Electronic Communications of the EASST
Volume 56 (2013)

Proceedings of the Combined workshop on Self-organizing, Adaptive, and Context-Sensitive Distributed Systems and Self-organized Communication in Disaster Scenarios (SACS/SoCoDiS 2013)

Keeping Pace with Changes
Towards Supporting Continuous Improvements and Extensive Updates in Production Automation Software

C. Haubeck, I. Wior, L. Braubach, A. Pokahr, J. Ladiges, A. Fay, W. Lamersdorf

12 Pages

Guest Editors: Michael Zapf, Florian Evers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Keeping Pace with Changes

Towards Supporting Continuous Improvements and Extensive Updates in Manufacturing Automation Software

C. Haubeck*, I. Wior+, L. Braubach*, A. Pokahr*, J. Ladiges+, A. Fay+, W. Lamersdorf*

* Distributed Systems and Information Systems, University of Hamburg
Vogt-Kölln-Straße 30, 22527 Hamburg, Germany
{haubeck, braubach, pokahr, lamersd}@informatik.uni-hamburg.de

+ Automation Technology Institute, Helmut-Schmidt-University
Holstenhofweg 85, 22043 Hamburg, Germany
{wior, ladiges, fay}@hsu-hh.de

Abstract: Every long-term used software system ages. Even though intangible goods like software do not degenerate in the proper sense, each software system degenerates in relation to the everlasting changes of requirements, usage scenarios and environmental conditions. Accordingly, operational software is commonly situated in a continuous evolution process in which manually conducted modifications and adaptations try to preserve or reinforce its quality. Unfortunately, such an unmanaged evolution inevitably leads to a discrepancy between the obsolete originally documented requirements and the updated software itself. For this reason, our contribution presents a coherent vision of an anti-aging cycle that preserves (non-)functional requirements as explicit runtime artefacts. The fulfilment of these requirements is validated based on conditionally triggered online test cases. In order to achieve an enhanced semantic test coverage, these test cases are adapted by monitoring, analysing and learning typical system behaviours. To explain our vision in more detail and demonstrate the benefit of a managed software evolution, our anti-aging cycle is exemplarily applied on the domain of manufacturing automation.[1]

Keywords: Runtime testing, requirement validation, manufacturing automation

1 Introduction

Long-term used software systems suffer from degeneration and must be repeatedly adapted to avoid a progressive decrease of quality and productivity of the system [Le80]. Nevertheless, many of today's software development projects neglect gradual software aging, because of the general assumption that intangible goods like software never suffer from any wearing [En09]. Even though the software itself does not degenerate in the proper sense, this assumption is a common fallacy, since software always degenerates in relation to the everlasting changes of requirements, usage scenarios and its underlying infrastructure. Naturally, not every software system is equally affected by software aging, as a matter of fact systems with a relatively long life cycle are more affected. One suitable example of long-living and aging software are automation systems of production facilities. Production facilities have high investment costs

[1] The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) for the project "Forever Young Production Automation with Active Components" (“Design for Future” SPP 1593).

and hence are often operated many years or even decades. Such systems have short downtimes and are operated in high cost pressure. Therefore, frequently occurring changes of the hardware configuration or the functionalities must be carried out during a very restricted time period or even during operation. In addition, production facilities are generally separated in distributed components which are often embedded in specific technical environments. This distribution exacerbates software aging by making it more difficult to access system knowledge. Accordingly, operational software is commonly situated in a continuous evolution process in which manually conducted software modifications and adaptations are performed to satisfy changing requirements in order to preserve the existing quality [RB02]. In consequence, the phase in which value is added to software products is shifting from the development phase to later phases of the lifecycle [So12].
For this reason our contribution presents a coherent vision of how knowledge of the development phase can be represented in software artefacts to be used at runtime. This enables to perpetuate and use this information during the evolutionary enhancement of the system. In order to present this vision, Chapter 2 illustrates our targeted evolution of requirements and software, and the underlying hardware system by presenting two fundamental evolution scenarios with which system operators need to deal nowadays. Subsequently, an anti-aging cycle, which considers these scenarios and aims at providing a systematic evolution assistance at runtime, is introduced. Chapter 3 then discusses the current research by presenting related software engineering approaches. For a deeper understanding, Chapter 4 motivates and introduces the production automation domain as a suitable application domain for studying evolutionary systems. This chapter clarifies and demonstrates the benefits of our vision by relating the various phases of the anti-aging cycle to a concrete production facility. The final chapter draws the conclusions and presents further research goals.

