Architecture-based Assessment and Planning of Software Changes in Information and Automated Production Systems

State of the Art and Open Issues

B. Vogel-Heuser, S. Feldmann, J. Folmer, S. Rösch
Institute of Automation and Information Systems
Technische Universität München
85748 Garching near Munich, Germany
{vogel-heuser, feldmann, folmer, roesch}@ais.mw.tum.de

R. Heinrich, K. Rostami, R. Reussner
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
{heinrich, rostami, reussner}@kit.edu

Abstract—**Information and automated production systems are long-living, evolvable systems. Consequently, modifications are performed to correct, improve or adapt the respective system. We introduce different approaches to analyze maintainability of software-intensive systems and propose two different case studies from the information and from the automated production systems domain as a basis to validate approaches on system evolution.**

Keywords-Modularity, Modeling, Architecture, Evolution

I. INTRODUCTION

Information systems (IS) and automated production systems (aPS) face very similar challenges. Both struggle with changing requirements and evolving platforms, as well as with designs and implementations that have grown over time. Frequently, IS and aPS are in operation over decades while they are continuously modified [1, 2]. Modifications may include corrections, improvements, or adaptations of the system to changes in its environment (e.g., hardware or software platform, technology, and user properties) as well as in requirements [3]. Thus, *maintainability* is an important quality aspect, especially for long-living systems. Maintainability is the ease with which a system (either IS or aPS) can be modified [4, 5]. Lehman and Belady [6] first applied the term *evolution* to software systems by referring to a long, broad sequence of changes to a system over its life-time [7]. For aPS, both anticipated changes to the aPS's architecture as well as unanticipated changes are referred to as evolution in the following [2]. IS and aPS comprise a variety of heterogeneous components consisting of electrical, mechanical or software artefacts. For maintainability analysis, it is important to understand the consequences of a given change request on the various heterogeneous system artefacts. Existing approaches, however, do not sufficiently consider system architecture descriptions and are mostly limited to software artefacts. Furthermore, there are mutual dependencies between software-intensive systems and their technical processes (automated by aPS) and business processes (automated by IS) [8, 9]. This makes the consideration of process and architecture elements as well as their dependencies necessary.

This paper introduces different approaches to analyze maintainability of software-intensive systems. We propose the derivation of maintenance task lists, i.e., specific changes to be performed, from architecture descriptions. Architecture-based maintainability cannot be empirically validated in a scientific way without the explicit use of case studies. Therefore, we propose two different case studies in this paper that are appropriate for conducting empirical research to validate approaches on system evolution as, to the best knowledge of the authors, there do not yet exist similar case studies that are already appropriate for this kind of evaluation. Section II introduces the term *architecture* for both kinds of systems before the state of the art and resulting challenges are discussed in Section III. We describe demonstrators for changes in the IS and aPS domain in Section IV. The paper is concluded in Section V.

II. INFORMATION AND AUTOMATED PRODUCTION SYSTEM ARCHITECTURE

According to [10], an IS consists of five component types: (1) Software components, which comprise all the programs and procedures, (2) hardware components, which comprehend all the physical devices of the system, (3) data, which includes all the knowledge in the system, (4) networks, which comprise all the communication channels, and (5) people, which cover system specialists and end-users. We focus on software architecture when talking about IS' architecture hereafter.

Software architecture complements a component definition, which focuses on the individual components and their interfaces [11]. From an automation perspective, according to [12] a process automation system may be separated into (1) the technical system that consists of sensors and actuators, (2) an automation system that comprises the automation platform and software as well as the communication system and (3) a human-process communication. Using the technical system, the automation system enables the realization of the technical process. The technical system is a mechanical system; the automation system includes electric and electronic devices. The architectural design principles differ depending on the

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works

discipline. Typically, Programmable Logic Controllers (PLCs) are used as control platforms in the automation domain, which are predominantly implemented in IEC 61131-3 languages [13]. Combining both definitions into the term automated production system (aPS), it can be concluded at the top level that an aPS' architecture is mainly characterized through its control system, which implements the technical process by determining and affecting the technical system's sensors and actuators and which provides the interface between human and technical process.

