Automation Component Aspects for Efficient Unit Testing

Dietmar Winkler
Vienna University of Technology
Inst. of Software Technology
Favoritenstr. 9-11/E188
A-1040, Vienna, Austria
Dietmar.Winkler@tuwien.ac.at

Reinhard Hametner
Vienna University of Technology
Automation and Control Institute
Gußhausstr. 27-29/E376
A-1040 Vienna, Austria
Hametner@acin.tuwien.ac.at

Stefan Biffl
Vienna University of Technology
Inst. of Software Technology
Favoritenstr. 9-11/E188
A-1040, Vienna, Austria
Stefan.Biffl@tuwien.ac.at

Abstract

Automation systems software must provide sufficient diagnosis information for testing to enable early defect detection and quality measurement. However, in many automation systems the aspects of automation, testing, and diagnosis are intertwined in the code. This makes the code harder to read, modify, and test. In this paper we introduce the design of a test-driven automation (TDA) component with separate aspects for automation, diagnosis, and testing to improve testability and test efficiency. We illustrate with a prototype, how automation component aspects allow flexible configuration of a "system under test" for test automation. Major result of the pilot application is that the TDA concept was found usable and useful to improve testing efficiency.

Key words: Test-Driven Automation, Test automation, Automation Component, Automation Software Development.

1. Introduction

An increasing part of added functionality in modern automation systems is implemented in software. Thus, software components become more complex. Systems requirements may change even late in the development process, lead to ad-hoc modifications of the product, and require systematic testing approaches. In current automation systems development products, software code and testing code is often intertwined in the code, which hinders efficient and systematic testing. Thus the code is hard to read and to modify during development and maintenance. Diagnosis functions are preconditions to evaluate the current status for a system under development and are the basis for defect prediction during operation and maintenance of a system [15]. Similar to test functionality, diagnosis aspects are typically scattered unsystematically in the code, which hinders the integration of individual diagnosis aspects into a comprehensive diagnosis strategy. Thus, diagnosis components are treated as add-ons (e.g., by applying external measurement) and separated from the logical system behavior. Nevertheless, the early integration of testing and diagnosis aspects during systems development can increase product quality and project development efficiency significantly [19].

Testing variants of an individual component (e.g., in a component library) is an increasing challenge in systems automation design. Thus, frequent reconfiguration of components is a key issue in modern systems development [22]. Increasing software complexity, availability of automation component variants, and late changing systems requirements require flexible and efficient mechanisms for design and testing (e.g., efficient reconfiguration of components and frequent test runs). Automated and frequent testing – a common practice in business IT software development [5] – enables early defect detection and helps identifying side effects.

Following a systematic design approach in the automation systems domain, we identified three major aspects in the design of a testable automation systems component [13]: (a) *automation aspects* represent the logical behavior of the system, (b) *diagnosis aspects* refer to measurement of the current systems behavior, and (c) *testing aspects* enable the setup for automated unit test cases. Testing aspects also support scenarios which are typically hard to test, e.g., by setting the system into a certain error state, which could appear during operation in case of device errors.

In this paper we introduce a design that structures aspects for automation components with a focus on automation, diagnosis, and testing functions. The separation of functional, diagnosis, and testing aspects enable flexible system reconfiguration and supports auto-

mated testing approaches to improve testability and test efficiency. We illustrate the automation component aspects in a prototype showcase that shows how the automation component allows flexible configuration of a system under test for test automation.

The remainder of this paper is structured as follows: Section 2 presents the related work on unit test automation in software engineering as a basis for the application in the automation systems domain and presents the concept of the IEC 61499 standard with respect to the test-driven automation component (TDA component) prototype. Section 3 describes the research issues. Section 4 summarizes the concept for automated unit testing, introduces the design for the TDA component, and illustrates the interaction of the TDA component with other TDA components and hardware interfaces. We present a pilot application and discuss lessons learned and findings in section 5. Finally, section 6 concludes and identifies future work.

2. Related Work

This section summarizes related work on unit test automation in software engineering and introduces to the automation systems domain.