2 Vision of a Managed Software Evolution

Since the decisive factor for evolution of operational software systems are general enterprise objectives and processes [HAN99], this contribution proposes to separate software evolution into two evolution scenarios originally inspired by Business Process Management [SS08]. The first scenario describes continuous improvements in which software systems are adapted at runtime. In this case modifications do not only include direct code changes, but also any other adaptation activities that influence the system behaviour. The second scenario deals with extensive updates which concern much more the internal structure of the software. These recurrently occurring updates often affect several functionalities by extending or modifying various aspects of the software. The main differences compared to continuous improvements are that extensive updates are typically performed concurrently to the actual operation and often include a planned requirements engineering process. The progress of software evolution highly depends on the considered system. Nevertheless, both scenarios are usually necessary, because without continuous improvements the software cannot keep up with the operational usage, and without extensive updates the constant evolution may be limited to an insufficient architecture or a fixed range of functionalities.
Regardless of the different evolution scenarios, the overall goal of any evolution must be to preserve the fulfilment of its requirements. Since an explicit documentation is generally considered as an annoying duty, changes of the requirements and of the software during operational phases are often represented only implicitly in the actual system behaviour. This leads to the conclusion that the software itself must take care of the fulfilment of its current

requirements, and is to allow changes of the behaviour only if these are explicitly intended. Self-modifying software has already made considerable progress; however in some operational domains like production automation, self-modification is still not accepted due to strict safety restrictions, the necessity of high expertise or the impossibility to project the system behaviour on code level. Therefore, this contribution aims at introducing a concept for self-adapting software under a (human) supervisor. This concept extends the *system under consideration* (i.e. the physical system and the current control system) by adding a *requirement-aware supervising system*. The latter allows for the specification, validation and learning of requirements based on its current state, while all crucial decisions remains in the (human) supervisor's responsibility.

The Anti-Aging Cycle

The main vision of this paper is to establish a (semi)automatic anti-aging cycle that prevents or at least counteracts the aging effect by establishing an adaptive monitoring and supervising mechanism at runtime. As depicted in Figure 1, the anti-aging cycle consists of four different phases which are needed to close the gap between originally specified requirements and the degree to which the current system usage still corresponds to these documented requirements.

Phase 1 - Explicit representation of requirements: In order to allow for a general evolution of requirements and software, the relevant knowledge must be identified, which is indispensable for the longevity of the system under consideration. Therefore this knowledge must be expressed in a requirement meta-model. In contrast to many existing approaches, the meta-model allows for domain dependent requirement extensions. In addition, the meta-model must provide means for evaluating their validity at runtime.

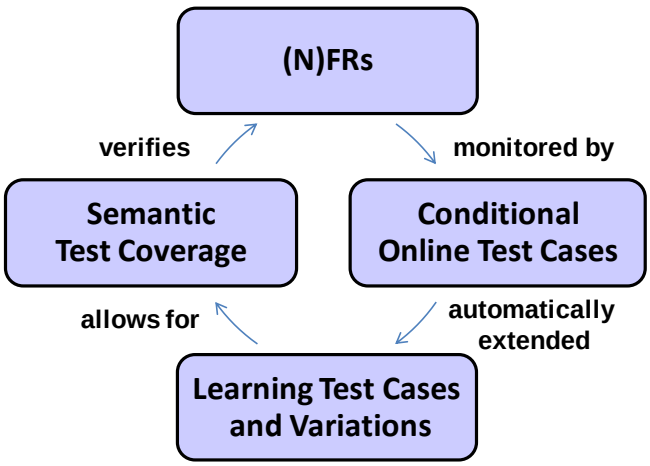

Figure 1: The Anti-Aging Cycle

Phase 2 - Validation of the fulfilment of requirements: To validate software systems different test cases have always been an essential and popular method. Hence, the second phase of the cycle tries to exploit the method of test cases by introducing conditionally triggered runtime test cases serving as a validation mechanism for their associated requirements. In this context test cases have a quite wide meaning, since any activity that compares an observed with a designated system behaviour can be a test case. The specification of test-based requirements allows for an online abnormality detection engine which

