strategy which supports automated frequent testing, see Section 3.1. In Figure 6.4 the test history is visualized, which presents an overview of the executed test cases and test scenarios. At any time, detailed information about the status of the SuT is available.

KDT is a valuable testing method for testing industrial automation applications. Based on the requirements for testing industrial control software, a basis subset of keywords is defined. The outcome of this is that IEC 61131-3 needs less keywords than IEC 61499. This has two reasons:

1. The IEC 61131-3 has a cyclic execution model and therefore no keyword such as `trigger event` is necessary.

2. The 4DIAC run-time environment (FORTE) enables a dynamic creation of FB networks which require additional keywords like `create resource` and `create function block`.

In order to validate the presented KDT technique, an industrial control application, a Pickup&Place unit, is used to show its feasibility. To sum up, testing IEC 61499 applications by the KDT method has more advantages because

the SuT does not need to be changed for the test process, whereas IEC 61131-3 needs additional force markers in the SuT implementation. Because of the standardized download interface of IEC 61499, the SuT environment is generated automatically by the IEC 61499 development tool, i.e. boot-file generation. By using the boot-file, an automatic WIKI-syntax is created by the developed wrapper generator, which enables an easier test process because of the generated FitNesse test template for the SuT.

This testing method is applicable for testing software units like IEC 61499 or IEC 61131-3 FB libraries because it is easy to connect the Test Management to the SuT, in this case to FBuT. An additional effort for IEC 61131-3 FBs is required by using force markers instead of IEC 61499 FBs. In the case of the IEC 61499 FBs test, the Test Management is able to connect to the FBuT directly, i.e. by using the IEC 61499 fixture. This testing method is also suitable for testing complex industrial applications, e.g. integration tests, system tests. For this purpose the test procedure is similar to the unit test.

6.1.2 Unit Testing with Service Sequences

For the validation of the function block test framework a selection of representative and comprehensible FBs is done. The selection of the FBs is performed on the basis of typical, i.e. most used in application development, execution features and FB properties that have to be considered for testing. A comprehensive taxonomy of the execution features and FB properties is presented in Figure 6.5. This taxonomy shows that FBs consists of data, events, and data types which have to be considered for testing.

As a result of the classification and evaluation process two well known FBs are chosen which are defined in Annex A of IEC 61499-1 [55]. The `E_PERMIT` and `E_CTU` FBs are apt to provide insight of the specification and execution of unit tests for FB.

Testing a Stateless Function Block - E_PERMIT: The `E_PERMIT` FB provides a permissive event propagation. As can be seen in Figure 6.6(a), the FB has one event input *EI* and an associated Boolean data input *PERMIT*. Since the behavior of the `E_PERMIT` FB is defined as stateless with respect to data and events, only two distinctive input vectors $\vec{I}_1 = (EI, false)$ and $\vec{I}_2 = (EI, true)$ exist. They differ in the value of the Boolean data input.

$$\vec{I}_1 \mapsto \boldsymbol{O_1} \quad : \quad \boldsymbol{O_1} = \emptyset \tag{6.1}$$

$$\vec{I}_2 \mapsto \boldsymbol{O_2} \quad : \quad \boldsymbol{O_2} = (\vec{O}_{2,1}) \quad : \quad \vec{O}_{2,1} = (EO) \tag{6.2}$$

Since the two test cases in Equation 6.1 and 6.2 are independent from each other, they are specified as separate test sequences. An additional test (Nega-

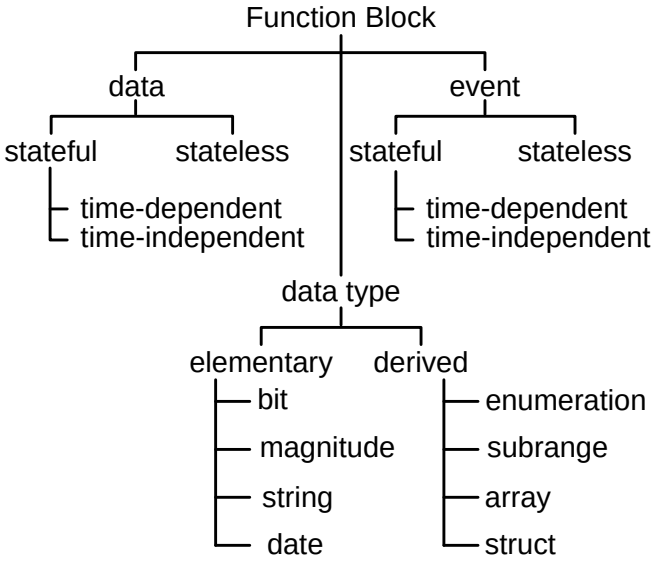
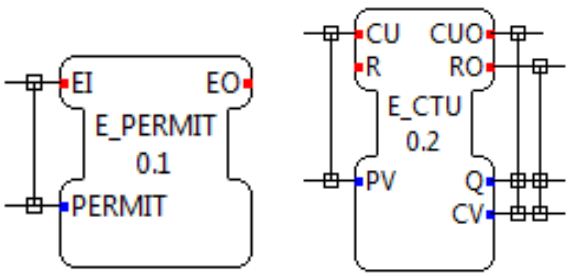


Figure 6.5: Taxonomy of observable function block execution features and properties.

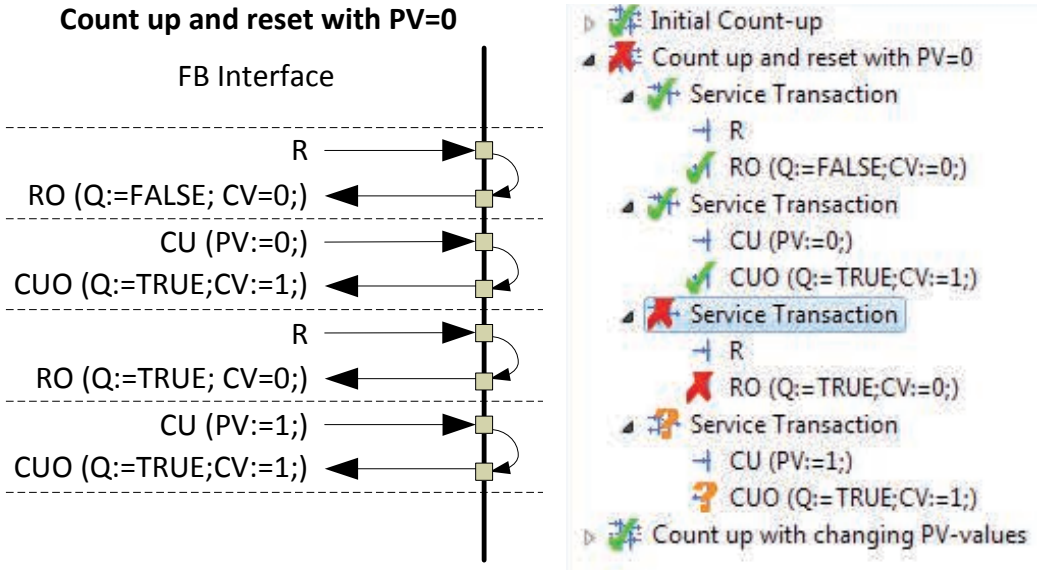
tive Test) is added which shall fail on execution to validate the functionality of the test environment.

The specification of the test cases is created with the service sequence editor of the 4DIAC-IDE, see Figure 6.7(a). All three mentioned test cases with the respective input and output vectors are visible in the figure. As the service sequences are included in the FB type definition and only IEC 61499 compliant description methods are used, any FB type editor may be used to extend or change the test cases. During the test process it can be required to edit the test specification because of late changes in the behavior specification, but it should not be the normal case by using the TFD strategy.



(a) E_PERMIT (b) E_CTU

Figure 6.6: IEC 61499 standard library FBs interface definition; (a) E_PERMIT - Stateless FB (b) E_CTU - Stateful FB.



(a) Test Specification with a Sequence Diagram (b) Test Results of the Unit Test Execution of 3 Test Scenarios

Figure 6.7: E_CTU FB: (a) Count up and reset with PV=0; Test specification with service sequence diagram (b) Test results; Test scenario 1 and 3 are executed successfully; Test scenario 2 failed due to test case 3 failed therefore test case 4 is not executed.

Testing a Stateful Function Block - E_CTU: The second chosen FB is the `E_CTU` FB. Its interface includes two input events (*CU*, *R*) and two output events (*CUO*, *RO*), as well as one data input (*PV* of 16-bit unsigned integer type UINT) and two data outputs (*Q* of Boolean type, *CV* of type UINT) as shown in Figure 6.6(b).

The counter value *CV* is increased at the occurrence of a count-up event *CU* and is reset to 0 on the occurrence of the reset event *R*. The behavior of the FB is dependent on the counter value *CV*. Therefore the `E_CTU` FB is stateful, because of the counter value is time-independent with respect to the counter value data, i.e. no timing information is stored in this FB.

The use of magnitude data types as input as well as output data highly increases the number of potential test cases, as stated in Section 2.7. There exist 2^{17} possible input vectors $\vec{I}_x$, i.e. 2 non-concurrent event inputs + 1×16-bit data input. Furthermore, this comprehensible example already has 2^{18} output interface states. Therefore, full testing of the `E_CTU` FB would require 2^{35}, i.e. $2^{17} \times 2^{18}$, test cases.

The requirement for a well-considered test case selection is evident. A representative set of test cases (including special tests and negative tests) has been specified for the `E_CTU` FB type, see Figure 6.7.

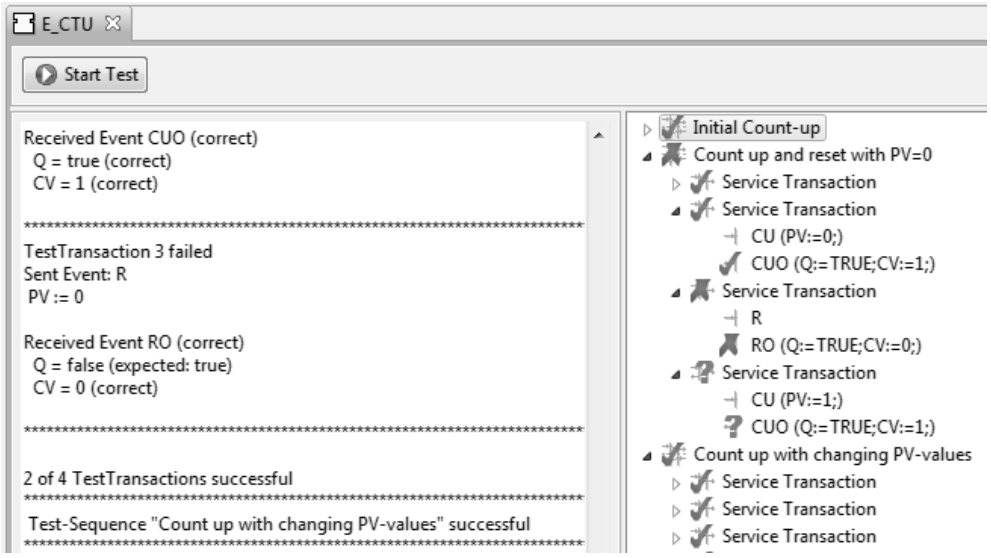


Figure 6.8: Visualization of the test results in 4DIAC-IDE; The left hand side demonstrates the test results in textual description, the right hand side pictures the test results in graphical form of the E_CTU FB, cf. Figure 6.7.

The test cases are parsed by the Test Management system, i.e. Test Runner, and run when tests are triggered by the user in the user interface, i.e. realized as plug-in in the 4DIAC-IDE. Within the same user interface, test results are reported both textually (showing detailed information on failed tests) as well as graphically (providing a good overview on test results), as presented in Figure 6.8. At the occurrence of the first error, see negative test, a test sequence is stopped and marked as failed, see cross mark in Figure 6.8. Further test cases within the failed test sequence are marked as untested, see question mark in Figure 6.7(b) and Figure 6.8.

This presented unit testing process is representative for a TFD approach. Because of the fast response of the test results, this unit test framework is suitable for TFD of industrial automation applications. FBs are identified as the software units to be tested in IEC 61499. Based on the semi-formal specification of their external observable behavior, a unit test specification method, i.e. service sequence diagrams, and an according unit test execution framework, see Figure 5.2, are derived. The main advantages of the presented solution are that it is fully IEC 61499 standard compliant and can be utilized in any IEC 61499 standard compliant environment. With two representative FBs taken from the IEC 61499 standard library, the usage is demonstrated and the applicability of the proposed approach is proven.