III. STATE OF THE ART AND CHALLENGES

Evolution of software-intensive systems – both IS and aPS – is supported by four key aspects. Modular architectures enable reuse (Section III.A). Architecture modeling notations reflect modular design (Section III.B). IS and aPS are variant-rich, making the specification of variability necessary (Section III.C). Change identification and management are crucial for the success of engineering projects (Section III.D)

A. Modularization

Modularity [14] enabled by a modular architecture allows reusing architecture elements [15]. Within the IS domain, the central idea of component-based software engineering (CBSE) [16] is to build complex software systems by assembling basic software components that are specified and implemented independently from their later context and that communicate via pre-defined interfaces to enable reuse. In the aPS domain such sophisticated modular approaches are not fully established. In particular "copy, paste and modify" is often applied for reusing automation control software [17, 18] or similarly in maintenance tasks, e.g., add, delete and modify [19]. The reasons for these shortcomings are manifold: On the one hand, Katzke et al. [18] as well as Jazdi et al. [15] identify conflicts of interest when choosing the correct module granularity as fine-grained modules increase reuse and flexibility, whereas coarse-grained modules improve efficiency and robustness. Choosing an appropriate granularity moreover depends on the engineering phase of the project [17], making the use of hierarchical models necessary that support coarse-grained modeling in early phases and fine-grained modeling in later phases. On the other hand, the modules' size affects reuse capabilities, as numerous small units make choosing appropriate modules and their combination difficult, whereas having a few large units hamper the modules' flexibility [15]. These challenges are exacerbated when modularizing aPS components considering all disciplines. While modules are often defined by the mechanical structure [20] or as mechatronic objects including the components from all disciplines [21], research has shown that a 1:1:1 relation between mechanics, electronics/electrics and software often is not applicable [22].

By providing a modular and reusable architecture, maintainability can be simplified as the systems' parts can easily be managed in central libraries by applying maintenance tasks to these parts. In summary, whereas sophisticated approaches for software modularization in the IS domain exist, an appropriate modularization within the aPS domain that covers the control system as well as the necessary aspects of disciplines is not yet available. Therefore, in order to enable support in estimating maintainability, the adequate size and granularity of modules and models must be investigated.

B. Architecture modeling

Software architectures in the IS domain can be described by two well-known basic concepts in software engineering – component models and architecture description languages. A component model is a standard for component specification, implementation, and deployment. An overview of various component models in software engineering is given in [23]. An architecture description language [24] is used to specify software architectures, i.e., the components of an IS and the interfaces between these components. However, both concepts only exhibit limited support for quality of service specification regarding components. The Palladio Component Model (PCM) [25] is a meta-model to specify component-based software architectures in a parametric way. Thus, the quality of a proposed architecture can be predicted.

The concept of architecture is manifold in automation, e.g., the different levels of a process automation system and the structure of the control nodes, i.e. centralized, decentralized or intelligent decentralized [12, 26]. A specific type of intelligent decentralized architecture is holonic architectures introduced by Leitão et al. [27]. Similar to IS, the concept of architecture is also used for the architecture of the automation software itself. Software engineering on the code level in aPS is mostly dominated by the languages of IEC 61131-3 [13] implemented on PLCs. Because of a centralized design the control software often shows a monolithic structure, which is difficult to maintain. Bonfè et al. [28] identified different patterns for hierarchical control, alarm handling and motion control applied in industrial control software for packaging machines. Vyatkin [29] introduced an IEC 61499-based system-level architecture. Although IEC 61499 [30, 29] provides improved maintainability due to its increased ability for software modularization, "IEC 61499 has [still] a long way in order to be seriously considered by the industry" [31].

To increase efficiency and reduce faults in software engineering, modeling languages based on the Unified Modeling Language (UML) [32] as well as adapted languages [34] have been proposed for the aPS domain. Moreover, especially Systems Modeling Language (SysML) [33] is applied for integrating the different views on the aPS within one single modeling notation [34, 31]. Approaches for managing the manifold models involved in engineering were proposed [28]. Transformations from UML-based software models to IEC 61131-3-compliant control software were developed [28, 35], as these languages have proven to be beneficial in terms of model comprehensibility, modularity and reuse potentials [36].

Contrary to the IS domain, different architectures address aspects such as simplifying reconfiguration or adaptation of software architectures in the aPS domain. A meta-modeling approach taking the aPS architecture for estimating maintainability into account has nevertheless not been described yet. Furthermore, in both domains mutual dependencies between the system and business or technical processes are not considered in maintainability estimation.

C. Variability modeling

Besides the need to specify and model the architecture of an IS or aPS, these systems face evolutionary challenges and struggle with changing requirements and architectures.