continuously observes the system behaviour in order to match actual behaviour against expected behaviour. Since our approach does not associate observed behaviours with specific code sections but reports detected abnormalities to be inspected, it leaves the responsibility of modification decisions in the hands of a human supervisor. It is important to highlight that this approach alone does not decide if an observed unknown and unforeseen behaviour is part of an intended code modification or if the system itself produces an unintended behaviour. This decision must be made by a supervisor which generally would be a human software operator, but can possibly be replaced by decision approaches like software agents. Nonetheless, the first two phases of the cycle provide a crucial support for adaption and evolution of software systems, because with explicit requirement artefacts and conditional triggered test cases the system can validate if its current requirements are still satisfied by the modified software.

Phase 3 - Learning and extension of test cases: Since test-based checks of requirements alone do not allow for both studied evolution scenarios, the software must be able to learn the system's typical behaviour and identify which test cases are most important. Therefore, the anti-aging cycle includes a supervised learning mechanism which on the one hand can analyse and compact system behaviour in order to generate new test cases, and on the other hand can parameterize and modify existing test cases. Such a learning mechanism must deal with huge amounts of information, which, therefore, must be pre-processed by filtering the relevant knowledge, disposing redundancies as well as abstracting and compressing information. Based on this pre-processed information various conditions and behaviours must be recognized. This recognition could for example be realised by (domain specific) methods for pattern detection and matching, data clustering or case-based reasoning techniques. The goal of the learning approach is to recognise implicitly contained system behaviours in order to express the characteristics of the system in adapted parameter ranges or new types of test cases. In conclusion, when a change is applied to the software, it can be tested against its designated behaviour (represented by manually generated test cases associated with the requirements) as well as against common usage scenarios of the software represented by the learned test cases.

Phase 4 - Enhanced test coverage: The combination of requirement validation at runtime and supervised learning of system behaviours in test cases leads to a semantic and adaptive test coverage which reflects and validates the current system behaviour. In order to close the anti-aging cycle, a link from learned test cases to requirements should be established by providing the learned results to a human operator, such that the system can understand how a successful or failed test case affects the software quality. This step can probably be automated by using (potentially domain specific) heuristics for proposing (or automatically establishing) mappings between learned test cases and requirements. As a result, the test case learning could be better directed towards requirements with poor test coverage. This approach supports both studied evolution scenarios. Continuous improvement is assisted by validating requirements with online test cases which detect relevant software modification due to resulting behaviour changes. In doing so, software operators are immediately and comprehensibly informed about intended as well as unintended requirement violations and (manual) counter measures can be taken to satisfy endangered requirements. Furthermore, this approach also assists with extensive updates, i.e. within a testing environment (for example a simulation model similar enough to the real system) all explicit specified or learned test cases of the real system can be performed on the updated software in order to check if the software is still operating in accordance to the original as well as new requirements.

General design of the anti-aging cycle: In order to clarify the connection between specification, validation and learning, Figure 2 shows the envisioned architecture of long-living software. Every software system contains specifications of requirements as system metadata following a domain-specific expended meta-model. For evaluation purposes, these requirements are provided with test cases to check their compliance with an evaluation runtime module. This module triggers conditional test cases and reports abnormalities to the human operator. If the observed behaviour is unintended, (manual) countermeasures can be initiated; otherwise the learning module can learn the intended behaviour (automatically). By extracting the test cases from real runtime data, the coverage of the software requirements is continuously kept up-to-date. This approach provides a further step towards systems that are aware of their requirements and enables a systematic solution for software evolution at runtime that serves as a good starting point for further software adaptation mechanisms.