However, as expected the limitations of service sequence diagrams in case of FBs with complex interfaces are identified. The sequences get very large and only a subset can be efficiently specified. This selected subset has to be cho-

sen manually by the test engineer. A solution for such FBs could be to model the test cases with UML and automatically generate the according service sequences as input for the unit test framework. This has the advantage that the test cases can be developed more easily and that the tests are stored as part of the FB. But this reduces the effort only for the initial test case generation. The same effort is required for maintaining the test cases during the life cycle of the FB.

To overcome the limitations of the presented approach a set of extensions to the IEC 61499 service sequences are required that reduce the effort for specifying tests as well as FB behavior by increasing their expressiveness. A first step will definitely be the introduction of sub-service sequences and calling mechanisms. By doing so, common tasks needed for several test cases. This is useful for the FB preparation which can be defined once and reused afterwards for other test sequences. Further required IEC 61499 extensions are control structures, conditional execution, and loops in order to implement repetitive test sequences. Finally, as control applications are real-time constrained applications, the timing behavior is an important property and quality feature. Therefore it should also be possible to specify the timing behavior in test sequences.

6.1.3 Model-based Testing

For the validation of the MBT technique a representative prototype implementation of a sorting machine is selected to demonstrate this approach for the field of discrete production systems. The basic functionality of the sorting machine is the transportation of palettes via a conveyor belt. After that, the parts are sorted from the palette into two boxes, depending to the color of the bottles. The sorting part is done by two pneumatic axis that act as handling units, a vacuum gripper, and a conveyor belt. Figure 6.9 shows a picture of the sorting machine.

UML diagrams are used for the test specification of the control and plant behavior. In this case, UML state chart diagrams include the required information for the test case and the test sequence generation. As presented in Section 4.6 and Section 5.4, UML state chart diagrams consist of states and transitions. Individual test cases can be derived directly from transition parameters of the state charts for the test case generation. The specified state chart description for testing needs not the complete information for a automated control code generation of the system implementation. Because if the same state chart specification is used for the industrial control code generation and the test case generation, only the model transformation procedure is verified [26, 142]. Therefore the specified test specification can be small UML diagram parts which are used for the testing procedure. Figure 6.10 shows a selection of the implemented and used UML state chart diagrams for the test

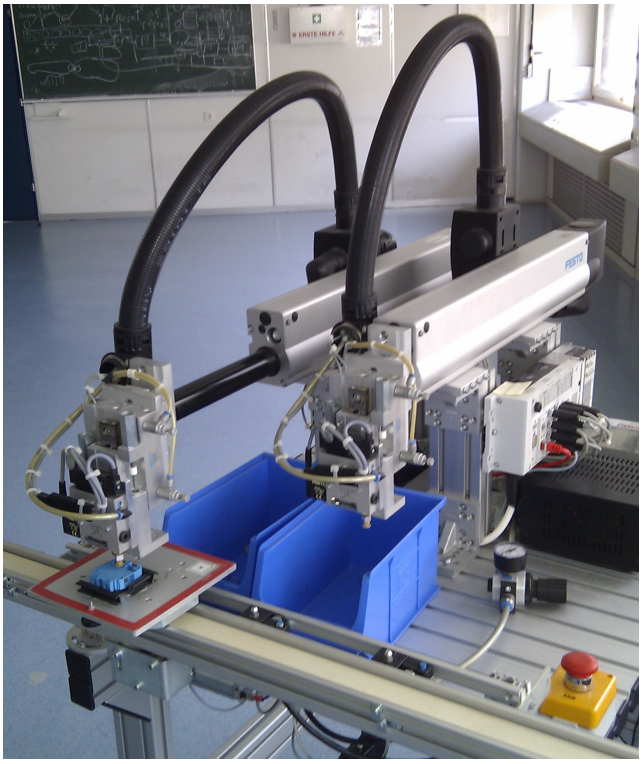
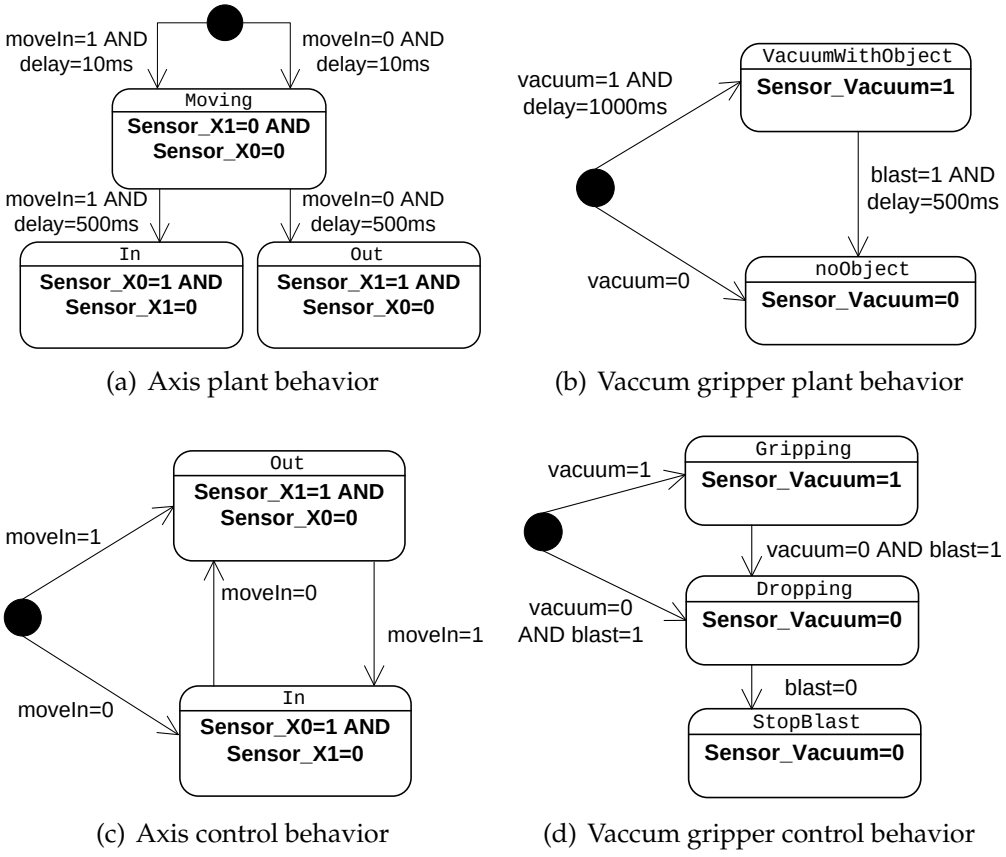


Figure 6.9: Picture of the sorting machine which consists of two pneumatic horizontal axis, two vertical axis, a vacuum gripper, and one conveyor for transporting pallets.

case generation process.

For identifying a test case or a test sequence, an explicit ID is generated, e.g. the test case is represented by TC1.1 and the test sequence is represented by TS1.1. A testing process which is only based on single test cases (TC1.1), see Figure 6.12, is not sufficient for testing industrial control software. Because system states are highly interwoven and the complexity of industrial automation systems tend to be immensely high. Such complexity cannot be tested only with test cases. Therefore a combination of test cases to test sequences, e.g. TS1.1, is required, see definition of test cases and test sequences in Chapter 3. In order to generate such test sequences automatically, the combination of single test cases has to fulfill the following rule: The post-condition parameters of a test case `n` has to be equal to the pre-condition parameters of the next test case `n+1` in the test sequence [26, 142].

In the example implementation, the test suite is included in one IEC 61499 BFB, see Figure 6.11. Therefore the interface of the BFB, the required algorithms, and the internal state machine, i.e. ECC, are generated automatically. An example implementation of a single test case and a test sequence is shown in Figure 6.12 and Figure 6.13.



(a) Axis plant behavior (b) Vaccum gripper plant behavior

(c) Axis control behavior (d) Vaccum gripper control behavior

Figure 6.10: State chart specification of the sorting machine. Axis component and the vacuum gripper component [26, 142].

The ECC is a finite state machine based on a Moore machine [162]. The implementation of the ECC is split into two parts, one for test cases and one for test scenarios. Figure 6.12 shows the ECC of a single test case implementation with the test cases ID=1.1 and ID=1.2. This test case represents the transition at the axis control behavior state chart in Figure 6.10(c). In the following the test execution for the test case ID=1.1 is shown:

1. The test case mode has to be selected, e.g. `Mode_TC=true`,
2. and the test case ID has to be defined, e.g. `TCselect=1.1`.
3. After the BFB initialization an event trigger `REQ` occurs and the generated algorithm of the BFB, e.g. Alg11_in_out, is executed.
4. The action parameter `moveIn=0` triggers an activation from the state IN to the state OUT.
5. An output event `CNF_TC` is sent.

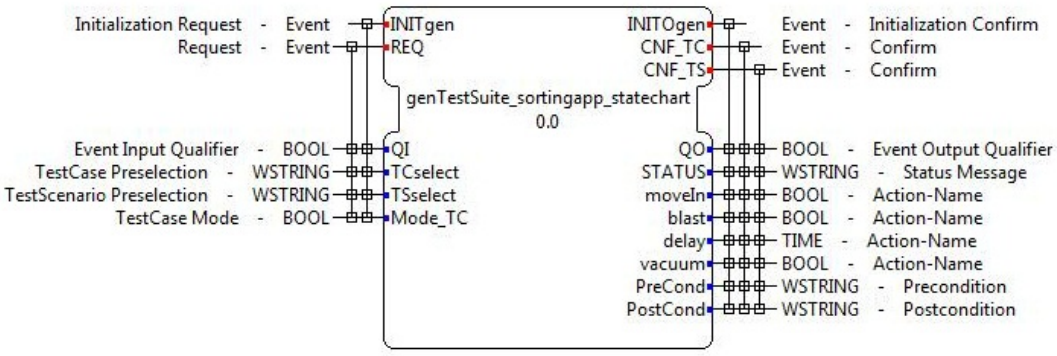


Figure 6.11: Resulting interface of the generated test suite BFB, i.e. <Model> IEC FB. This FB includes all test cases and test scenarios.

6. The results of the test case is positive or negative which is reported to the "REPORTING" FB.

The second part of the ECC implementation is required for the test sequences, see Figure 6.13. A test sequence is defined by a specific set of test cases which are executed in a predefined sequential order which is specified in the test specification [26,142]. In the case of the presented ECC, the test case mode has to be selected, e.g. Mode_TC=false, and the test sequence ID has to be parametrized, e.g. TSselect=1.1. After initializing the generated FB through an event trigger, the internal algorithms are executed in the following order:

1. Alg13_control_IN
2. Alg11_IN_OUT
3. Alg12_OUT_IN

When each algorithm is executed correctly, an output event `CNF_TC` is sent. If all test cases are completely executed, an output event `CNF_TS` is sent. After that the test sequence execution is finished and the next selected test case or selected test sequence is ready for execution.

There are some limitations of the implementation. The specification of the plant behavior state chart diagram includes timing information, for example: How much time does the axis need from one end position to the other end position? This information is not used because testing of timing behavior in a control application is not supported at this moment. Further, the extracted information and generated test cases from the UML state chart diagram are only positive test cases. No negative test cases and special test cases are generated automatically. Such kind of test cases have to be specified separately and will not be derived and extracted automatically from the specified positive test cases. This aspects should be considered in future research.

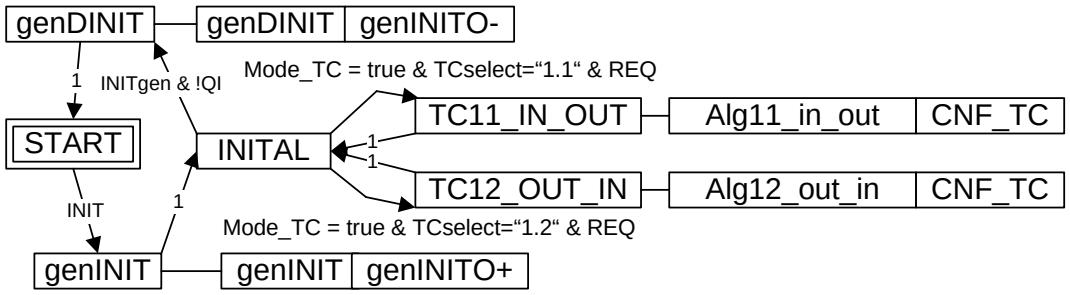


Figure 6.12: Results of the automatically generated ECC test case, which is included in one BFB and represents two single test case, i.e. test case ID=1.1 and test case ID=1.2 [26, 142].