Moreover, because of multiple stakeholders involved in the (IS or aPS) engineering process, these systems are variant-rich, making the specification of these systems' variability necessary as a basis for identifying the change effort from one variant to the other. The advantages of variability models and product line technologies are well recognized in the computer science and IS domain [37]. Therein, various languages exist, e.g., [38, 39], enabling specification of the variability of a product line. Within feature models, program functionalities and their usage constraints in product lines are defined within hierarchically organized features [40]. A software product line refers to a set of products being distinguished in terms of features [41].

Although research in the field of product line technologies is quite sophisticated and user support is available, e.g. using tools [38, 42], verification technologies [43] and methods for measuring complexity [44, 45], variability modeling and product line technologies are up to now rarely investigated for improving the aPS engineering [46, 47, 48]. Some approaches address this drawback, e.g., by extending modeling languages for modeling variants in the aPS domain [49] and by applying product line variability for automating the maintenance and reconfiguration process of control software based on the IEC 61499 standard [50]. Moreover, product lines are used for embedded software design in the automotive sector [51] and for combining architecture models with product line technologies to generate IEC 61131-3-compliant code [52]. An automatic feature model synthesis for variant-rich simulation models was presented in [53], which, among other things, enables validation of variant-rich simulation configurations. Feldmann et al. [48] introduce discipline-specific developer feature models to model the different variability aspects of the subsystems.

To provide a variability modeling approach, the results of the IS domain must be adapted and integrated into aPS architecture models [48]. Hence, the joint investigation of variability is indispensable. This is the foundation for identifying the change effort from one variant to the other.

The adequate support of engineers in developing their software solution, on top of the methodological support for modular engineering and variability modeling, makes appropriate tools necessary [54]. Although some software tools already exist, e.g. EPLAN Engineering Center, Siemens COMOS Platform and CODESYS Application Composer, which enable the definition of modular units, there are still open challenges. Whereas EPLAN Engineering Center and Siemens COMOS Platform support interdisciplinary engineering aspects, CODESYS Application Composer only focuses on a PLC development perspective. First efforts towards integrating aspects concerning system variability were performed in these software tools. However, the definition of variable solution elements from both the customer's and the developer's perspective has not yet been completely investigated.

D. *Change effort identification and maintainability estimation*

Work related to change effort identification and maintainability estimation can be put into four categories.

1) Task-based Project Planning

Hierarchical Task Analysis (HTA) [55] is a method for decomposing a high-level task into a hierarchy of subtasks in a systematic and structured fashion. Obermeier et al. [36] applied HTA to PLC modeling as well as programming with Function Block Diagram and Sequential Function Chart, two of the IEC 61131-3 languages [13]. To evaluate the effort on the HTA level, detailed knowledge of the functionality to be maintained is necessary and therefore the design decisions need to be elaborated in detail. Therefore, in most cases, HTA is not appropriate to support the comparison of design alternatives before their realization. Decomposition is also addressed by the Goals, Operators, Methods, and Selection rules model (GOMS) [56]. The Keystroke level method [57] calculates execution time for an entire task by summing up the estimated times of the individual actions. Function Point Analysis (FPA) [58] estimates the size of a system by adding up the number and weights of all transaction and data elements. The Comprehensive Cost Model (COCOMO) II [59] comprises different approaches for cost estimation during requirements phase and architectural design phase by applying the abstract measure of function points (applications points in COCOMO) based on an informal requirements description. Yet, if used at all, these techniques only apply coarse-grained architectural artifacts which make accurate predictions difficult.

2) Architecture-based Project Planning

Conway's Law [60] incorporates software design and project organization by stating that the software architecture must reflect an organization's communication structures. Architecture-Centered Software Project Planning (ACSPP) [61] deems software architecture as an artefact in project planning while combining top-down and bottom-up effort estimation techniques. Based on experience in mechanical engineering, Carbon et al. [62] proposes a procedure to align product design and production planning in a software development context. The procedure allows for early identification of potential problems in production and for assessing the software architecture regarding completeness. Existing approaches on architecture-based project planning, however, do not support estimating change efforts based on a given architecture and do not allow for automated change impact analysis and derivation of change activities.

3) Architecture-based Software Evolution:

Garlan et al. [63] propose a pattern-based approach to assist in expressing architectural evolution and for reasoning about the correctness and quality of evolution paths. Naab [64] analyses software architecture maintainability at design-time though neglects operation and management tasks related to the architecture, such as component deployment. As discussed before, work on architecture-based software evolution does not support change effort estimation and impact analysis.