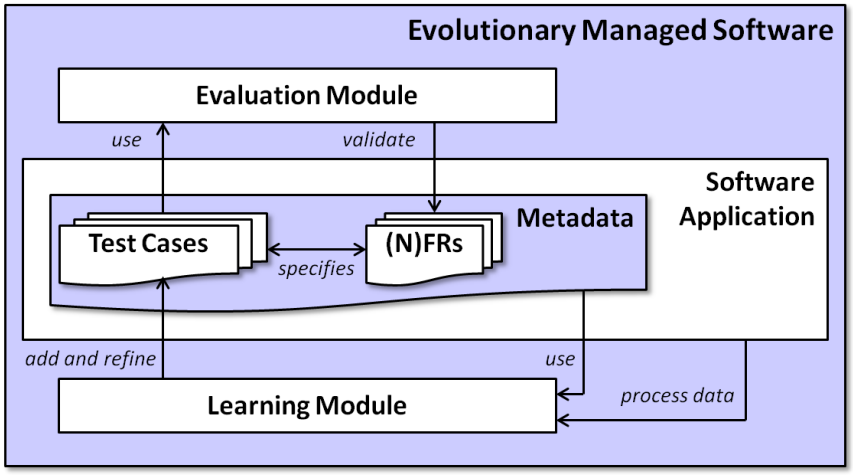

Figure 2: Envisioned architecture

3 Related Work

Initially the research in software evolution began with investigations performed to understand change processes in software code. Since then, various research fields have been developed providing suitable programming concepts for the evolution of software systems and system architectures, and the exchange of components in complex software systems [MD08]. Our approach focuses on a comprehensive handling and evolution of requirements knowledge at runtime. In the following, different software engineering approaches are discussed which aim at explicitly representing knowledge within software artefacts and using this knowledge to ensure software quality. Important categories are reengineering, requirements traceability and testing.

Reengineering aims at providing methods to analyse existing software as well as to adapt and further develop this software towards new requirements. The field of reengineering has a lot of different branches. For example, reverse engineering deals with tools to extract knowledge from existing software systems and its source code [CC90]. Refactoring and migration focus on restructuring, adaptation and improvement of existing software systems [Me04, Le03]. However, our contribution provides a more general approach in which these methods and tools

will be helpful, e.g. to provide possibilities to extract implicit system knowledge in order to generate test cases.
Furthermore, our approach deals with requirement engineering by aiming at establishing some kind of requirements traceability. According to [GF94] "requirements traceability refers to the ability to describe and follow the life of a requirement, in both forwards and backwards directions". In this way, requirements traceability can be seen as a knowledge carrying approach for evolutionary software and helps with quality assurance and maintenance. Unsolved problems with respect to requirements traceability mainly concern the missing standardization of requirement types and their characteristics, leading to heterogeneous languages and informally described requirements. On the one hand this complicates the systematic requirement elicitation, and on the other hand it hinders to a large extent the automatic runtime exploitation. In order to alleviate this drawback to some degree, it has been proposed to connect requirements to test scenarios [Eg03]. The test scenarios allow for a validation of the requirements as they represent executable software artefacts. This general approach has inspired our vision of an anti-aging cycle, which will follow this idea and extend it towards online test execution and automatic test learning.
In testing, a suite of test cases is executed in order to validate system behaviour. A test case itself represents a well-known type of software knowledge. Automated test cases represent an executable specification of software functionality and therefore implicitly contain knowledge about expected software behaviour. Although this knowledge is only implicit, it is directly accessible to developers by executing the test case and observing the result. The importance of testing for software evolution is e.g. highlighted by [Mo08]. Regression testing is an approach supporting software evolution through testing. A test suite representing previously correct system behaviour is executed against changed software, thus ensuring that such change does not break the existing functionality. A major problem of regression testing is, however, that the approach performance is directly dependent of the quality of the test suite. Developing a good test suite is often as complicated as building the software itself, especially if it is generated manually. Therefore, some approaches use techniques to generate test cases automatically. For instance, [BM08] proposes an approach for generating test data for improved test coverage. Moreover, in user interface testing tools allow to record interactions with the system and to use playback for testing against the recorded interactions. Behavioural regression testing [JOX10] uses the software itself to generate a test suite that directly resembles currently implemented system behaviour. Therefore, behavioural regression testing reduces defects when the software is changing, even when insufficient test cases are available beforehand. A disadvantage of the approach is that existing defects cannot be detected in this way. Model-based testing [AD97] allows for generating provable complete test cases, but requires a formal specification of all software requirements. Suitable specifications include finite state machines, finite state grammars or Markov chains. The opposite direction is approached in [WG11] by using learning techniques for extracting a system model from existing test cases, as a way of transferring the implicit knowledge of the test cases into explicit knowledge. The extracted formal model is then used to monitor the running system, therefore making sure that the system behaviour remains in line with its test cases, even when changes are applied to an already deployed system. Similar to our approach of an anti-aging cycle, here online monitoring is used to improve software quality. Yet, unlike our anti-aging cycle, learning is based on manually specified test cases instead of observed system behaviour and no relation to requirements is established.