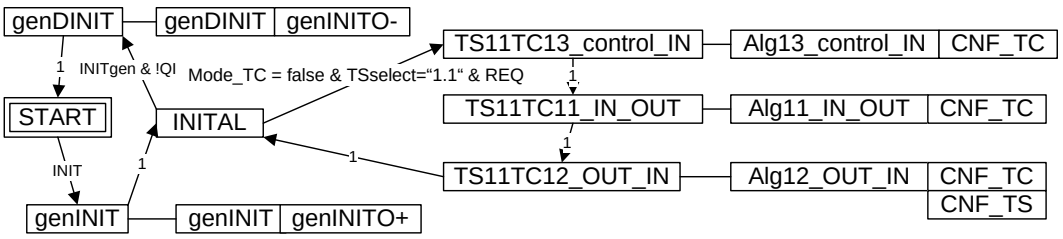


Figure 6.13: Results of the automatically generated ECC test scenario, which is included in one BFB and consists of a number of test case. The combination of several test cases ensure a test sequence, i.e. test sequence ID=1.1 [26, 142].

The results of the MBT implementation have shown that a detailed specification between the test suite interface, i.e. test suite FB and the SuT is of great importance for the model transformation process. The specification by using UML state chart diagrams is a suitable tool for the test specification of control applications. In order to split the specification into the control behavior specification and the plant behavior specification is an excellent methodology. Therefore the tests can be focused in more detail to test the SuT. This test approach is applicable for testing platform independent industrial automation systems because of the implemented model transformation rules of both standards, i.e. IEC 61131 and IEC 61499. The test cases and test sequences are included in one FB which allows to test in all application levels by instantiation of the test suite FB according to the SuT application, i.e. unit level for library tests, sub-system level, and system level. Note that the test suite FB interface is different for testing different application levels because the test suite FB interface is mirrored implemented to the SuT.

6.2 Resulting Comparison of the Testing Methods

All three testing techniques, the KDT, the unit testing with service sequences, and the MBT have their advantages and drawbacks.

All demonstrated testing techniques from Chapter 5 have not their strength for testing at all *testing levels* as defined in Section 3.2.

The KDT method is suitable for testing in all testing levels as presented in Section 6.1.1. The unit testing level is representative for testing a FB up to a FB library. The definition of a sub-system is a FB network with a number of FBs in a FB network, see Figure 3.8. Sub-system tests are also testable by using KDT as well as test whole system applications.

The second proposed testing technique is the unit testing method with sequence chart diagrams which is evaluated in Section 6.1.2. This testing technique is only suitable for testing on unit level which means that FB like BFB, CFB, SIFB are testable. IEC 61499 CFB can also be seen as sub-system because of the internal number of FBs. In this case a CFB is defined as a unit.

The MBT technique evaluated in Section 6.1.3 is also applicable for testing in all levels such as the KDT method. This method is not the best choice for unit testing because of the increased test specification and test generation effort, but a unit test is also possible. In the case of the presented example implementation, MBT is a powerful method for integration testing of sub-systems and the testing of whole system applications.

A significant difference occurs in the *test specification* method. The execution behavior has to be considered for testing IEC 61131 and IEC 61499 applications. IEC 61131 has a cyclic execution while an IEC 61499 applications has an event-based execution. This property has to be considered in the test specification process, for instance, add additional parameters for event trigger. Furthermore, on the one hand, the effort to develop the test specification is considered, such as formal test specification or textual and manual test specification and on the other hand, the test specification tool is considered. This means, for instance which UML diagram is the best choice for the test specification?

The test specification for the KDT method is done manually by using a formal textual description. By applying a small set of the defined keywords, non-programmers are able to read and write test cases without knowing details on the implementation behavior of the SuT. The test specification effort is high, but the simple test execution and test reporting of the Test Management system, i.e. FitNesse, makes this testing technique suitable for industrial requirements.

Test specification for unit tests by using UML sequence charts is a powerful tool. IEC 61499 allows the specification of multiple service sequences which are more restricted than UML sequence chart diagrams. Each IEC 61499 service sequence diagram describes a certain execution aspect of the FB. The main

advantage of this approach is that the test specification is directly included in the type definition of the FB. Therefore, the dedicated test specification is always part of the FB definition. No inconsistencies between different specification versions can occur and the test specification cannot be lost. The test specification effort is acceptable and the test execution and test reporting of the unit test is easy to handle because the Test Management system is directly integrated into the development tool, in this case 4DIAC-IDE.

The UML diagram family is a powerful specification tool in different fields of applications. The developed MBT technique for testing control software has the most flexible test specification possibility. At present, only the state chart diagram is used for the test specification, but also the whole UML diagram family can be used for the test specification. For this purpose, the transformation rules have to be extended for the test case generation. Currently the test execution has to be considered and done manually in the industrial control application because of the non-existing systematic Test Management system. Also the further processing of test reporting has to be carried out manually because the test results are stored in the test reporting FB.

An advantage of all these proposed testing techniques is that the techniques are suitable for testing on *different target platforms*. Real hardware PLCs or SoftPLCs which are running on common used PCs or industrial PCs are applicable. Hardware PLCs and SoftPLCs should possess the same execution behavior.

Not all proposed testing techniques are *suitable for testing* both IEC 61131 and IEC 61499 applications. The proposed testing technique for unit testing with sequence chart test specification is only applicable for IEC 61499 FBs. IEC 61131 does not support any additional specification methods included in the interface description such as sequence charts in IEC 61499, see [171–173]. Therefore the implementation of this testing technique is limited to IEC 61499 FB testing. Nevertheless, both testing techniques, KDT and MBT, are suitable for both implementation standards which are presented in Section 6.1.

An overview of the resulting comparison of the discussed testing techniques is presented in Table 6.1. The assessment of the different properties are defined by the values: Good, middle, and bad.

- *Good* means that the property is fully suitable and fully compatible with the testing technique.
- *Middle* means that the property is partly suitable and compatible with the testing technique. A testing process is possible but with high effort.
- *Bad* means that the property is not suitable and not compatible with the testing technique. Testing is not possible.

There are some limitations in the proposed concept of the test framework and the testing techniques. The presented testing techniques in Chapter 5 are

Properties	KDT	UTT	MBT
Unit test level	+	+	o
Sub-system test level	+	-	+
Integration testing	+	-	+
System test level	+	-	+
Testing specification effort	-	o	+
Test execution effort	+	+	o
Testing on different target platforms	+	+	+
Test Management system available	+	+	-
IEC 61131 suitable	+	-	+
IEC 61499 suitable	+	+	+

Table 6.1: Overview of the resulting comparison of the presented testing techniques; Keyword-driven Testing (KDT), Unit testing technique (UTT), Model-based Testing (MBT); + ... good , o ... middle , - ... bad.

suitable for testing industrial automation software. These testing methods can be enhanced by a combination of the presented testing techniques in order to reduce the specification effort. For instance, service sequence test specifications based on the IEC 61499 standard can become very large. A solution could be to model the test cases by UML sequence chart diagrams and generate the according service sequence diagrams as an input for the unit test framework in 4DIAC-IDE automatically. Furthermore, the test specification of the presented MBT technique currently uses state chart diagrams. The test specification of this testing method can be extended by supporting other UML diagrams such as sequence chart diagrams or other diagram types which are defined in Section 4.5. Therefore, the M2M transformation has to be extended by adding transformation rules.

In the current proposed approach the timing behavior of the SuT is not considered. The presented test case specification based on models, i.e. UML, supports timing specification but at the moment the test execution does not consider this timing information from the test case specification. In order to support the "timing" feature, the test framework has to be extended, which means that the Test Management system has to communicate directly in the depth with the run-time environment of the PLC, e.g. real-time measurements. This leads to running the Test Management system on the PLC. The execution behavior of PLCs on hardware and SoftPLCs related to testing real-time behavior requires future research.

The presented MBT method uses only Boolean data types, compared to the KDT method and the unit testing method with service sequence diagrams support continuous data values. Note that in safety application only Boolean input and output variables are allowed as input and output parameters. Con-

tinuous values can be represented in Boolean values by checking the boundary value [156]. Future applications of the MBT framework should support continuous test data values which can be achieved by extending the M2M transformation rules.

6.3 Summary

In this chapter the implementations of the proposed testing techniques for testing industrial automation software are presented. Each testing technique is implemented in a different example applications in order to show the detailed benefits of its testing property. The used applications are typical components from the field of discrete manufacturing and discrete production systems. After that, an evaluation and comparison of the implemented testing methods are explained. The comparison is based on the different testing levels, the testing effort of each testing technique, test execution review, and the testing on different target platforms, e.g. hardware PLC, SoftPLC. Additionally, a comparison for their suitability of the common development standards for industrial automation systems, i.e. IEC 61131 and IEC 61499, is given. Finally, the limitations of the proposed testing methods is presented.

The application design is a key aspect for increasing the overall software quality of industrial automation software. The next chapter will present resulting design rules and design guidelines for structuring industrial automation applications, which enables easier testing of control software. Further, reusability of automation components is achieved.

CHAPTER 7

Resulting Design Rules for Application Structure

In current industrial control applications, logical software code and testing code - if it exists - is often intertwined in the code, which hinders efficient and systematic testing [111]. Thus, the code is hard to read and modify during development and maintenance. A systematic testing process within the development phase is a significant benefit which reduces valuable development time. An encapsulated, component-based development approach with diagnosis and testing functionality is required for the design of industrial control software which supports an easy reuse for other industrial control applications. Such components need an easy method to be retested after changes. These requirements need special design rules for the application structure.

Today's state of the practice for the development of industrial automation applications is characterized by an approach where control applications are developed individually for each application. This results in a high development effort per plant as there is only little reuse of software components between different applications. In most cases, the testing of the application is typically performed rather late in the development cycle alongside the plant in an ad-hoc manner. The final application often contains many late changes and fixes which may break the overall application design. This leads to code clutter and couplings between application parts which results in reduced reusability of the application components [169]. The development time increases and finally it leads to high control application development costs. Therefore, means are necessary to reduce the development effort as well as the development uncertainty of industrial control applications [169].

7.1 Overall Architecture Definition

An important step towards an improved design structure is an architecture which defines properties in the industrial automation application structure. First, the architecture has to be decomposed into independent parts. A key feature for reducing the development effort is the reusability of application parts [174]. A further key requirement for reusing application parts in different applications is the decoupling of the application parts. There is the need that independent application parts should interact with each other in a defined way in order to clearly decouple the application parts. The application parts have to behave the same independent way in the context they are used in. This allows that once tested and validated components can be reused without additional testing effort [169].

According to the definition of a unit and a component in Section 3.2.1, the term Automation Component (AC) is used for reusable control parts. With the AC in hand, a definition how applications are composed is possible. Previous investigations have shown that a strong hierarchical approach is deemed to be a necessity for industrial automation [169, 174]. In order to look at the functional and mechanical view of production facilities for defining the hierarchy structure, a strong modularization and compositional approach is noticed.

For instance, plants are structured in cells which themselves contain several machines. The machines themselves are also built from mechanical assemblies and these are built from devices such as a cylinder or a pump. The main advantage is that the components on each level can be reused in many applications because they are independent from their usage. The second point is that sub-components are replaced easily as long as the new ones meet the mechanical and functional specifications. Furthermore, by mapping the mechanical structure of a plant to the structure of the control architecture also the programming of large automation systems are greatly simplified.

Figure 7.1 shows a hierarchical structure of an industrial automation application according to the mechanical and functional hierarchy of a plant. Each of the ACs represents a mechanical or logical unit which specifies different software-specific aspects for the implementation and execution of these units. This means that a lower level AC provides the component functionality, e.g. to rotate a pneumatic cylinder to the left or right. On a higher level, the AC provides the functionality of a sub-system, e.g. to moving a part from one conveyor belt to a second conveyor belt. Higher level ACs are composed from two or more of such ACs and aggregate the functionality of their contained ACs.