4) Scenario-based Architecture Analysis

Requirement changes evoke changes in the system, yet drawing inference from the extent of changes in requirements on the effort for implementing the changes is not possible without considering the system's architecture. Some existing approaches target scenario-based software architecture analysis, but lack a formalized architecture description or are limited to software development without taking management tasks into account. Software Architecture Analysis Method (SAAM) [65] evaluates software architectures regarding modifiability by using an informal architecture description (mainly the structural view).

SAAM gathers change scenarios and tries to find components affected by interrelated scenarios for cost estimation. In contrast to SAAM, the Architecture Trade-Off Analysis Method (ATAM) [65] identifies trade-offs between various quality aspects by considering the effect of architectural decisions. Architecture Level Prediction of Software Maintenance (ALPSM) [66] defines and weights scenarios to evaluate their impact on the overall maintenance effort based on component size estimations. ALPSM heavily depends on the expertise of the architects and provides little guidance through tool support. Architecture-Level Modifiability Analysis (ALMA) [67] is a combination of ALPSM and the approach by Lassing et al. [68] to consider ripple effects by taking expert knowledge into account. A disadvantage is that ALMA does not involve software management activities. Architecture-Centric Project Management (ACPM) [61] takes software architecture as the central artefact for planning and management activities. For architecture-based cost estimation, the architecture is applied to decompose planned software changes into various tasks to realize the changes. For each task the assigned developer is asked to estimate the effort of realizing the change. KAMP [69] goes beyond ACPM by using formalized architectural models where ACPM uses only the structural view of an architecture and therefore does not consider management costs, such as expenses for re-deployment. KAMP combines several strengths of existing approaches. It makes explicit use of formal software architecture models, provides guidance and automation by tool support, and considers management as well as development effort. Applications of KAMP for deriving work plans to solve performance issues [70] and for automated software project planning [71] have already been proposed.

Despite existing approaches for investigating changes in the software engineering discipline, e.g. change impact analysis for data [72], object-oriented software [73] and UML diagrams [74], management of changes to aPS designs is still a relatively young research field [75]. In Rieke et al. [76], an approach for addressing change propagation between domain-specific models and a domain-spanning system model by identifying arising inconsistencies within the models is proposed. Göring and Fay [77] provide a meta-model for the domain of instrumentation and control systems and propose methods for analyzing the respective system designs. Nevertheless, appropriate tools for managing changes within interdisciplinary designs of aPS, especially focusing on the control system, are not yet sufficiently available [78], making further, interdisciplinary investigation of changes [79], techniques for managing and predicting changes [80] as well as for estimating maintainability necessary.

IV. OPEN DEMONSTRATORS FOR SOFTWARE CHANGES

To introduce the differences in software evolution in the two domains, two case studies are used to briefly present typical changes, i.e., adding a self-healing mode in the aPS case study and adding new features or changing platform in the IS case study. The evolution scenarios reflect specific perspectives of the two domains. The IS and aPS domain differ in the levels of granularity of evolution scenarios. Consequently, both domains consider different first class elements – abstract functionality in the IS domain and physical/logical parts in the aPS domain. Changes in both case studies affect the model and code level.

A. *Software Changes in IS – The Common Component Modeling Example (CoCoME)*

To explore and investigate the applicability of fundamental research results regarding evolution in the IS domain, CoCoME serves as an open case study. CoCoME resembles a trading system as it may be applied in a supermarket chain handling sales. This includes processing sales at a single cash desk in a store of the chain as well as enterprise-wide administrative tasks like ordering of products that are running out, inventory management, or generating reports. An overview of CoCoME is given in Fig. 1. Basically, CoCoME is organized as a layered software architecture as depicted in Fig. 2, which allows for distributing the system on server nodes and for remote communication. Detailed description is given in [81, 82].

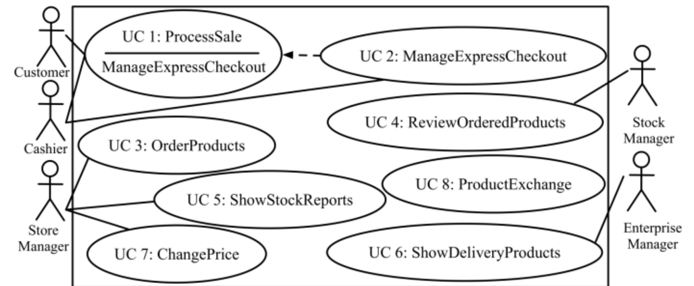


Figure 1. Overview of use cases of the CoCoME trading system [81]