4 Anti-Aging Cycle in Production Automation

The longevity of automation systems is increasingly important for the competitiveness of producing companies. Today's companies have to be able to react to changing customer requirements and environmental conditions. This includes changing demands for different products and changing conditions because of shorter product lifecycles [SJF11]. Automated production systems have to be able to follow these changes and hence are subjected to various modifications during their lifetime.
It is industrial practice to modify software in production machines directly in reaction to changing application requirements, this without proper documentation, requirements engineering approaches, and model-based or model-driven software engineering. As a consequence, the software becomes increasingly complex and difficult to manage which is why code modification is prone to errors, often resulting in production standstills and operational losses. The alternative approach, i.e. a complete re-programming of the machine, is not desirable either, since much technician expertise is implicitly included in the software and would be lost in case of re-programming. In order to counteract such problems, it is a major goal of software engineering research in production automation to develop methods which make the knowledge contained in this software accessible and manageable. Therefore, knowledge should be represented in the software system allowing for long-term maintenance and software development according to new and unforeseeable requirements.

4.1 Description of an exemplary anti-aging cycle for a production facility

In the following all four phases of the anti-aging cycle are applied on a concrete application example that demonstrates the benefits of our approach by motivating three adaption cases. In this example a medium-sized automotive supplier produces air tubes for installation in engine compartments in order to supply the engine with air. These tubes are mainly composed of rubber, are strengthened by fabric strings and are produced basically in three production stages arranged in sequence. The resulting facility initially can produce three tube variations.

Phase 1 - Explicit representation of requirements in production automation: In accordance to the first phase of the anti-aging cycle all relevant requirements need to be specified in a requirement meta-model. As the ISO/IEC standard 25010 [Is11] shows, a wide range of requirements for software systems exists. The meanings and priorities of requirements, however, differ substantially within each application domain. Thus, every requirement has to be interpreted within the context of an application domain and only within this domain requirements can be precisely described. In production automation, requirements are not only concerning the control software but also the plant's physics. Hence, the system including the software as well as the interacting physical plant needs to be considered as a whole when defining requirements. For distributed automation systems, the non-functional requirements (NFRs) analysability, testability, time behaviour and resource utilisation are of high importance [Fr11]. Further important NFRs are e.g. fault tolerance, scalability, reusability or adaptability [SJF11]. With regard to our demonstration example, the automotive supplier considers quality of the production and adaptability in its requirement model.

Phase 2 - Validation of the fulfilment of requirements in production automation: In the second phase of the cycle, the production system must be validated and check if it still fulfils all current requirements. While functional requirements (FRs) in production automation can often be directly validated, NFRs in production automation as well as NFRs in general are

difficult to evaluate, because many requirements are imprecisely described. In a first approach we try to decompose NFRs into corresponding process variables in order to make them automatically measurable (this is one of our further research topics). In the demonstration example, the quality requirement is measured by the absolute length difference between produced air tubes and its expected tube length. This fact can be checked in a conditional test case which can for example be triggered by a rule engine. Whenever this conditional test case fails, a quality requirement violation is reported. Within our approach a more advanced test case contains the possibility to be evaluated within a simulation application. For instance, the adaptability requirement can be tested by observing the production of a significant sample of air tubes with different lengths on a simulated production system. The sample shall cover lengths within the possible future customer´s demand. If the sample can be produced in simulation, then the requirement is fulfilled. To present the benefits of our approach we assume in a ***first adaptation case*** that one stage of the air tube production process was modified to decrease the production time. In such a scenario the conditional test cases should enable the system to evaluate if the modification is still in line with the quality and adaptability requirements and inform the operator about any violation.