To utilize the full advantages of this concept three rules have to be considered, based on [169]:

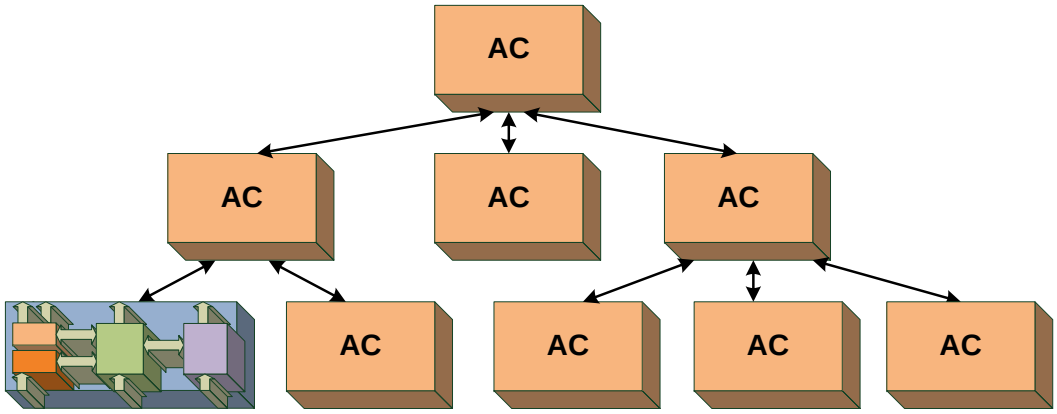


Figure 7.1: Hierarchical overview of an industrial automation application by using defined Automation Components (ACs) based on [169].

Strict hierarchy: An AC does not know who is using its interface and cannot make assumptions on the provider of an interface it is using. An AC is aware of the ACs' interface and the functionality of only one level below. This is all the information it can use to provide its own functionality. It is not allowed to use an ACs' interface that is located in an AC lower than one level below itself.

Single access point: ACs may interact only by means of the provided interface. This implies that another AC serving the same kind of interface can replace an AC and the user, i.e. upper level AC, of the AC can be left untouched. There may be established AC profiles for typical components in a similar way to existing device profiles for a field bus [175]. This especially will make a replacement of similar ACs easier.

Decoupling of ACs: ACs located on the same logical level in the plant may not directly interact and may not interact through their interface. If this functionality is needed, then it points to a design error because such a direct interaction is clearly a coordinating function between the two ACs. As stated in the first point, this is the task of an upper level AC.

7.2 Automation Component Model

An AC is defined as a collection of hardware, software, and properties, describing the interaction between different parts and specifying different software specific aspects for the implementation and execution of these objects.

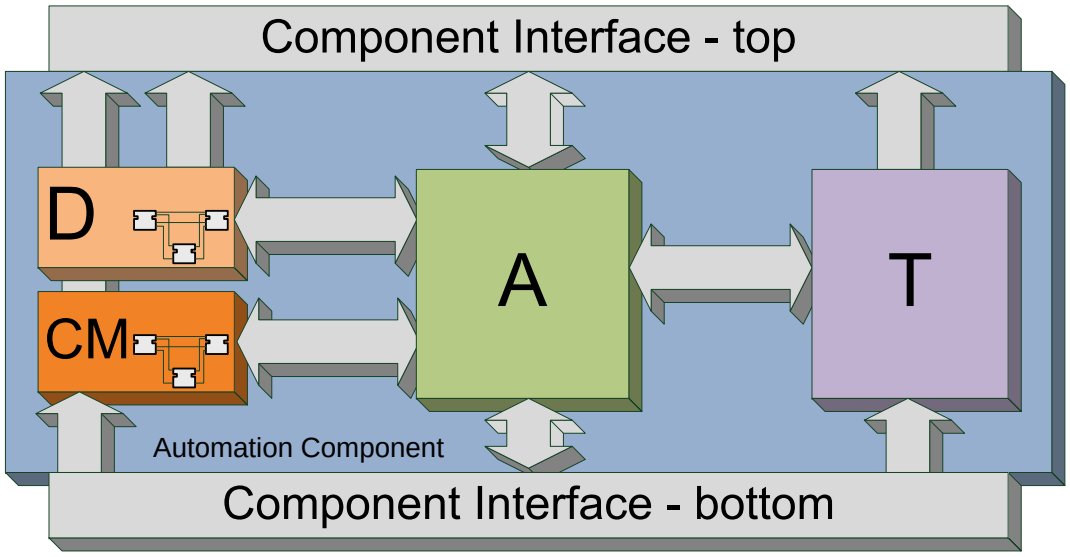


Figure 7.2: Structural overview of an AC component which includes functional aspects and interaction via defined interfaces within a hierarchical systems design. Diagnosis (D), Condition Monitoring (CM), Automation (A), Testing (T) - Sub-Components [169].

7.2.1 Sub-Component Models

A structural overview of the AC including four sub-components is presented in Figure 7.2. The sub-components are:

- A sub-component for automation aspects, e.g. logic, behavior, and implementation.
- A sub-component that handles test aspects, e.g. unit tests, integration tests, and factory acceptance tests.
- A sub-component for run-time fault analysis aspects, e.g. diagnosis, and finally.
- A sub-component for run-time fault prediction aspects, e.g. condition monitoring and data analysis.

Automation Sub-Component: The automation sub-component contains all information which is required to realize the desired behavior as specified. This sub-component must not depend on other sub-components and includes all requested functionality in itself. An existence or functionality of other sub-components is not necessary for the automation sub-component to fulfill its requirements. But other sub-components may influence the behavior

of the automation sub-component. For instance, the diagnosis component may change the execution flow and internal states in case of errors, or the test sub-component may change the state and data values due to an external demand, e.g. from the Test Management system. The automation sub-component, based on [169], contains:

- Automation components, e.g. sensors or actuators, to control the procedural units.
- Specification documents, e.g. requirements, design definitions.
- Electrical plan for wiring and interface to the control system.
- Function plan (software) for the implementation of the industrial automation software, e.g. industrial programming languages as defined in IEC 61131 or IEC 61499.
- The main (public) signal interface description.

All other sub-components can be removed or deactivated without changing the functionality of the automation sub-component.

Diagnosis Sub-Component: The diagnosis sub-component of the AC contains functionality which is responsible to check the correctness of the executed program and validates processed data for damage avoidance. Additional features in the diagnosis part handles already occurred errors in such a way, that no subsequent errors will occur. The diagnosis sub-component supports the automation sub-component in a way so that the AC is more robust against hardware faults.

Condition Monitoring Sub-Component: Condition monitoring is a management technique which uses the regular evaluation of the actual operating condition of plant equipment, production systems, and plant management function for optimizing the total plant operation. This technique is used to increase the availability, performance, consequential damage, and machine life as well as reduces consequential damage, spare parts inventories and breakdown maintenance of the machinery [176]. The information is monitored in the form of raw data. By using modern signal processing and analysis techniques, faults are detected early. The condition monitoring sub-component is responsible for monitoring the condition of the industrial control system, i.e. monitoring the hardware, and for predicting the future condition based on historical data. This sub-component has no influence on the primary functionality of the AC but it reduces down-times of the system by avoiding unscheduled maintenance work.

Additional software functionality and hardware equipment may be needed in order to add condition monitoring functionality [169]:

- Specification documents define required analysis methods, for instance Failure Mode and Effects Analysis (FMEA) or Hazard and Operability (HAZOP).
- Hardware, e.g. sensors, for measuring the condition of system elements.
- Wiring the new sensors in electrical plans and mapping the inputs to new signal data.
- Add functionality to software parts for processing, recording, and visualizing by using the new signal data.

Test Sub-Component: The functionality of the test sub-component is to support the testing process for testing the AC, in order to find systematic errors during the development process.

The test sub-component, based on [169], contains:

- Code inspections, reviews, automatic code analysis such as LINT[22] for automation, and code verification (against specification).
- Test scripts for automatic execution and operation of tests.
- Additional software and interfaces to communicate with the Test Management system (test drivers, mock-ups, specification and implementation of simulation components).
- Requirements concerning the enhancement of functionality of the Test Management system, e.g. interfaces, commands.

7.2.2 Interfaces

The AC has a three dimensional interface which is able to communicate with other ACs. The defined interfaces are presented in Figure 7.3:

- Inter-Component Interface
- Intra-Component / Inter-Sub-Component Interface
- Inter-Tool Interface

[22] LINT is a white-box testing method (see Section 1.3.2) and can be used to check C code for errors that may cause a compilation failure or unexpected results at run-time. http://docs.oracle.com/cd/E19957-01/806-3567/lint.html, visited: March 2013

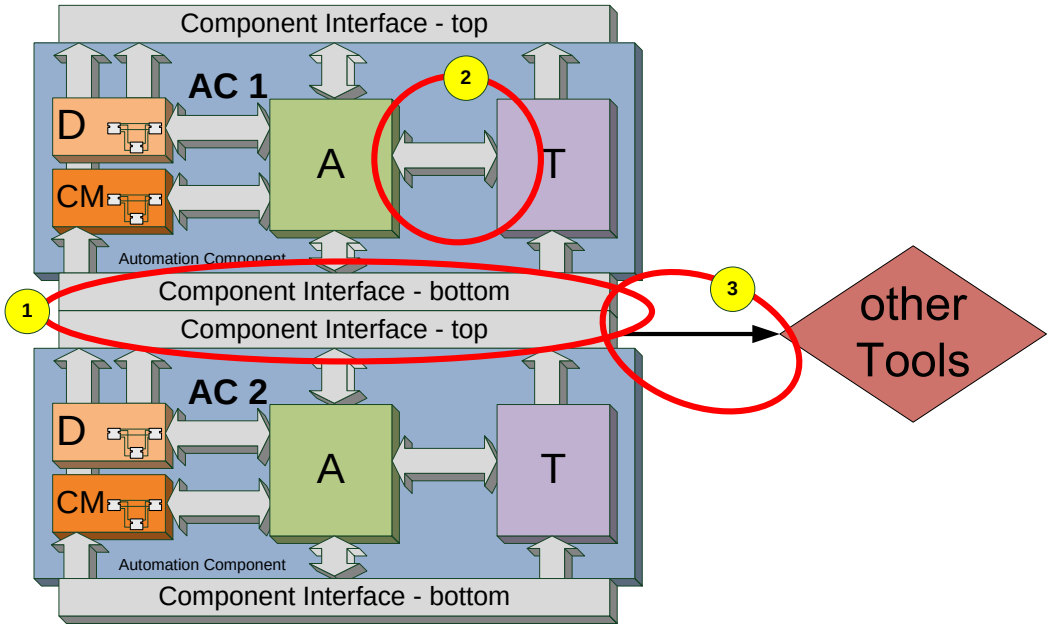


Figure 7.3: Interface definition in AC networks. (1) Inter-Component Interface, (2) Intra-Component / Inter-Sub-Component Interface, (3) Inter-Tool Interface.

Inter-Component Interface: This interface is located between ACs for an aggregation of components, see Figure 7.3 (1). The bottom interface of a higher level AC is connected with the top interface of two or more lower level ACs. The Inter-Component interface of ACs is defined by the top/bottom interface of the ACs. The main (must have) interface in the AC is the automation sub-component. The interfaces from the test sub-component and the diagnosis sub-component are additionally used to control test functionality and read internal states via the diagnosis interface. The implementation of the interfaces may result in hidden interfaces, e.g. direct access from the external Test Management system into a specific component [169].

Intra-Component / Inter-Sub-Component Interface: The location of this interface is between sub-components of ACs, see Figure 7.3 (2). The interface defines which sub-component is allowed to read and write (modify) data or states in other sub-components.

Inter-Tool Interface: An interface between different tools is necessary and used to describe the different aspects of an AC, see Figure 7.3 (3). This interface is located orthogonal to the previous two interfaces. Note that this interface allows only reading data from an AC. Write values from external tools to the AC will break the overall application design.

7.3 Summary

In this chapter a new component architecture is specified which supports component oriented design, reusability, and encapsulation of functional parts for developing industrial automation applications. A validation of these design rules is implemented in a example application presented in [169]. The AC model and the hierarchical structure is a valuable approach for encapsulated and reusable components. The encapsulation of the automation, diagnosis, condition monitoring, and test sub-components allow other components to monitor the condition of the system. Further, supported test functionality is available for testing after changes have been made. The ACs interact by exchanging data via the defined interfaces. These interfaces allow the communication to higher-level ACs,to lower-level ACs as well as to external tools. Because of the defined interfaces the component variants are seamlessly exchangeable.