2.1 Automating Software Unit Tests

Testing in traditional IT Software development processes (e.g., sequential development approaches like the V-Modell XT) focus on an early definition of test cases (TCs) and a late execution of these TCs after designing and implementing the software product [3][8]. This test-first approach, e.g., early definition of TCs based on requirements, enables an increased understanding of the product and the application domain before implementation and early defect detection and correction if a TC cannot be defined properly [4]. The increasing need for flexibility fosters the application of agile development process approaches, e.g., SCRUM, to respond to frequent changing requirements [1]. Test-Driven development (TDD) is a common agile practice for business IT software development with focus on small and manageable iterations. The TDD approach consists of 4 major steps [4]: (1) *Think*. Select a specific requirement and implement the corresponding TC prior to the implementation of the functional behavior; (2) *Red Test Result*. Execute the TC. As there is no correct implementation the TC must fail; (3) *Green Test Result*. Ongoing implementation of the test-case related functionality and TC execution until the TC passes; (4) *Refactor*. Optimize the implementation design without changing functionality and execute TCs. After finishing step 4, select the next batch of requirements.

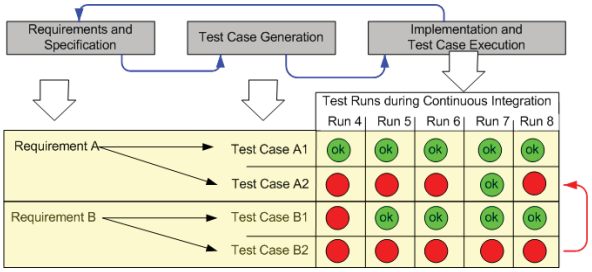

Figure 1: Test Execution Reports [19].

The application of this TDD approach leads to automated frequent testing and can be included in a continuous integration strategy [5]. Figure 1 presents a sample snapshot of TDD application in a business IT software development project. Requirements are broken down to TCs 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 TCs and/or requirements are affected by individual test runs negatively, can be identified easily. For instance, test execution of TC B2 (requirement B) has a negative impact on TC A2 (requirement A) in test run 8. Note that these short iterations allow the identification and removal of defects early.

Nevertheless, a continuous integration approach requires an appropriate test framework which enables frequent execution of a high number of TCs in a certain system setting [5]. Note that the test framework also includes mock objects to simulate not implemented systems behavior to enable automated testing of components or systems [13].

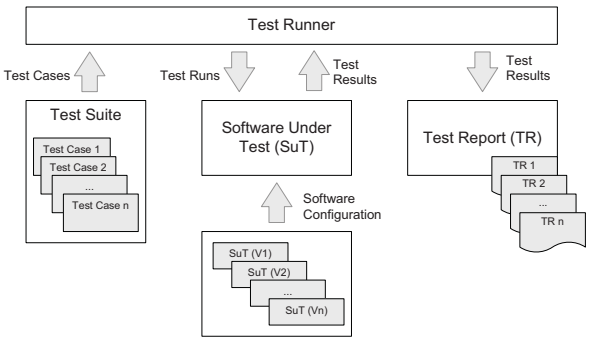

Figure 2: Test Framework for Automated Testing.

Figure 2 presents a schematic overview of a test framework, e.g., based on Unit Tests [6], for automated TC execution: A number of individual TCs are collected in a Test Suite. The Software under Test (SuT) is a configuration of the software and systems product (e.g.,

component versions and/or variants). The Test Runner applies the TCs to the SuT and provides fast feedback on the test results, which are presented by Test Reports (TR). Experiences from the development of business IT software system show that even small applications require a considerable number of TCs for sufficient test coverage. Test coverage refers to the degree of covering all code fragments by TCs [18]. In practice, the high effort to run these tests manually limits the testing intensity in a project with limited resources. Test automation encourages high testing intensity for each new software version. Nevertheless, the application of automated testing requires additional effort for the setup of the testing framework at the beginning. Project managers have to find a tradeoff between manual testing (similar – high – effort for every test run) and automated testing (higher effort for test framework implementation and lower effort for every additional test run).