Phase 3 - Learning and extension of test cases in production automation: Over time, the air tube supplier has to face changing market conditions and will deliver to further car manufacturers or will produce for new car models. Hence the supplier has to evolve and modify its production system, which probably already has grown over years, to meet new requirements. Some of the requirements could not be foreseen beforehand and surely new requirements will appear in the future which cannot be foreseen by now. The resulting continuous improvements and extensive updates are then considered in the third phase by applying test case learning techniques.

Software evolution by continuous improvements: In daily usage of the production system, errors are handled, improvements in the process are done, and it is reacted on changes of the requirements. This fact requires modifications and changes in hardware and software which are mostly performed directly by the production technicians. In terms of our example, the control software for the tube production is initially developed for a fixed set of tube lengths and therefore test cases are provided only for these known parameters. But in accordance to our ***second adaptation case,*** a new tube type is added to the portfolio and a technician changes the production system to include that tube length. This modification will lead to supposedly false tube length and a quality violation will occur. As a consequence a human operator is alerted (see phase 2) and classifies the changes as desired. Therefore, the quality test cases should learn the new parameter range without any changes to the control software itself.

Software evolution by extensive updates: The control software is developed for a constant winding density of the fabric strings. In the ***third adaptation case*** it is assumed that the supplier has to adapt its production in order to produce air tubes with a new winding density. This requirement was not foreseen beforehand but the control system has to be updated correspondingly. When applying extensive updates, it is important to consider if these changes will conflict with the previous requirements. Therefore, it is desirable to be able to test if a planned software update will limit the ability of the software to produce the current tube lengths or to adapt to other tube lengths. Based on the current test cases of the system, such extensive updates can be checked beforehand in a simulated environment of the considered system in which the updated software is tested against the new functionalities as well as the current requirements.

Phase 4 - Provide enhanced test coverage in production automation: The continuous improvements in phase 3 lead to an enhanced semantic and adaptive test coverage, which reflects current system behaviour more comprehensive and hence allows for a better validation of the specified requirements. It is desirable to be capable of connecting a learned test case to its corresponding requirement in order to address the non-fulfilment of the right requirement if a test case fails. While this probably can only be done manually, test cases can, in conjunction with domain specific heuristics, give valuable hints why a modification violates a system requirement. However, since test cases are the mayor knowledge carriers, it is even conceivable to utilize the anti-aging cycle without any predefinition of requirements. Instead, test cases learned during well operating production periods are used for future production periods (continuous improvements) or for the evaluations of planed software or hardware modification (extensive updates)

4.2 Results of the demonstration example based on a prototype implementation

In the following an initial case study based on a prototypical implementation is presented that demonstrates the feasibility of our anti-aging cycle along the three aforementioned adaptation test cases. Therefore, we carry out a first "proof of concept" prototype that is realised as an extension for active components using the Jadex middleware [BP12]. This prototype connects test cases triggered by a rule engine to the quality and adaptively requirement. The prototype also contains a simple mechanism to parameterise test cases out of observed variables. Furthermore, the underlying Jadex platform allows for a separated simulation environment in order to execute simulation-based test cases. It must be stated that the used prototype has only few simple and limited functionalities and it is used only for demonstration purposes. The example of an automotive supplier is realised by a simulation model that produces approximately one tube every minute.

Figure 4 shows the eight hour progress of the executed simulations presenting the absolute length difference (quality test) and the number of tests cases carried out. The simulation was repeated 30 times and the average values of these runs are indicated. The upper graphic illustrates the absolute length difference average ΔL between the produced length L_{prod} and the expected length L_{exp} of finished tubes, calculated according to (1)

$$\Delta L = \frac{1}{P}\sum_{i=1}^{P} \min_{j} \left| L_{\exp,j} - L_{prod,i} \right| \qquad (1)$$

where P is the number of products manufactured within 10 minutes, and $j=1,...,3$ corresponds to the three different quality test cases considered. The bottom graphic illustrates the number of test cases carried out. Here, each bar corresponds to the total test cases executed within 10 minutes. For instance, after 10 min of simulation, 30 tests cases were executed and a length difference average of 1 was obtained.