CHAPTER 8

Conclusion and Outlook

8.1 Conclusion

Nowadays, the functional requirements of industrial automation systems increase steadily. Industrial automation software becomes more important in current and future industrial applications. Such software allows to cope with the ever increasing complexity of modern applications. The development of software components for industrial automation systems becomes more time consuming and deals with adaptations of the industrial control code on short notice. In order to address these challenges and reduce the development time and costs for upcoming projects, the engineering efficiency needs to be increased in terms of reusability support of tested and testable software components. This thesis aims at the development of a test infrastructure with appropriate testing techniques for industrial automation software in order to make a significant step towards increasing the quality of industrial automation software.

In this thesis a new test framework is introduced which is able to support testing of industrial automation software based on IEC 61131 and IEC 61499. With the use of the new test framework and the test strategies such as Test-First Development (TFD), the software quality can be increased to reduce the faults in the developed software. An identification of three testing levels of industrial automation systems is presented. Based on a bottom-up design, one identifies a detailed view on individual components and units on level one, a more abstract view on the sub-system with focus on the interaction of components on level two, and the system level as well as business level on level three.

With the knowledge of the identified test levels and by considering the newly defined criteria, see Section 4.3, a selection of Unified Modeling Language (UML) models useable for specifying tests in the industrial automation domain is proposed. A model-based test specification approach enables flexible changes of requirements. This model-based test specification, especially the state chart diagram, is used in the test case extraction implementation. Furthermore, this knowledge is used for the new developed testing technique definitions.

Research Question 1: Which test method is suitable for the different application layers? The classification of the testing technique to their testing levels is presented in Section 3.2, especially in Figure 3.5. Furthermore, four testing techniques are presented which are able to test industrial automation software considering the TFD strategy.

1. First, the manual testing method is presented in which all processes have to be done manually, i.e test case specification, test case generation, test case execution, and test result analysis. The manual testing method is suitable for testing at all testing levels, i.e. unit test level, sub-system test level, system test level.

2. The second presented testing technique is the Keyword-driven Testing (KDT) method. New defined keywords are used for the test case specification which are automatically executed by the Test Management system to the System under Test (SuT). This testing method is suitable for testing at all testing levels. However, the benefits of this testing method are in library tests and integration testing because of the direct access from the Test Management system to the Function Block (FB) interfaces.

3. The third presented testing technique is the unit testing method with service sequence diagrams. Service sequence diagrams, defined in the IEC 61499 standard, are used for the test specification. A service sequence diagram is more restricted than the UML sequence chart diagram. Both diagram types are suitable tools for the test specification of industrial software components. The test infrastructure is directly included in the IEC 61499 development environment 4DIAC-IDE and is available as an open-source plug-in. This testing method is only suitable in the unit testing level, i.e. testing of FBs and FB libraries, because of the restricted Test Management system which allows only a test of one FB interface.

4. The fourth and last presented testing technique is the Model-based Testing (MBT) method for industrial applications. This presented testing

method is the most flexible technique because of the extended test specification possibility. Therefore, only selected UML model diagrams based on the proposed criteria are suitable for the test specification. By using the Model-to-Model (M2M) transformation, the test information is automatically extracted from the test specification. Finally, an executable IEC 61131 or IEC 61499 test application is generated automatically. The MBT method is suitable for testing at all testing levels. This method has the most benefits in integration tests, i.e. sub-system test level, and system tests, i.e. system test level, because of the dynamic generation of the test application.

Research Question 2: Is the test-first development approach from the software engineering domain a suitable method for developing and testing industrial automation code? The outcome of the comparison between the presented testing methods from Chapter 5 shows that all testing techniques are suitable for the TFD strategies which is evaluated in Section 6.1.

Research Question 3: How should control applications be designed to enable the testing of their functionality in an easier way? In current industrial automation applications, logical software code and testing code are often intertwined in the code which hinders efficient and systematic testing. These requirements need special design rules for the application structure. A new automation component architecture and a new component called Test-Diagnosis-Automation (TDA) component for developing industrial automation applications is presented. The component consists of four internal sub-components, i.e. automation, diagnosis, condition monitoring, and test sub-component to monitor the condition of the automation system. The software quality can be increased based on the component-based structure, the new defined architectural design, and the presented design rules.

Research Question 4: Can testing improve the reusability of control software? Reusability of developed software components is an important requirement for efficient software development. Because of the need for high software quality components that are easy to reuse for other industrial applications, the components have to be encapsulated and fully tested. These components need an easy method to be retested after changes. A strict interface description of such components is defined which enables efficient testing. Also the components can easyly be exchanged in existing control applications. The combination of the TFD approach and the new development architecture definition for control applications enables the reusability of control software components. Therefore tested components, such as the TDA component, improve the reusability of control software.

This work is an important step towards improving the software quality for industrial automation systems. It is a new approach for testing industrial automation systems on different levels of applications. With this approach, control engineers are able to specify test cases and execute them during the development phases. Furthermore, not only control engineers can use this test framework support, but also test engineers are addressed, who test systems on higher application levels. This test framework is one possible solution for requirement changes on short notice in the development phase to keep the software quality on a high level.

8.2 Outlook

The presented work is a first step towards improving the quality of industrial automation software modeled in IEC 61131 and IEC 61499. The presented implementation examples show that the approaches and concepts of this work are feasible in practice. Nonetheless, some points remain open for discussion and need a further investigation. In order to bring the here presented testing techniques into industrial use.

A first step in future research are the presented limitations of the testing methods which are introduced in Section 6.2. These limitations should be considered in order to optimize the specification effort by using different UML diagrams. Furthermore, support testing of timing behavior of the SuT should be considered.

Further research in the field of other and extended test specification methods such as the UML Testing Profile [21] should be considered. There is much potential to simplify the test specification procedure.

The new automation component approach for developing industrial automation applications in a hierarchical manner is presented to support testability and reusability of the industrial software components. This TDA component design should be used in larger and more complex industry applications in order to investigate the benefits in more detail. Furthermore, a refinement of the the TDA component approach should be conducted.

This thesis is focused on black-box testing. Future research is needed in White-box testing and Gray-box testing for the control software testing domain. Therefore available testing methods from the software engineering domain should be considered to adapt them to the industrial automation domain.

A further challenge is to provide information to support quantitative managerial decision-making during the software life-cycle [177]. Software metrics can support this challenge. There exist a lot of metrics which concentrate on the structure of a program or code than on the contents itself [178], i.e. white-box testing. The size metric is one of the mostly used metrics, i.e. Lines of Code (LOC). Further metrics are the “Halstead Measure” and the “McCabe’s

Cyclomatic Complexity". The Halstead Complexity Measure focuses on the software measurement by measure in terms of operators and operands [179]. This measure is computed statically from the code. Another complexity measure is the Cyclomatic Complexity from Thomas J. McCabe [180, 181] which is related to the flow graph of a program. This metric was developed to indicate the testability and understandability of a program. The effort for testing software components can be determined by such metric calculations. Future work should consider such calculation and measurement methods to determine the quality of software.

Bibliography

[1] H. M. Sneed, *Software-Projektkalkulation: Praxiserprobte Methoden der Aufwandsschätzung für verschiedene Projektarten*. Carl Hanser Verlag GmbH & CO. KG, 2005.

[2] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002.

[3] B. Favre-Bulle and G. Zeichen, "Die Zukunft der Produktionswissenschaft," 2005, a Study.

[4] T. Cleff, *Basiswissen Testen von Software*. W3L GmbH, 2010.

[5] B. Kormann, D. Witsch, and B. Vogel-Heuser, "Automatische Testfallgenerierung mittels Model-Checking für Steuerungsprogramme," in *Tagungsband GMA-Kongress Automation*, 2010.

[6] K. Stroggylos and D. Spinellis, "Refactoring–Does It Improve Software Quality?" in *Proceedings of the 5th International Workshop on Software Quality*, ser. WoSQ '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/WOSQ.2007.11

[7] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 768 –781, Nov. 2011.

[8] I. Hegny, T. Strasser, M. Melik-Merkumians, M. Wenger, and A. Zoitl, "Towards an Increased Reusability of Distributed Control Applications Modeled in IEC 61499," in *Proceedings of 2012 IEEE 17th International*

Conference on Emerging Technologies and Factory Automation (ETFA 2012), 2012, Krakow, Poland; 2012-09-17 – 2012-09-21. [Online]. Available: http://publik.tuwien.ac.at/files/PubDat_209963.pdf

[9] IEEE Standards Board, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, p. 85, Sep. 1990.

[10] H. Buwalda, D. Janssen, and I. Pinkster, *Integrated Test Design and Automation: Using the TestFrame Method*. Addison-Wesley, 2002. [Online]. Available: http://books.google.at/books?id=6IVQAAAAMAAJ

[11] S. Bärisch, "Domain-Specific Model-Driven Testing," Ph.D. dissertation, University Kiel, 2010.

[12] B. Hailpern and P. Santhanam, "Software Debugging, Testing, and Verification," *IBM Syst. J.*, vol. 41, no. 1, pp. 4–12, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1147/sj.411.0004

[13] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, 1st ed. Wiley Computer Publishing, 2002, iSBN-10: 0471081124, ISBN-13: 978-0471081128.

[14] E. W. Dijkstra, "The Humble Programmer," *Commun. ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1972. [Online]. Available: http://doi.acm.org/10.1145/355604.361591

[15] M. Conrad, "Modell-basierter Test eingebetteter Software im Automobil," Ph.D. dissertation, University of Technology Berlin, 2004.

[16] R. Stetter and M. Erben, "Automatisches Testen bei SPS-Steuerungssoftware," *atp edition*, pp. 31–34, 2008.

[17] British Computer Society Specialist Interest Group in Software Testing (BCS SIGiST), "Standard for Software Component Testing," British Computer Society, Apr. 2001. [Online]. Available: http://www.testingstandards.co.uk/

[18] G. E. Mogyorodi, "Requirements-Based Testing - Cause-Effect Graphing," Software Testing Services, 2005-2010. [Online]. Available: http://softestserv.ca/RBT_Cause-Effect_Graphing2.pdf

[19] G. J. Myers, *Methodisches Testen von Programmen*. Oldenbourg Verlag, 1991.

[20] B. Beizer, *Software Testing Techniques*, 2nd ed. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[21] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. Berlin, Heidelberg: Springer-Verlag, 2007.

[22] J. Bach and P. J. Schroeder, "Pairwise testing: A best practice that isn't," in *22nd Annual Pacific Northwest Software Quality Conference*, 2004, pp. 180–196.

[23] M. Grochtmann and K. Grimm, "Classification Trees for Partition Testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993. [Online]. Available: http://dx.doi.org/10.1002/stvr.4370030203

[24] M. Grochtmann, "Test Case Design Using Classification Trees," in *STAR'94*, 1994, 8 - 12 May 1994, Washington, D.C.

[25] A. Petrenko and N. Yevtushenko, "Testing from Partial Deterministic FSM Specifications," *IEEE Trans. Comput.*, vol. 54, no. 9, pp. 1154–1165, Sep. 2005. [Online]. Available: http://dx.doi.org/10.1109/TC.2005.152

[26] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Automated Test Case Generation for Industrial Control Applications," in *Recent Advances in Robotics and Automation*, ser. Studies in Computational Intelligence, G. Sen Gupta, D. Bailey, S. Demidenko, and D. Carnegie, Eds. Springer Berlin Heidelberg, 2013, vol. 480, pp. 263–273. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37387-9_20

[27] C. Meinel and C. Stangier, "Modular Partitioning and Dynamic Conjunction Scheduling in Image Computation," in *Proc. of the 2002 IEEE/ACM Int. Workshop on Logic and Synthesis (IWLS02)*, 2002, pp. 391–396.

[28] Federal Aviation Administration (FAA), "Software Verification Tools Assessment Study," Air Traffic Organization Operations Planning Office of Aviation Research and Development, Tech. Rep. DOT/FAA/AR-06/54, 2007. [Online]. Available: actlibrary.tc.faa.gov

[29] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, ser. The Addison-Wesley Object Technology Series. Addison-Wesley, 2000. [Online]. Available: http://books.google.at/books?id=P3UkDhLHP4YC

[30] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, 1978.

[31] U. Vigenschow, *Testen von Software und Embedded Systems - Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*, 2nd ed. dpunkt.verlag, Heidelberg, 2010.