However, in automation systems engineering the current observed practice relies mainly on manual and limited testing in a hierarchical systems design. Additionally, automation products are designed graphically (e.g., applying function blocks notation [10][21]) and with structured text to implement systems requirements. Based on these current practices the test coverage seems to be rather low because of insufficient testing tool support. The test framework, adopted from business IT software development can help engineers in the automation systems domain to increase test intensity in a more efficient way. The framework for test automation allows a flexible configuration of components for individual test runs. Nevertheless, a systematic component approach is required to support test automation, i.e., need for capturing information from various test runs (diagnosis aspects) and need for setting up TCs efficiently (testing aspects).

2.2 Concepts of the IEC 61499 Standard

In contrast to models for business IT software development function blocks (FBs) are common practices for modeling systems behavior in the automation systems domain [22]. The IEC 61499 is a standard for Industrial Process Measurement and Control Systems (IPMCS) defining a function-block-oriented paradigm for distributed systems development. The standard includes several models – application models, system models, device models, resource models, and FB models – which allow developing distributed control applications using a graphical approach [10][12]. A FB, the base model of the IEC 61499 family, is a software component, which is self-contained and allocates its functionality through a determined interface. For instance, this approach can be used to activate an actuator. Resultant, the speed of processing the system configuration can be increased and it brings more clarity to the software program. It can build simple or sophisticated solutions using FBs or parts of software, each containing particular algorithms, to configure a solution without the need for programming from scratch [20]. The model interface has been expanded from the interface of the IEC 61131-3 standard and consists of two parts: the event and the data interfaces. The data interface consists of data in- and outputs, which are already presented at the FB in IEC 61131-3. The interface is extended with an additional event interface. A trigger on the event inputs starts the execution of the FB. During execution of the FB the input data will be processed and the output data will be generated. Additionally, an event output can be triggered.

2.3 Reconfiguration Approach

Current production automation systems need to be flexible, adaptable, and have to allow for rapid changes within short time intervals at low cost. The down-time of a system, caused by system failures, changes and maintenance tasks, is an important cost driver in flexible production automation systems. Thus, there is a need to support reconfiguration and maintenance tasks with low or limited impact on the overall systems availability. The concept of dynamically reconfigurable control software components is a promising approach to increase flexibility of systems and allows efficient modifications and changes of a system with limited impact on systems availability [22]. Examples for required dynamic reconfiguration of automation systems are: (a) Modifications of functional behavior of the control program; (b) Control program extensions to achieve added value of the system; (c) Maintenance activities in terms of replacing components; (d) Modifications of parameters and software components; and (e) Shifting software parts to alternative devices (e.g., on distributed control systems.

Nevertheless, a flexible automation systems design is required to (a) support these types of automation system maintenance and (b) enable automated test runs (e.g., regression testing) after systems modifications.

3 Research Issues

Frequent and changing requirements and the need for flexible reconfiguration of components and systems require efficient testing mechanisms, e.g., provided by automated testing approaches from business IT software development. Nevertheless, a structured systems design, i.e., separation of functional, diagnosis and test functionality and efficient interaction mechanisms in a hierarchical systems design is a precondition for successful test automaton framework application. Thus, the goal of

automating testing in the automation systems domain leads to two research issues:

(a) *How can we adapt the Test Framework for automated testing from business IT software development to the automation systems domain?* This research issues focuses on the design and implementation of an automated testing framework to enable frequent TC execution in various test runs. As reconfiguration is a key issue in the automation systems domain, the testing framework has to address flexible and component-oriented systems design.

(b) *Which component structure is required to enable flexible and reconfigurable automation systems and supports automated systems testing?* Our observation in automation systems projects has been that functional, diagnosis, and testing aspects are intertwined in the software code and hinders efficient development, maintenance, and testing. Thus, a strict separation of individual aspects with well-defined interfaces for systems and component interaction is expected to increase maintainability, testability and testing efficiency.

Further, FBs according to the IEC 61499 standard represent the state-of-the-art in important parts of the automation systems domain. Thus, both research issues have to consider the implementation of components, which are based on the IEC 61499 standard.