In addition, after every simulated hour, the control system checks for the adaptability requirement and executes a corresponding subset of 10 test cases on a separated simulation (crossed squares shown every 60 min in Fig.4). According to the *first adaptation case*, after a simulated time of 80 min the production is modified to decrease the production time. We assume in the underlying simulation model, that this modification causes a simultaneous increase in the absolute difference of the tube lengths. As expected the recurrent triggered

quality test cases notice this increase and after multiple test failures a requirement violation is reported. This violation is registered by the human operator, who therefore reverses the modification in minute 110.

The *second adaptation case* is initialised after 3.5 hours of operation. In this case a new tube length is introduced into the system. After indicating a quality violation, the human operator classifies this violation as intended. The software system therefore learns that a new tube length is intended and generates an additional test case. The additionally learned test case is evident in the bottom of the figure (starting from minute 250), since now 40 test cases per 10 min are carried out in average, instead of 30 as before. Apart from these continuous improvements in the *third adaptation case,* an extensive software update is initialised. In this case air tubes of a new winding density are demanded and hence the production shall be updated. To ensure that the updated software still fulfils the already existing requirements a simulation-based approach is used. Therefore, in a separated test environment (with a synchronised clock) the updated software is validated with the previously learned test cases. At the separated test environment a new test case is used to evaluate the new winding density functionality (see bottom figure starting from minute 300). Until the update is finally introduced (at minute 420), twice as many test cases are carried out as before, since they are executed in the still operating system (thin frames) and in the separated test environment (thick frames). In this way the update is concurrently validated and can afterwards be safely introduced into the system.

It needs to be emphasized that this first prototyped is only used to demonstrate the feasibility of our vision and the anti-aging cycle's ability to handle both considered types of evolution scenarios. More complex evolutionary and adaptive mechanisms will be part of our upcoming work. Nevertheless, in this contribution we have already shown that our approach of an anti-aging cycle can provide significant benefits towards requirement-aware systems and thus an increasingly managed evolution and adaptation of software systems.