[32] H. M. Hanisch, A. Lobov, J. L. Martinez Lastra, R. Tuokko, and V. Vyatkin, "Formal Validation of Intelligent Automated Production Systems towards Industrial Applications," *Intl. J. of Manufacturing Technology and Management*, vol. 8, no. 1, pp. 75 – 106, 2006.

[33] E. M. Clarke, E. M. J. Clarke, and O. Grumberg, *Model Checking*. MIT Press, 2000.

[34] V. Vyatkin and G. Bouzon, "Using Visual Specifications in Verification of Industrial Automation Controllers," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 5:1–5:9, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1155/2008/251957

[35] T. Hussain and R. Eschbach, "Statistical Testing of IEC 61499 Compliant Software Components," in *Preprints of the 13th IFAC Symposium on Information Control Problems in Manufacturing, Moscow, Russia*, 2009.

[36] R. Pelánek, "Fighting State Space Explosion: Review and Evaluation," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, D. Cofer and A. Fantechi, Eds. Springer Berlin Heidelberg, 2009, vol. 5596, pp. 37–52. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03240-0_7

[37] K. Loeis, M. Younis, and G. Frey, "Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems," in *Emerging Technologies and Factory Automation (ETFA 2005), 10th IEEE Conference on*, vol. 1, Sep. 2005, pp. 4 pp.–250.

[38] D. Pollmächer, W. Zimmermann, and H.-M. Hanisch, "Translation Validation for Model-based Code-generators for PLCs," in *Emerging Technologies and Factory Automation (ETFA 2005), 10th IEEE Conference on*, vol. 1, Sep. 2005, pp. 113 –120.

[39] D. Hurnaus and H. Prähofer, "Programming Assistance Based on Contracts and Modular Verification in the Automation Domain," in *Symposium On Applied Computing - SAC*. ACM, 2010, pp. 2544–2551.

[40] S. Preusse and H.-M. Hanisch, "Verifying Functional and Non-functional Properties of Manufacturing Control Systems," in *3rd Intl. Workshop on Dependable Control of Discrete Systems*, Jun. 2011, pp. 41 –46.

[41] V. Dubinin, V. Vyatkin, and H.-M. Hanisch, "Modelling and Verification of IEC 61499 Applications using Prolog," in *IEEE Conf. on Emerging Technologies and Factory Automation*, Sep. 2006, pp. 774 –781.

[42] G. Čengić and K. Åkesson, "On Formal Analysis of IEC 61499 Applications, Part A: Modeling," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 136 –144, May 2010.

[43] C. Gerber and H.-M. Hanisch, "Does portability of IEC 61499 mean that once programmed control software runs everywhere?" in *10th IFAC Workshop on Intelligent Manufacturing Systems*, 2010, pp. 29 – 34.

[44] C. Sünder and V. Vyatkin, "Functional and Temporal Formal Modelling of Embedded Controllers for Intelligent Mechatronic Systems," *Intl. J. of Mechatronics and Manufacturing Systems*, vol. 2, no. 1/2, pp. 215 – 235, 2009.

[45] S. Preusse and H.-M. Hanisch, "Specification and Verification of Technical Plant Behavior With Symbolic Timing Diagrams," in *3rd Intl. Design and Test Workshop*, Dec. 2008, pp. 313 –318.

[46] T. Hussain and G. Frey, "UML-based Development Process for IEC 61499 with Automatic Test-case Generation," in *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, Sep. 2006, pp. 1277 –1284.

[47] B. Bouyssounouse and J. Sifakis, Eds., *Embedded Systems Design-The ARTIST Roadmap for Research and Development*, ser. Lecture Notes in Computer Science. Springer Verlag Berlin Heidelberg, Feb. 2005, vol. 3436/2005.

[48] R. Hametner, D. Winkler, and A. Zoitl, "Agile Testing Concepts Based on Keyword-driven Testing for Industrial Automation Systems," in *38th Conference of the IEEE Industrial Electronics Society (IECON)*, Montreal, Canada, 2012.

[49] B. Kitchenham and S. Pfleeger, "Software Quality: The Elusive Target [special issues section]," *Software, IEEE*, vol. 13, no. 1, pp. 12–21, jan 1996.

[50] ISO/IEC 9126, *Software engineering - Product quality*. International Electrotechnical Commission (IEC), 2001.

[51] B. W. Böhm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," in *Proceedings of the 2nd international conference on Software engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 592–605. [Online]. Available: http://dl.acm.org/citation.cfm?id=800253.807736

[52] IEC 61131-3, *IEC 61131-3 Standard - Programmable controllers - Part 3: Programming languages*. International Electrical Commission, 2003.

[53] R. W. Lewis, *Programming Industrial Control Systems Using IEC 1131-3 (I E E Control Engineering Series)*. Stevenage, UK: Institution of Engineering and Technology, 1998.

[54] C. Sünder, M. Wenger, C. Hanni, I. Gosetti, H. Steininger, and J. Fritsche, "Transformation of Existing IEC 61131-3 Automation Projects Into Control Logic According to IEC 61499," in *Emerging Technologies and Factory Automation (ETFA 2008), IEEE International Conference on*, Sep. 2008, pp. 369 –376.

[55] IEC 61499-1, *Function blocks – Part 1: Architecture*. Geneva: International Electrical Commission, 2005.

[56] R. Lewis, *Modeling Control Systems Using IEC 61499 - Applying Function Blocks to Distributed Systems*. London: Institution of Engineering and Technology, 2001.

[57] F. Xia, Z. Wang, and Y. Sun, "Towards Component-based Control System Engineering With IEC 61499," in *Intelligent Control and Automation, 2004. WCICA 2004. Fifth World Congress on*, vol. 3, Jun. 2004, pp. 2711 – 2715 Vol.3.

[58] A. Zoitl, "Basic Real-Time Reconfiguration Services for Zero Down-Time Automation Systems," Ph.D. dissertation, Vienna University of Technology, Automation and Control Institute, Vienna, Nov. 2007.

[59] M. Seitz, V. Ehret, M. Kiefer, A. Ziegler, E. Kruschitz, and E. Usselmann, "Automatisches Testen von Automatisierungssystemen," 2009. [Online]. Available: http://www.automatisierungs-region.de

[60] IEC 61508, *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. International Electrical Commission, 2005.

[61] IEC 61511, *Functional Safety - Safety Instrumented Systems for the Process Industry Sector*. International Electrical Commission, 2004.

[62] F. Auinger, M. Vorderwinkler, and G. Buchtela, "Interface Driven Domain-independent Modeling Architecture for "soft-commissioning" and "reality in the loop"," in *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future - Volume 1*, ser. WSC '99. New York, NY, USA: ACM, 1999, pp. 798–805. [Online]. Available: http://doi.acm.org/10.1145/324138.324504

[63] H. Schludermann, T. Kirchmair, and M. Vorderwinkler, "Soft-commissioning: Hardware-in-the-loop-based Verification of Controller Software," in *WSC '00: Proceedings of the 32nd conference on Winter simulation*. San Diego, CA, USA: Society for Computer Simulation International, 2000, pp. 893–899.

[64] K. Kabitzsch, A. Gellrich, and J. Naake, "Automatisierte Steuerungstests vereinfachen die virtuelle Inbetriebnahme in der Fabrikautomation," *atp edition*, vol. 4, pp. 14–16, Apr. 2012.

[65] D. Korotkiy and K. Bender, "Universelle Architektur zur Testautomatisierung," *atp edition*, vol. 5, pp. 26–31, May 2010.

[66] T. Hussain and R. Eschbach, "Automated Fault Tree Generation and Risk-based Testing of Networked Automation Systems," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, Sep. 2010, pp. 1 –8.

[67] M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Simulation-based Testing of Distributed Systems," University of Colorado, Department of Computer Science, Tech. Rep., 2006.

[68] J. Greifeneder, P. Weber, M. Barth, and A. Fay, "Simulationsbasierte Steuerfunktionstests," *atp edition*, vol. 4, pp. 34–41, Apr. 2012.

[69] H. Prähofer, R. Schatz, C. Wirth, and H. Mossenbock, "A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 4, pp. 641–651, Nov. 2011.

[70] ——, "Deterministic Replay Debugging of IEC 61131-3 SoftPLC programs," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, July 2010, pp. 1110–1117.

[71] C. Wirth, H. Prähofer, and R. Schatz, "A Multi-level Approach for Visualization and Exploration of Reactive Program Behavior," in *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, Sep. 2011, pp. 1–4.

[72] G. Pietrek and J. Trompeter, *Modellgetriebene Softwareentwicklung - MDA und MDSD in der Praxis*. ebtwickler.press, 2007.

[73] S. D. Panjaitan and G. Frey, "Development Process for Distributed Automation Systems Combining UML and IEC 61499," *Int. J. Manufacturing Research*, vol. 2, no. 1, pp. 1–20, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijmr/ijmr2.html#PanjaitanF07

[74] T. Kühne, "What is a model?" in *Language Engineering for Model-Driven Software Development, Number 04101 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl*, 2005.

[75] T. Stahl, M. Völtner, S. Efftinge, and A. Haase, *Modellgetriebene Softwareentwicklung*, 2nd ed. dpunkt.verlag, 2007.

[76] H. Stachowiak, *Allgemeine Modelltheorie*. Springer-Verlag, 1973. [Online]. Available: http://books.google.at/books?id=DK-EAAAAIAAJ

[77] M. Utting and B. Legeard, *Practical Model-based Testing*. Morgan Kaufmann, 2007.

[78] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based Testing in Practice," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 285–294. [Online]. Available: http://doi.acm.org/10.1145/302405.302640

[79] A. Korff, *Modellierung von eingebetteten Systemen mit UML und SysML*. Springer, 2008, iSBN 978-3-8274-1690-2.

[80] T. Weilkiens, *Systems Engineering mit SysML/UML - Modellierung, Analyse, Design*. dpunkt.verlag, Heidelberg, 2008, iSBN 978-3-89864-577-5.

[81] C. Pang and V. Vyatkin, "Automatic Model Generation of IEC 61499 Function Block Using Net Condition/Event Systems," in *6th IEEE International Conference on Industrial Informatics (INDIN)*, 2008, pp. 1133–1138.

[82] M. Wenger, A. Zoitl, C. Sünder, and H. Steininger, "Semantic Correct Transformation of IEC 61131-3 Models Into the IEC 61499 Standard," in *IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2009, pp. 1 –7.

[83] ——, "Transformation of IEC 61131-3 to IEC 61499 Based on a Model Driven Development Approach," in *7th IEEE International Conference on Industrial Informatics (INDIN)*, Jun. 2009, pp. 715 –720.

[84] M. Wenger, R. Hametner, and A. Zoitl, "IEC 61131-3 Control Applications vs. Control Applications Transformed in IEC 61499," in *IFAC Workshop on Intelligent Manufacturing Systems (IMS 10)*, vol. 10, 2010, pp. 30–35.

[85] G. Frey and K. Thramboulidis, "Einbindung der IEC 61131 in modellgetriebene Entwicklungsprozesse," in *Kongress Automation, Baden-Baden, Germany*. VDI-Berichte 2143, 2011, pp. 21–24.

[86] K. Thramboulidis and G. Frey, "An MDD Process for IEC 61131-based Industrial Automation Systems," in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, Sep. 2011, pp. 1 –8.

[87] S. Panjaitan and G. Frey, "Combination of UML Modeling and the IEC 61499 Function Block Concept for the Development of Distributed Automation Systems," in *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, Sep. 2006, pp. 766 –773.

[88] V. Dubinin, V. Vyatkin, and T. Pfeiffer, "Engineering of Validatable Automation Systems Based on an Extension of UML Combined With Function Blocks of IEC 61499," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, April 2005, pp. 3996 –4001.

[89] T. Hussain and G. Frey, "Defining IEC 61499 Compliance Profiles using UML and OCL," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2, Jun. 2007, pp. 1157 –1162.

[90] B. Vogel-Heuser, S. Braun, B. Kormann, and D. Friedrich, "Implementation and Evaluation of UML as Modeling Notation in Object Oriented Software Engineering for Machine and Plant Automation," in *18th IFAC World Congress*, 2011, pp. 9151–9157, Aug. 28 - Sep. 2, 2011.

[91] K. Thramboulidis, "Model-integrated Mechatronics - Toward a new Paradigm in the Development of Manufacturing Systems," *Industrial Informatics, IEEE Transactions on*, vol. 1, no. 1, pp. 54 –61, Feb. 2005.

[92] G. J. Myers, *The Art of Software Testing*, 2nd ed. Wiley, Jun. 2004.