4 Component-based Test Automation in the Automation Systems Domain

The application of the test framework from business IT development enables an automated TC execution based on various systems variants. This flexibility can support (a) flexible arrangement and reconfiguration of individual automation components and (b) frequent TC execution.

4.1 Unit Test Framework for Automation Systems

Based on the unit test framework, derived from business IT software development and presented in section 2.1, we identified similar components for the test framework in the automation systems domain: (a) a Test Runner for test management, (b) Test Suites consisting of sets of individual TCs, (c) Software/Systems under test (SuT), and (d) Test Reports. The Test Runner applies relevant TCs, derived from the Test Suite, to the SuT and provides test reports for every test run (see Figure 1 for a test report overview from a sample business IT software project). Figure 3 illustrates the application of the unit test framework from business IT software development to the automation systems domain. Note that the SuT consists of a TDA component (or assembled TDA components) encapsulating separated functional aspects, diagnosis aspects, and testing aspects with defined interaction via interfaces (see section 4.2 for TDA component details).

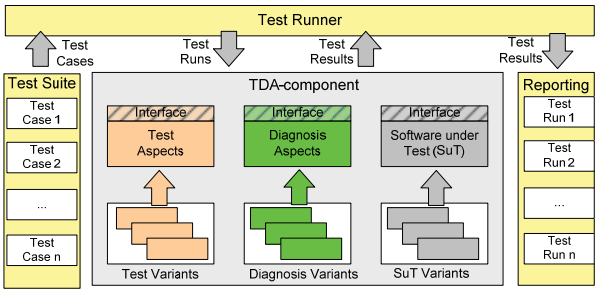

Figure 3: Test Framework for Automated Testing in the Automation Systems Domain.

The configuration of the SuT is a notable benefit of the TDA approach, because every sub-component of a TDA component, i.e., functional, diagnosis, and test aspects, can be flexibly combined as needed by the application. As defined interfaces are used for communication and data exchange within the TDA component (*internal communication*) and for interaction with other TDA components and/or sensors and actors (*external communication*) within a hierarchical systems design, the individual components can be exchanged and tested easily. However, missing components must be mocked in sufficient detail to simulate required behavior which has not been implemented so far. Mock objects [11] are common practices in business IT software development.

The application of the suggested test framework for automated testing of automation systems enables systematic, frequent test runs and is a precondition for continuous integration [5] approaches in the automation systems domain.

4.2 Test Automation Component Aspects

A strict separation of functional behavior (automation component), diagnosis functions (diagnosis component), and test functions (test components) enables a flexible exchange of variants for individual components. Figure 4 illustrates an example and a more detailed view on the TDA component and its interaction, including functional aspects and interaction via defined interfaces within a hierarchical systems design. Note that the sensor and the valve are also represented as TDA components including a similar component design.

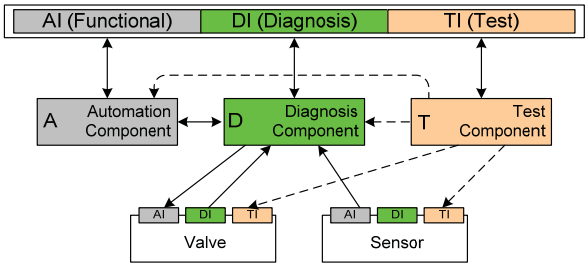

Figure 4: Test-Driven Automation Component.

Basically, the individual components interact by exchanging data via the corresponding interfaces. The interfaces allow the communication to higher-level TDA components as well as to lower-level TDA components. Because of the defined interfaces the component variants are seamlessly exchangeable. Figure 4 presents an example: The automation component (A) gets instructions from a higher level TDA component via the functional interface (AI). After processing, the instructions are passed via the diagnosis component (D) to the lower-level TDA component, in our case the valve. The diagnosis component obtains measurement data via the diagnosis interface of the valve and forwards diagnosis information to (a) the automation component (same level) to respond to instructions and (b) via the diagnosis interface to a higher level component. Regarding testing functionality, the test component (test function) is able to set the system in a certain state to check the response on defined system states, typically error states that are typically hard to reach. For instance, the testing function can simulate a blocking valve or wrong sensor data, which can be addressed in the test framework.