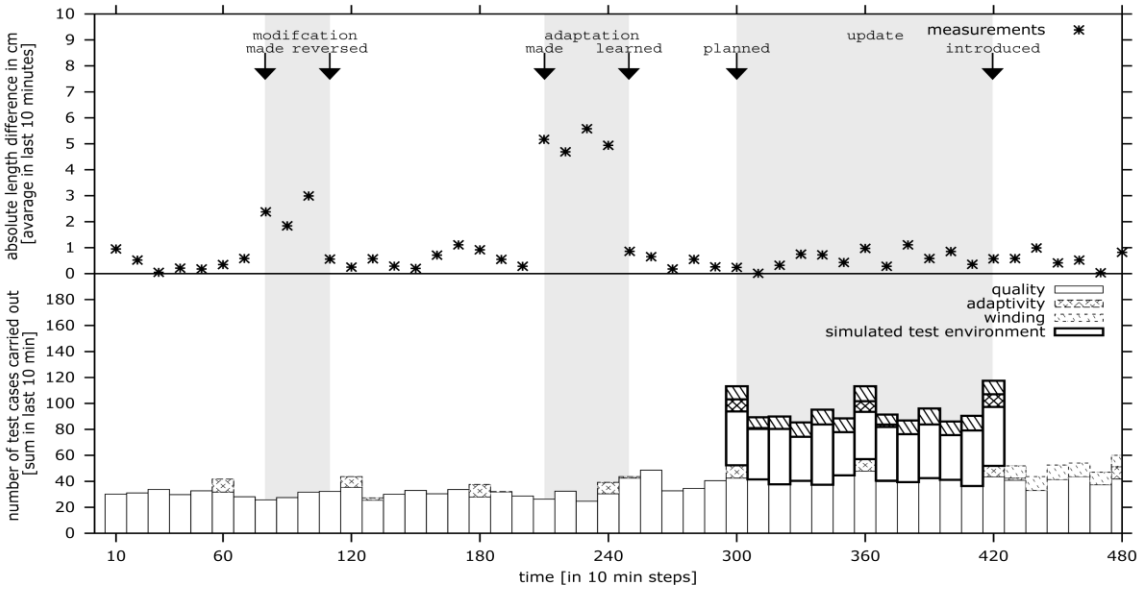

Figure 3: Results of the adaptation scenarios

5 Conclusion and Future Work

In this contribution we have presented an anti-aging cycle as a mechanism to manage the evolutionary enhancement of long-living software systems. In order to prevent evolving software from aging, our coherent concept supports two kinds of evolution scenarios, continuous improvements and extensive updates, and provides constantly support for keeping the implicit behaviour of the evolved software in line with its explicit requirements. First, a validation mechanism is used to detect requirement violations and hence to keep the system behaviour in line with the stored requirements. Furthermore, a learning process is applied to keep stored knowledge up-to-date and therefore to reach an enhanced test coverage to allow for improved validation mechanisms. To present the benefits of our vision, the anti-aging cycle is exemplary applied on a production facility. Further research towards this vision should include a comprehensive representation of requirements and test cases, an automatic mechanism to extract relevant system behaviour into test cases and finally a possibility to automatically link learned test cases to requirements.

6 References

[AD97] Apfelbaum, L.; Doyle, J.: Model Based Testing. In Software Quality Week Conference, 1997; S. 296-300.

[BM08] Beer, A.; Mohacsi, S.: Efficient Test Data Generation for Variables with Complex Dependencies. In Proc. of 2008 Int. Conf. on Software Testing, Verification, and Validation, 2008; S. 3-11.

[BP12] Braubach, L.; Pokahr, A.: Developing Distributed Systems with Active Components and Jadex. In Scalable Computing: Practice and Experience, 13(2), 2012; S. 3-24.

[CC90] Chikofsky, E.J.; Cross, J.H.: Reverse engineering and design recovery: a taxonomy. Software. In IEEE Software, 7(1), 1990; S. 13-17.

[Eg03] Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. In IEEE Transactions on Software Engineering, 29(2), 2003; S. 116-132.

[En09] Engels, G. et al.: Design for Future. Legacy-Probleme von morgen vermeidbar? In Informatik Spektrum, 32(5), 2009, S. 393–397.

[Fr11] Frank, T. et.al.: Dealing with non-functional requirements in Distributed Control Systems Engineering. In Proc. of 16th Conf. on Emerging Technologies and Factory Automation, 2011; S. 1-4.

[GF94] Gotel, O.C.Z; Finklestein, A.C.W.: An analysis of the requirements traceability problem. In Proc. 1st Int. Conf.on Requirements Engineering, 1994; S. 94-101.

[HAN99] Hegering, H.G.; Abeck, S.; Neumair, B.: Integrated Management of Networked Systems: Concepts, Architectures, and Their Operational Application. Morgan Kaufmann, 1999.

[Is11] ISO/IEC 25010, Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models, 2011.

[JOX10] Jin, W.; Orso, A.; Xie, T.: Automated Behavioral Regression Testing. In Proc. of 3rd Int. Conf. on Software Testing, Verification, and Validation, 2010; S. 137-146.

[Le03] Lee, E. et al.: A reengineering process for migrating from an objectoriented legacy system to a component-based system. In Proc of 27th Annual International Computer Software and Applications Conference, 2003; S. 336-341.

[Le80] Lehman, M.M.: On understanding laws, evolution and conservation in the large program life cycle. In: System and Software 1980 (1(3)), S. 213–221.

[MD08] Mens, T.; Demeyer, S.: Software Evolution. 2008.

[Me04] Mellor, S.J.: MDA Distilled: Principles of Model-Driven Architecture. 2004.

[Mo08] Moonen, L. et.al.: On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In Software Evolution, 2008; S. 173-202.

[RB02] Rausch, A.; Broy, M.: Evolutionary Development of Software Architectures. In: Technology for Evolutionary Software Development, 2002; S. 16-33.

[SS08] Schmelzer, H.; Sesselmann, W.: Geschäftsprozessmanagement in der Praxis, 2008.

[SJF11] Schreiber, S.; Jerenz, S.; Fay, A.: Anforderungen an Steuerungskonzepte für moderne Fertigungsanlagen. In Automation 2011, VDI-Bericht 2143, 2011; S. 7-11.

[So12] Sommerville, I.: Software Engineering, 2012.

[WG11] Werner, E.; Grabowski, J.: Model Reconstruction: Mining Test Cases. In Proc. of 3rd Int. Conf. on Advances in System Testing and Validation Lifecycle, 2011; S. 97-102.