[93] L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, 1st ed. Addison-Wesley Professional, 2009, iSBN-10: 0321534468, ISBN-13: 978-0321534460.

[94] H. M. Sneed, M. Baumgartner, and R. Seidl, *Der Systemtest: Von den Anforderungen zum Qualitätsnachweis*, 2nd ed. München: Hanser, 2009.

[95] M. Gogolla and M. Richters, "On Combining Semi-formal and Formal Object Specification Techniques," in *Recent Trends in Algebraic Development Techniques*, ser. Lecture Notes in Computer Science, F. Presicce, Ed. Springer Berlin Heidelberg, 1998, vol. 1376, pp. 238–252. [Online]. Available: http://dx.doi.org/10.1007/3-540-64299-4_37

[96] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[97] S. Mouchawrab, L. Briand, Y. Labiche, and M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 161–187, Mar. 2011.

[98] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test Case Generation Based on State and Activity Models," *Journal of Object Technology*, vol. 9, no. 5, pp. 1–27, Sep. 2010. [Online]. Available: http://www.jot.fm/contents/issue_2010_09/article1.html

[99] E. Essler, "Testfälle erfolgreich modellieren," *ObjectSpectrum*, vol. Testing, p. 4, 2008.

[100] D. Winkler, S. Biffl, and T. Östreicher, "Test-Driven Automation: Adopting Test-First Development to Improve Automation Systems Engineering Processes," in *European System and Software Process Improvement and Innovation*, 2009.

[101] A. Spillner, "The W-MODEL - Strengthening the Bond Between Development and Test," University of Applied Sciences Bremen, Tech. Rep., 2002.

[102] ISO/IEC 29119, *Software Testing*. International Electrotechnical Commission (IEC), May 2007.

[103] W. Droeschel and M. Wiemers, *Das V-Modell 97: Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*. Oldenbourg Wissenschaftsverlag, 1999, iSBN-10: 3486250868, ISBN-13: 978-3486250862.

[104] A. Spillner, T. Linz, and H. Schaefer, *Software Testing Foundations*. Heidelberg: dpunkt., 2005.

[105] Software and Systems Engineering Standards Committee, "IEEE Standard for Software and System Test Documentation," *IEEE Std 829-2008*, pp. 1 –118, 2008.

[106] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What Do We Know About Test-Driven Development?" *Software, IEEE*, vol. 27, no. 6, pp. 16 –19, Nov. 2010.

[107] L.-O. Damm and L. Lundberg, "Results From Introducing Component-level Test Automation and Test-Driven Development," *J. Syst. Softw.*, vol. 79, pp. 1001–1014, Jul. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1159517.1159528

[108] L. Madeyski, *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2010.

[109] K. Beck, *Test Driven Development by Example*. Addison-Wesley, 2002.

[110] M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[111] D. Winkler, R. Hametner, and S. Biffl, "Automation Component Aspects of Efficient Unit Testing," in *IEEE International Conference on Emerging Technologies and Factory Automation*, Palma de Mallorca, Spain, Sep. 2009.

[112] M. Karlesky and G. Willams, "Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns," in *Proceeding of the Embedded Systems Conference*, 2007.

[113] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems*. Springer Berlin Heidelberg NewYork, 2005, iSBN-10 3-540-26278-4.

[114] A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald, "Linking Model-Driven Development and Software Architecture: A Case Study," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 83–93, Jan. 2009.

[115] M. Mlynarski, B. Güldali, M. Späth, and G. Engels, "From Design Models to Test Models by Means of Test Ideas," in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa '09. New York, NY, USA: ACM, 2009, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/1656485.1656492

[116] P. Chevalley and P. Thevenod-Fosse, "Automated Generation of Statistical Test Cases From UML State Diagrams," in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, 2001, pp. 205 –214.

[117] C. Seidner and O. Roux, "Formal Methods for Systems Engineering Behavior Models," *Industrial Informatics, IEEE Transactions on*, vol. 4, no. 4, pp. 280 –291, Nov. 2008.

[118] F. A. M. do Nascimento, M. F. da S. Oliveira, M. A. Wehrmeister, C. E. Pereira, and F. R. Wagner, "MDA-based Approach for Embedded Software Generation From a UML/MOF Repository," in *Proceedings of the 19th annual symposium on Integrated circuits and systems design*, ser. SBCCI '06. New York, NY, USA: ACM, 2006, pp. 143–148. [Online]. Available: http://doi.acm.org/10.1145/1150343.1150383

[119] R. Hametner, D. Winkler, T. Östreicher, S. Biffl, and A. Zoitl, "The Adaptation of Test-Driven Software Processes to Industrial Automation Engineering," in *8th IEEE International Conference on Industrial Informatics*, Osaka, Japan, 2010.

[120] K.-Y. Cai, "Optimal Software Testing and Adaptive Software Testing in the Context of Software Cybernetics," *Information and Software Technology*, vol. 44, no. 14, pp. 841 –855, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584902001088

[121] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/1353673.1353681

[122] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One Evaluation of Model-based Testing and its Automation," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 392–401. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062529

[123] P. Fröhlich and J. Link, "Automated Test Case Generation from Dynamic Models," in *Object-Oriented Programming 2000 - ECOOP*, E. Bertino, Ed. Springer Berlin Heidelberg, 2000.

[124] Y. Kim, H. Hong, D.-H. Bae, and S. Cha, "Test Cases Generation from UML State Diagrams," *Software, IEE Proceedings* -, vol. 146, no. 4, pp. 187–192, Aug. 1999.

[125] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*, ser. Embedded Systems. Springer, 2010. [Online]. Available: http://books.google.at/books?id=EXboa4sXlRsC

[126] D. Simon, *An Embedded Software Primer*, ser. An Embedded Software Primer. Addison Wesley, 1999, no. Bd. 1. [Online]. Available: http://books.google.at/books?id=xG2ZD55_BJAC

[127] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publisher, 1997.

[128] A. Zoitl, *Real-Time Execution for IEC 61499*. Durham, North Carolina, USA: International Society of Automation (ISA), 2009.

[129] B. P. Douglass, *Doing Hard Time: Devloping Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Boston: Addison-Wesley, 1999.

[130] J.-M. Berge, O. Levia, and J. Rouillard, Eds., *High-Level System Modeling: Specification and Design Methodologies*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[131] T. Yu, "Testing Embedded Systems Applications," Master's thesis, The Graduate College at the University of Nebraska, 2010.

[132] E. Bringmann and A. Krämer, "Model-Based Testing of Automotive Systems," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, Apr. 2008, pp. 485 –493.

[133] E. Lehmann, "Time Partition Testing - Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen," Ph.D. dissertation, University of Technology Berlin, Nov. 2003.

[134] J. Bieman and J. Schultz, "Estimating the Number of Test Cases Required to Satisfy the all-du-paths Testing Criterion," *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 8, pp. 179–186, Nov. 1989. [Online]. Available: http://doi.acm.org/10.1145/75309.75329

[135] K. Sayre and J. Poore, "Stopping Criteria for Statistical Testing," *Information and Software Technology*, vol. 42, no. 12, pp. 851 – 857, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584900001105

[136] B. Böhm, "Software Engineering Economics," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 1, pp. 4–21, Jan. 1984.

[137] D. Hoffman, "Cost Benefits Analysis of Test Automation," in *STAR West 1999*, 1999.

[138] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the Cost of Software Quality," *Commun. ACM*, vol. 41, no. 8, pp. 67–73, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/280324.280335

[139] M. Voak, "Entwurf und Entwicklung einer parametrisierbaren Steuerung eines Testfall-Generierungs-Tools auf Basis von datenorientierten Testverfahren," Master's thesis, Vienna University of Technology, Institut für Rechnergestützte Automation, Sep. 2007.

[140] S. Fraser, D. Astels, K. Beck, B. Böhm, J. McGregor, J. Newkirk, and C. Poole, "Discipline and Practices of TDD: (Test Driven Development)," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 268–270. [Online]. Available: http://doi.acm.org/10.1145/949344.949407

[141] M. Blackburn, R. Busser, and A. Nauman, "Interface-Driven Model-Based Test Automation," in *International Conference On Software Testing Analysis and Review*, 2004.

[142] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Test Case Generation Approach for Industrial Automation Systems," in *The 5th International Conference on Automation, Robots and Automation*, Auckland, New Zealand, 2011.

[143] D. Winkler, R. Hametner, T. Östreicher, and S. Biffl, "A Framework for Automated Testing of Automation Systems," in *IEEE International Conference on Emerging Technologies and Factory Automation*, Bilbao, Spain, Sep. 2010.

[144] E. Komova, "Automated Software Testing in Machine Automation," Master's thesis, Lappeenranta University of Technology, Faculty of Energy Technology, 2011.

[145] S. Vegas, N. Juristo, and V. Basili, "Maturing Software Engineering Knowledge through Classifications: A Case Study on Unit Testing Techniques," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 551 –565, Jul. 2009.

[146] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York: ACM Press, 2002.

[147] C. Sünder, A. Zoitl, J. Christensen, H. Steininger, and J. Fritsche, "Considering IEC 61131-3 and IEC 61499 in the Context of Component Frameworks," in *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, Jul. 2008, pp. 277 –282.

[148] M. Wenger, M. Melik-Merkumians, I. Hegny, R. Hametner, and A. Zoitl, "Utilizing IEC 61499 in an MDA Control Application Development Approach," in *IEEE Conf. on Automation Science and Engineering*, Aug. 2011, pp. 495 –500.

[149] H. Unbehauen, *Regelungstechnik II - Zustandsregelungen, digitale und nichtlineare Regelsysteme*. Braunschweig/Wiesbaden: Vieweg Verlag, 2000, iSBN 3-528-73348-9.

[150] G. Melnik, G. Meszaros, and J. Bach, *Acceptance Test Engineering Guide, Thinking about Acceptance*. Microsoft, 2009, vol. 1.

[151] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, Apr. 2001. [Online]. Available: http://doi.acm.org/10.1145/367008.367020

[152] S. Ambler, *Elements of UML 2.0 Style*. Cambridge University Press, 2005.

[153] S. Friedenthal, R. Steiner, and A. Moore, *Practical Guide to SysML: The Systems Modeling Language*. Elsevier, 2008.

[154] R. Hametner, D. Winkler, T. Östreicher, N. Surnic, and S. Biffl, "Selecting UML Models for Test-Driven Development Along the Automation Systems Engineering Process," in *IEEE International Conference on Emerging Technologies and Factory Automation*, Bilbao, Spain, 2010.

[155] A. Lüder, L. Hundt, and S. Biffl, "On the Suitability of Modeling Approaches for Engineering Distributed Control Systems," in *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, Jun. 2009, pp. 440 –445.

[156] G. Frey, R. Drath, B. Schlich, and R. Eschbach, "Entwicklung sicherer Steuerungsapplikationen mit Safety-Automaten," in *Kongress Automation, Baden-Baden, Germany*, 2012, pp. 47–50.

[157] PLCopen TC5: Safety Software Technical Specification, *Part 1: Concepts and Function Blocks*, Online Available; http://www.plcopen.org, Feb. 2006. [Online]. Available: http://www.plcopen.org/

[158] A. Zylberman and A. Shotten, "Test Language - Introduction to Keyword-Driven Testing," *Quality Assurance and Software Testing*, pp. 1–9, 2010.

[159] Rashmi and N. Bajpai, "A Keyword Driven Framework for Testing Web Applications," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 3, no. 3, pp. 8–14, 2012.

[160] M. Fewster and D. Graham, *Software Test Automation*. Addison-Wesley Professional, 1999.

[161] V. Vyatkin and V. Dubinin, "Refactoring of Execution Control Charts in Basic Function Blocks of the IEC 61499 Standard," *IEEE Transactions on Industrial Informatics*, vol. 6, pp. 155–165, 2010.

[162] A. Zoitl, T. Strasser, and A. Valentini, "Open Source Initiatives as Basis for the Establishment of new Technologies in Industrial Automation: 4DIAC a Case Study," in *IEEE International Symposium on Industrial Electronics (ISIE)*, Jul. 2010, pp. 3817 –3819.

[163] K. Thramboulidis, "Using UML in Control and Automation: A Model Driven Approach," in *Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on*, Jun. 2004, pp. 587 –593.

[164] 4DIAC, "Framework for Distributed Industrial Automation and Control." [Online]. Available: http://www.fordiac.org

[165] G. Čengić and K. Åkesson, "On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 145 –154, May 2010.