5 Test-Driven-Automation Component applied to Distributed Control Systems

This section summarizes the results of a pilot application (a simple triangle generation application) of the TDA component based on a distributed control system and is realized with the IEC 61499 standard. Further, we illustrate the application of the suggested test automation framework for automated testing of this pilot application.

5.1 Requirements and Specification

This pilot application describes the implementation of a triangle generator with TDA components designed according to the IEC 61499 standard. Major requirements include counting up to a predefined limit value and – after reaching the limit – counting down to a predefined offset value. We selected this kind of application to demonstrate time-dependent systems behavior with stateful characteristics within these time intervals. Figure 5 illustrates the expected systems behavior of the triangle generating system to derive parameters for TCs.

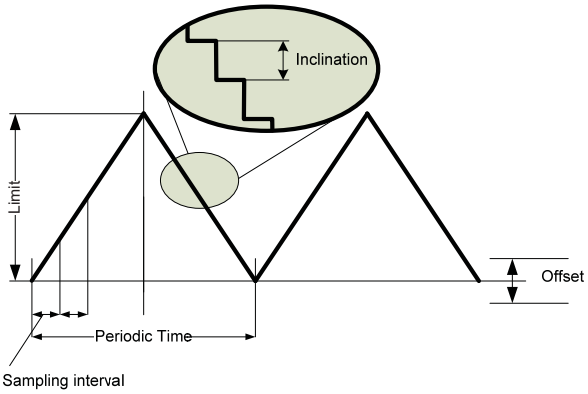

Figure 5: Time Sequence Diagram of a Triangle Generator with the Parameters.

Figure 6 presents the TDA component in IEC 61499 function blocks (FB) design. The TDA component consists of a tester FB, a triangle generator FB, and a diagnosis FB realized as composite FBs. These composite FBs comprise a network of basic FBs as their encapsulation enhances clarity of the design. In more detail, the FB diagram consists of three major entities:

FB 1. "Triangle_Gen" represents the functional interface (e.g., automation interface of the TDA component) and generates the curve of a triangle. Starting with an event input on "INIT" (initialization) the FB confirms on the event output "INITO". The calculation of the FB starts with the event input "START". At the end of one calculated cycle, an event output will be sent on indication "IND". Additional data input parameters are offset, periodic time, sampling interval, and limit value, which are illustrated in Figure 5. After finishing calculation, the output data are available. Note that the present value (PV) represents the actual value of the inclination. If PV is zero, the Boolean value Q0 is true, otherwise Q0 is false. If the value of PV is higher than the limit value, the Boolean value Q is true.

FB 2. "Tester" (via testing interface) defines the predefined values offset, periodic time, sampling interval, and limit (see Figure 6) as its data output. The FB "Tester" emits predefined test data parameters to test the system and receives the measurement parameters PV, Q0, and Q from the triangle generator. These parameters in combination with the predefined values calculate the state of the triangle generator. That means that the behavior of the system under test can be manipulated by the predefined parameter values. The FB "Tester" is initialized by the event input "INIT". At this

time the predefined values like limit are provided at the data output and this is confirmed by the event output "INITO". The event input "REQ" is triggered after the complete initialization of all FBs and confirmed by the event output "CNF".

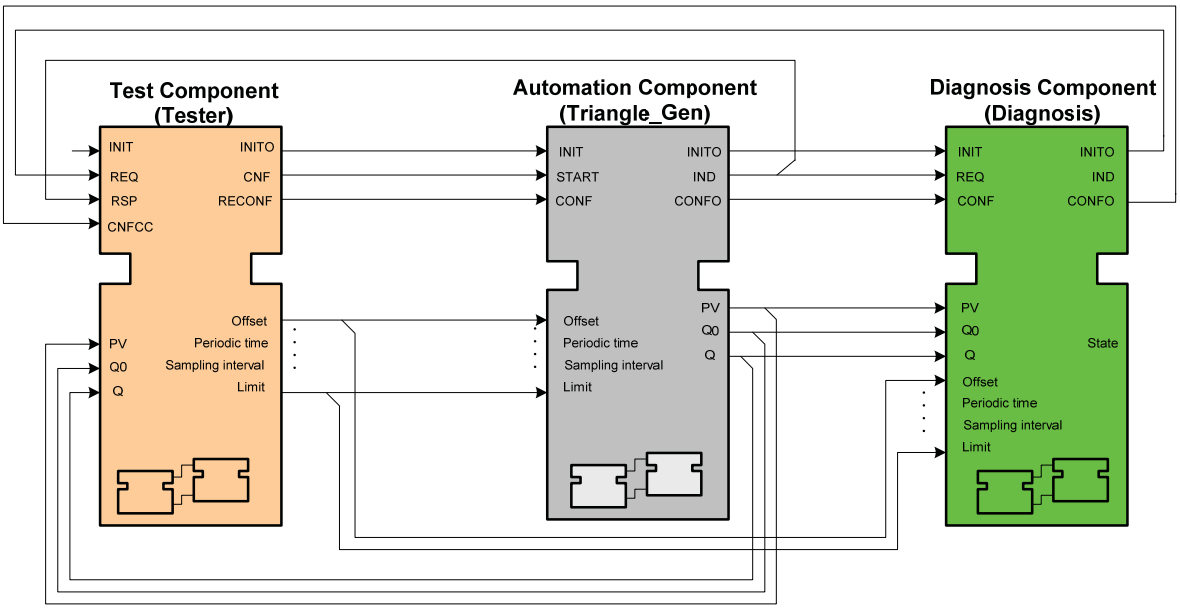

Figure 6: Sample Application with applied Function Blocks in IEC 61499.

Table 1: Sample Test Cases for Triangle Generator Application.

No.	Description	Level	Type*	Pre-condition	Input	Expected Result	Post-condition
1	Set Limit to a defined value	Comp.	NC	System running	Upper limit = 100	Limit set to 100	Modified limit
2	Set Limit to a negative value	Comp.	EC	System running Limit = 100	Upper limit = -10	Error message	No modification of triangle limit (Limit = 100)
3	Upper Limit and offset are similar	Comp.	SC	System running, limit=offset=20	Upper Limit = 20 Offset=20	Constant signal System warning.	Limit and offset = 20 Warning displayed
4	No limit signal. Missing connection between *Tester* and *Triangle_gen*.	Comp.	EC	System running current limit=20	Upper limit = 100	Error message	No modification of the limit (Limit=20), error message

* Note that test cases should include normal test cases (NC), special test cases (SC) representing systems behavior in the border area of regular systems behavior, and error cases (EC) for testing the error states of the system design.

FB 3. "Diagnosis" (diagnosis interface) gets all values from the tester FB and the triangle generator FB. The main task of the diagnosis component is to log all changes of the system, in this case the triangle generator, and to define independently and automatically the important variances for reporting them to other components (e.g., to a higher level). Therefore the time jitter of the sampling interval can be measured and the inclination failure can be detected. The diagnosis FB needs to be initialized with the event input "INIT", which is confirmed by the event output "INITO". All changes of the triangle generator are triggered by the event input "REQ". If the internal execution of the diagnosis component is finished an output event "IND" is sent. This design enables efficient TC generation as a pre-condition for automated testing.

5.2 Test Automation and Test Cases

Traditional TCs in the automation systems domain are defined and executed manually. For instance a modification of the graphical representation (see Figure 6) describes a certain TC. In context of automated testing using a testing framework, a test runner provides the setup of individual test scenario. For instance, individual TC variants can be exchanged easy (e.g., parameter change). Nevertheless, the variation of automation and diagnosis components require an online reconfiguration

system. The test runner enables an automated TC organization and execution of individual settings. Because of configuration options, various settings can be executed automatically. Selected TCs are applied to SuT configuration automatically and test reports are generated based on the current configuration. Table 1 presents a set of selected TCs for testing the set-functionality of the upper boarder of the triangle functionality. The testing aspect provides various limits (normal cases, special cases, and error cases) with respect to the automation function and the diagnosis aspect reports current status information on the current systems behavior. The results of every test run is reported and documented in a test report.