[166] I. Hegny, M. Wenger, and A. Zoitl, "IEC 61499 Based Simulation Framework for Model-Driven Production Systems Development," in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2010, pp. 1 –8.

[167] The Eclipse Foundation, "Xtend," Eclipse Documentation. [Online]. Available: http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01s05.html

[168] PLCopen Technical Committee 6, "XML Formats for IEC 61131-3, Version 2.01 - Official Release," PLCopen, Tech. Rep., May 2009.

[169] R. Hametner, A. Zoitl, and M. Semo, "Component Architecture for the Efficient Development of Industrial Automation Systems," in *6th IEEE Conference on Automation Science and Engineering Proceedings*, Toronto, Canada, Aug. 2010.

[170] X. Wang and P. Xu, "Build and Auto Testing Framework Based on Selenium and FitNesse," in *Proceedings of the Int. Conf on Information Technology and Computer Science*, 2009, pp. 436–439.

[171] IEC TC65/WG6, *Programmable controllers – Part 3: Programming languages*. Geneva: International Electrotechnical Commission (IEC), 1993.

[172] IEC 61131-5, *Programmable controllers - Part 5: Communications*. Geneva, Switzerland: International Electrotechnical Commission (IEC), Nov. 2000.

[173] IEC TC65/WG6, *IEC 61499: Function blocks for industrial-process measurement and control systems – Parts 1 to 4*. Geneva: International Electrotechnical Commission (IEC), 2004-2005.

[174] C. Sünder, A. Zoitl, and C. Dutzler, "Functional Structure-based Modelling of Automation Systems," *Int. J. Manufacturing Research*, vol. 1, no. 4, pp. 405–420, 2006.

[175] J. P. Thomesse, "Fieldbuses and Interoperability," *Control Engineering Practice*, vol. 7, no. 1, pp. 81 –94, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066198001403

[176] A. Davies, *Handbook of Condition Monitoring: Techniques and Methodology*, 1st ed. Springer, Dec. 1997, iSBN-10: 0412613204, ISBN-13: 978-0412613203.

[177] N. E. Fenton and M. Neil, "Software Metrics: Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 357–370. [Online]. Available: http://doi.acm.org/10.1145/336512.336588

[178] M. B. Younis, "Re-Engineering Approach for PLC Programs Based on Formal Methods," Ph.D. dissertation, University of Kaiserslautern, 2006.

[179] A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," *ACM Comput. Surv.*, vol. 10, no. 1, pp. 3–18, Mar. 1978. [Online]. Available: http://doi.acm.org/10.1145/356715.356717

[180] T. McCabe, "A Complexity Measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[181] T. J. McCabe and C. W. Butler, "Design Complexity Measurement and Testing," *Commun. ACM*, vol. 32, no. 12, pp. 1415–1425, Dec. 1989. [Online]. Available: http://doi.acm.org/10.1145/76380.76382

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, im Oktober 2013

Dipl.-Ing. Reinhard Hametner

Curriculum Vitae - Reinhard Hametner

Education

01/2009 – 10/2013	PhD-Student and scientific project assistant at the Vienna University of Technology - Automation and Control Institute
10/2001 – 11/2008	Study of Electrical Engineering at the Vienna University of Technology Main field: Automation and Control, Degree: Dipl.-Ing.
09/1995 – 06/2000	HTL St. Pölten, Electrical Engineering

Awards

2012	Best Presentation Award at the 38th Annual Conference on IEEE Industrial Electronics Society (IECON 2012)
2011	Scholarship for short research visits from the Vienna University of Technology
2010	Scholarship for short research visits from the Vienna University of Technology

Work Experience

09/2013 – now	Thales Austria GmbH, Safety Management
01/2009 – 08/2013	Research Assistant and Project Manager at Vienna University of Technology, Automation and Control Institute
01/2004 – 12/2008	T-Mobile Austria, Core Network Planning (part-time)
06/2005 – 11/2008	BFI Vienna, Teaching Electrical Engineering
07/1996 – 08/2003	Summer jobs: Brauunion Österreich, Teufl & Co GesmbH, Elektrik Lindwurm, and T-Mobile Austria

See detail information in:

- XING [23],
- LinkedIn [24], and
- ResearchGate [25].

[23] XING: https://www.xing.com/profile/Reinhard_Hametner
[24] LinkedIn: http://www.linkedin.com/in/reinhardhametner
[25] ResearchGate: https://www.researchgate.net/profile/Reinhard_Hametner

Talks with Proceedings Entry

Reinhard Hametner, Georg Schitter, Andreas Voigt, Alois Zoitl: *"Implementation Guidelines for Closed Loop Control Algorithms on PLCs"*; Talk: IEEE International Conference on Industrial Technology (ICIT), Cape Town, South Africa; 02-25-2013 - 02-28-2013, in: *"ICIT 2013 - IEEE International Conference on Industrial Technology"*, 2013, ISBN: 978-1-4673-4568-2.

Reinhard Hametner, Dietmar Winkler, Alois Zoitl: *"Agile Testing Concepts Based on Keyword-driven Testing for Industrial Automation Systems"*; Talk: 38th Annual Conference on IEEE Industrial Electronics Society (IECON), Montreal, Canada; 10-25-2012 - 10-28-2012; in: *"IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society"*, 2012, ISBN: 978-1-4673-2420-5; 3707 - 3712.

Reinhard Hametner, Benjamin Kormann, Birgit Vogel-Heuser, Dietmar Winkler, Alois Zoitl: *"Test Case Generation Approach for Industrial Automation Systems"*; Talk: 5th International Conference on Automation, Robotics and Applications (ICARA), New Zealand, Wellington; 12-06-2011 - 12-08-2011; in: *"Proceedings of the 5th International Conference on Automation, Robotics and Applications"*, 2011, ISBN: 978-1-4577-0328-7; 57 - 62.

Gottfried Koppensteiner, Reinhard Hametner, Rene Paris, A. Moser Passani, M. Merdan: *"Knowledge Driven Mobile Robots Applied in the Disassembly Domain"*; Talk: 5th International Conference on Automation, Robotics and Applications (ICARA), New Zealand, Wellington; 12-06-2011 - 12-08-2011; in: *"Proceedings of the 5th International Conference on Automation, Robotics and Applications"*, 2011, ISBN: 978-1-4577-0328-7; 52 - 56.

Benjamin Kormann, Birgit Vogel-Heuser, Reinhard Hametner, Alois Zoitl: *"Engineering Process for an Online Testing Process of Control Software in Production Systems"*;
Talk: IEEE International Conference on Emerging Technologies and Factory Automation, France, Toulouse; 09-05-2011 - 09-09-2011; in: *"Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)"*, 2011, ISBN: 978-1-4577-0018-7; 4 pages.

Monika Wenger, Reinhard Hametner, Alois Zoitl, Andreas Voigt: *"Industrial Embedded Model Predictive Controller Platform"*; Talk: IEEE International Conference on Emerging Technologies and Factory Automation, France, Toulouse; 09-05-2011 - 09-09-2011; in: *"Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)"*, 2011, ISBN: 978-1-4577-0018-7; 4 pages.

Monika Wenger, Martin Melik-Merkumians, Ingo Hegny, Reinhard Hametner, Alois Zoitl: *"Utilizing IEC 61499 in an MDA Control Application Development Approach"*; Talk: 2011 IEEE Conference on Automation Science and Engineering, Trieste, Italy; 08-24-2011 - 08-27-2011; in: *"2011 IEEE Conference on Automation, Science and Engineering"*, August 24-27,2011, Trieste, Italy, Proceedings, 2011, ISBN: 978-1-4577-1731-4; 495 - 500.

Reinhard Hametner, Dietmar Winkler, Thomas Östreicher, Stefan Biffl, Alois Zoitl: *"The Adaption of Test-Driven Software Processes to Industrial Automation Engineering"*; Talk: IEEE International Conference on Industrial Informatics (INDIN), Osaka, Japan; 07-13-2010 - 07-16-2010; in: *"Proceedings 8th IEEE International Conference on Industrial Informatics (INDIN)"*, 2010, ISBN: 978-1-4244-7299-4; 7 pages.

Reinhard Hametner, Alois Zoitl, Mario Semo: *"Automation Component Architecture for the Efficient Development of Industrial Automation Systems"*; Talk: 6th International Conference on Automation Science and Engineering (IEEE CASE 2010), Toronto, Canada; 08-21-2010 - 08-24-2010; in: *"6th annual IEEE Conference on Automation Science and Engineering (CASE)"*, 2010, ISBN: 978-1-4244-5448-8; 6 pages.

Reinhard Hametner, Dietmar Winkler, Thomas Östreicher, Natascha Surnic, Stefan Biffl: *"Selecting UML Models for Test-Driven Development Along the Automation Systems Engineering Process"*;
in: *"Proceedings IEEE Emerging Technologies and Factory Automation (ETFA)"*, 2010, ISBN: 978-1-4244-6849-2, 4 pages.

Dietmar Winkler, Reinhard Hametner, Thomas Östreicher, Stefan Biffl: *"A Framework for Automated Testing of Automation Systems"*; in: *"Proceedings IEEE Emerging Technologies and Factory Automation (ETFA)"*, IEEE, 2010, ISBN: 978-1-4244-6849-2, 4 pages.

Martin Melik-Merkumians, Monika Wenger, Reinhard Hametner, Alois Zoitl: *"Increasing Portability and Reuseability of Distributed Control Programs by I/0 Access Abstraction"*; Talk: IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Bilbao, Spain; 09-13-2010 - 09-16-2010; in: *"Proceedings IEEE Emerging Technologies and Factory Automation"*, 2010, ISBN: 978-1-4244-6849-2; 4 pages.

Monika Wenger, Reinhard Hametner, Alois Zoitl: *"IEC 61131-3 Control Applications vs. Control Applications Transformed in IEC 61499"*; Talk: 10th IFAC Workshop on Intelligent Manufacturing Systems, Lisbon, Portugal; 07-01-2010 - 07-02-2010; in: *"Preprints 10th IFAC Workshop on Intelligent Manufacturing Systems (IMS)"*, 2010, 6 pages.

Dietmar Winkler, Reinhard Hametner, Stefan Biffl: *"Automation Component Aspects for Efficient Unit Testing"*; Talk: IEEE International Conference on Emerging Technologies and Factory Automation, Mallorca, Spain; 09-22-2009 - 09-26-2009; in: *"2009 IEEE Conference on Emerging Technologies and Factory Automation (ETFA)"*, 2009, 978-1-4244-2728-4/1946-0759; 8 pages.

Talks without Proceedings Entry

Reinhard Hametner:
"IEC 61131-3 Model for Model-Driven Development"; Talk: *38th Annual Conference on IEEE Industrial Electronics Society (IECON 2012)*, Montreal, Canada; 10-25-2012 - 10-28-2012.

Reinhard Hametner:
"Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language"; Talk: *IEEE International Conference on Industrial Informatics (INDIN 2011)*, Lisbon, Portugal; 07-26-2011 - 07-29-2011.

Book Chapters

Reinhard Hametner, Benjamin Kormann, Birgit Vogel-Heuser, Dietmar Winkler, and Alois Zoitl: *"Automated Test Case Generation for Industrial Control Applications"*, *"In Recent Advances in Robotics and Automation"*, ser. Studies in Computational Intelligence, G. Sen Gupta, D. Bailey, S. Demidenko, and D. Carnegie, Eds. Springer Berlin Heidelberg, 2013, vol. 480, pp. 263-273, ISBN: 978-3-642-37386-2, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37387-9_20

Gottfried Koppensteiner, Christoph Krofitsch, Reinhard Hametner, David P. Miller, Munir Merdan: *"Application of Knowledge Driven Mobile Robots for Disassembly Tasks"*, *"In Recent Advances in Robotics and Automation"*, ser. Studies in Computational Intelligence, G. Sen Gupta, D. Bailey, S. Demidenko, and D. Carnegie, Eds. Springer Berlin Heidelberg, 2013, vol. 480, pp. 311-321, ISBN: 978-3-642-37386-2, [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37387-9_24

Diploma and Master Thesis

Reinhard Hametner: "Modellierung von Regelungsstrategien in einer event-basierten Echtzeitsteuerungsumgebung";
Supervisor: M. Vincze;
Technische Universität Wien, Institut für Automatisierungs- und Regelungstechnik, 2008; final examination: 11-20-2008