A benefit of the TDA approach is the ability to handle various settings of (a) the System under Test and (b) the implementation of test settings (i.e., TCs). For instance, the stable interface of the TDA design enables the replacement of individual composite FBs, e.g., the underlying FBs of the triangle component as long as the interfaces (and the communication between various components) remain stable (reconfiguration of the current setting). Also, test setting variants can be executed automatically. The overall setup of the tests is controlled and managed by the Test Runner.

5.3 Reconfiguration of TDA Aspects and Components

To show the benefits of the TDA component approach and its ability to support reconfiguration, we applied a FB approach based on the IEC 61499 standard and used the capabilities of the FBDK [8] and 4DIAC-IDE [1] tools for modeling. For instance, the TDA approach enables a manual modification of parameters for the "Tester" FB. In context of this paper, modifying parameters refers to parameter exchange and not to a common system reconfiguration processes as described in [23]. In case of changing test parameters, the tests have to be stopped, initialized with modified parameters, and restarted (e.g., realized as event input "CONF" and event output "CONFO" within the automation component in Figure 6). Configuration in the automation systems domain requires an online configuration process including hardware and software components.

A second option for reconfiguration refers to the exchange of individual components, e.g., the diagnosis and/or automation components. This reconfiguration process includes an online reconfiguration involving software and hardware components (see section 2.3 for details) [7][8].

6 Conclusion and Further Work

The increasing need for flexibility in the automation systems domain and the trend to shift functionality from hardware to software solutions lead to increased complexity of software components. Up to now, code-and-fix approaches in software development of systems engineering hinder efficient development and maintenance processes. The increasing complexity and the observed systems engineering processes require more efficient and frequent testing approaches. Unfortunately, automation functions, testing functions, and diagnosis functions are largely intertwined in the code and tests are typically conducted manually (setup of TCs and test procedures) by using a graphical approach. Thus, each TC and TC variants are treated as a separate configuration (maybe even a separate "project"), which requires a high effort, i.e., similar effort for every test case definition and execution, in case of changes during development and maintenance.

Thus, we reported on the need for (a) separating automation, testing, and diagnosis aspects to enable systematic systems development and (b) to introduce the concept of automated testing in the automation systems domain to enable efficient verification and validation processes. Based on the experience from business IT software development, we presented (a) an approach for a test automation framework and (b) a test-driven automation (TDA) component design which enables efficient and flexible exchange of functional, testing, and diagnosis aspects including defined internal and external interaction. To illustrate these concepts, we showed the design of a TDA component using a small pilot application based on FBs according to the IEC 61499 and the application of the suggested test automation framework.

An interesting finding was that the individual FBs, describing functional, diagnosis and test aspects are represented by individual classes (e.g., in Java by applying FBDK or C) which enable the application of test automation practices from business IT development. Additionally, a configuration file (e.g., represented in XML format) is used to setup the configuration of an individual application. The nature of handling individual components and configurations in the automation systems domain enables (a) reconfiguration of systems if the interfaces are comparable and (b) can be tested automatically by applying a component-based test automation framework. The defined interaction via interfaces and the systematic separation of the TDA subcomponents enable flexible and exchangeable handling of the individual components in terms of reconfiguration. Additionally, the implementation of the test framework provides a promising approach for systematic and frequently automated test runs and enables efficient verification and validation.

As limitation of the proposed solution we see the application of the proposed approaches with legacy systems as the introduction and the technological change

may require considerable effort, e.g., extensive redesign which seems to be comparable to a new project. This approach might not be reasonable for an existing systems solution.

Further work includes (a) a refinement of the TDA component approach to learn details on the application opportunities, (b) the implementation and evaluation of the TDA component in a more complex automation systems solution, (c) the implementation of the purposed automated testing approach to investigate the benefits of automated testing in the automation systems domain, and (d) the investigation of the test automation framework in larger project contexts to verify the expected benefits.

Acknowledgements

We want to thank our partners from academia and industry in the logi.DIAG project for their valuable discussions and feedback. Parts of this work were funded by the Austrian Research Funding Agency (FFG) grant logi.DIAG (Bridge7-196929).

References

[1] 4DIAC, *Framework for Distributed Industrial Automation and Control - Runtime Environment and IDE*, http://www.fordiac.org, last access April 2009.

[2] Beedle M., Schwaber K.: *Agile Software Development with Scrum*, Prentice Hall, 2008.

[3] Biffl S., Winkler D., Höhn R., Wetzel H.: *Software Process Improvement in Europe: Potential of the new V-Modell XT and Research Issues*, in SPIP 11(3), pp.229-238, Wiley, 2006.

[4] Damm L.-O., Lundberg L.: *Quality Impact of Introducting Component-Level Test Automation and Test-Driven Development*, Proc. EuroSPI, 2007.

[5] Duvall M.P., Matyas S., Glover A.: *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007.

[6] Hamill P.: *Unit Test Framework*, O'Reilly Media, ISBN-10: 0596006896. 2004.

[7] Hummer O., *Downtimeless System Evolution: Current State and Future Trends*, IEEE 2007

[8] HOLOBLOC, Inc. *FBDK – The Function Block Development Kit*, http://www.holobloc.com, last access April 2009.

[9] Höhn R., Höppner S.: *Das V-Modell XT. Grundlagen, Methodik und Anwendungen*, eXamen Press, 2008.

[10] IEC 61499-1. *Function blocks - Part 1: Architecture. International Electrical Commission*, Geneva, 2005. Public Available Specification.

[11] Karlesky M., Williams G.: Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns, Proc. Emb. Systems Conf, CA, USA, 2007.

[12] Lewis R.: *Modelling control systems using IEC 61499 – Applying function blocks to distributed systems*. The Institution of Electrical Engineers, London, UK, 2001. ISBN 0-85296-796-9.

[13] logi.DIAG: Test-Driven Automation in Systems Environments, http://www.logidiag.at (last access: 16.06.2009).

[14] Karlesky M., Williams G.: *Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns*, Proc. Emb. Systems Conf, CA, USA, 2007.

[15] Nandi S., Toliyat H.A.: *Condition Monitoring and Fault Diagnosis of Electrical Machines*, Conf on Industry Applications, Phoenix, US, 1999.

[16] Sünder C., Zoitl A, Dutzler C.: *Functional Structure-Based modelling of Automation Systems,* Jounal of Manufacturing Research, 1(4), pp405-420, 2007.

[17] V-Modell XT Framrwork: http://www.v-modell-xt.de/ (last access: 16.06.2009).

[18] Whalen MW., Rajan A., Heimdahl MPE, Miller SP.: *Coverage Metrics for Requirements-Based Testing*, Proc. of Int. Symp on Software Testing and Analysis, Portland, US, 2006.

[19] Winkler D., Biffl S., Östreicher T.: *Test-Driven Automation – Adopting Test-First Development to Improve Automation Systems Engineering Processes*, to appear at EuroSPI, Madrid, Spain, 2009.

[20] Xia F., Wang Z. and Sun Y.: *Towards Component-Based Control System Engineering with IEC61499*, National Laboratory of Industrial Control Technology, Hangzhou, China, 2004.

[21] Zhang W., Diedrich C., Halang W.A.: *Specification and Verification of Applications Based on Function Blocks*, Computer-Based Software Development for Embedded Systems (LNCS 3778), Springer, 2005.

[22] Zoitl A.: *Real-time Execution for IEC 61499,* ISA, ISBN-10: 1934394270, 2008.

[23] Zoitl A.: *Dynamic Reconfiguration of Distributed Control Applications with Reconfiguration Services based on IEC 61499*, Proceedings of the IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications (DIS'06), 2006.