CoCoME evolved from a community case study of component-based software modelling and analysis. The specific characteristics of CoCoME are that it contains on the one hand properties of real-world business IS and on the other hand is so condensed that academic groups can use it within reasonable time. We evolved CoCoME to a study subject [83] on which methods in the context of software evolution are applied. Since CoCoME has been applied and evolved successfully in various research projects, e.g. SLA@SOI [1] and Q-Impress [2], several implementations exist, spanning different platforms and technologies, such as plain Java code, service-oriented frameworks, or hybrid cloud-based architectures [84]. Furthermore, various development artifacts are available, such as requirements specification or design documentation that evolved over time. CoCoME is well suited to serve as a demonstrator of IS evolution, as the supermarket context is generally comprehensible and the study subject has appropriate complexity. As CoCoME represents a distributed IS, several quality properties (e.g., performance, reliability, and security) are affected by evolutionary changes. Consequently, a wide variety of problems regarding evolution can be investigated with CoCoME. Moreover, CoCoME satisfies requirements on a research platform derived from related work in empirical research [83] and provides distinct evolution scenarios covering adaptive and perfective evolution.

A *perfective evolution* to keep the trading system usable in a changing environment is represented by adding a pick-up shop to satisfy emerging customer requirements, as the CoCoME company is in competition with online shop vendors (such as

[1] http://sla-at-soi.eu, retrieved on 7/20/2015.

[2] www.q-impress.eu, retrieved on 7/20/2015.

Amazon). This design-time modification includes adding new use cases, modifying the existing system architecture and implementation by adding new web shop components. *Adaptive evolution* is covered by a platform alteration to reduce operating costs of the resources. Therefore, the CoCoME company migrates some resources to the Cloud. The enterprise server and its connected database are now running in the Cloud, which requires changes in deployment, infrastructure services, and configuration. The introduction of the Cloud enables flexible adaptation and reconfiguration of the system, but it causes new challenges regarding quality aspects which must be considered in development and operation. Furthermore, in order to accommodate the self-adaptiveness of software architectures, *reconfiguration* during system operation is addressed by a database migration scenario. The CoCoME company starts a big advertising campaign. Advertisements lead to an increased amount of sales. Thus, the performance of the system may suffer due to limited capacities of the cloud provider currently hosting the enterprise database. Migrating the database from one cloud provider to another may solve the scalability issues. However, this may cause privacy issues due to violations of privacy constraints [85]. This scenario includes the application of instrumentation and simulation techniques during operation.

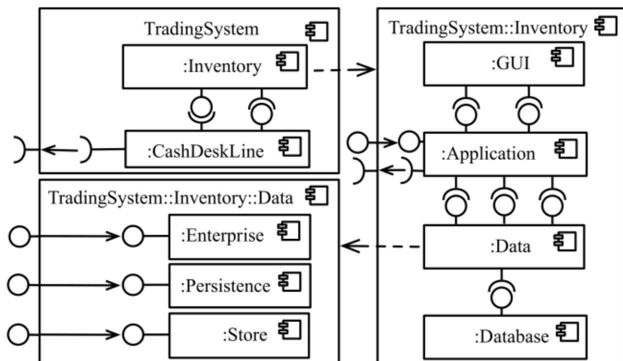


Figure 2. Overview of software architecture of CoCoME trading system

For all the evolution scenarios of CoCoME, both structure and behavior are documented using UML and additionally PCM [25], which allows for using simulation and analysis techniques easily based on software design models. Furthermore, also requirements, design rationales and alternatives, as well as simulation data and data gathered while observing the running system are part of the documentation. The documentation is continuously being developed, improved and made available to the CoCoME community[3].

B. *Software Changes in aPS – Pick and Place Unit (PPU)*

To explore and investigate the applicability of fundamental research results to the challenges in the aPS domain, the Pick and Place Unit (PPU) with its 22 digital inputs, 13 digital outputs and 3 analog output signals provides simple discrete event automation tasks [86] and serves as an open case study [87]. The PPU is limited in size and complexity but provides a trade-off between problem complexity and evaluation effort. Scalability to real-world aPS may have to be investigated in further case studies. To explore and investigate the evolution of aPS, sixteen scenarios (i.e.: variants of the PPU) are defined [46], including both sequential evolution (e.g., evolution steps to enlarge the PPU's functionality and performance) and parallel evolution (e.g., same functionality on different types of platforms, i.e., PLCs). Besides pure software changes that do not incorporate adaptations in the mechanics and automation hardware of the PPU, changes to these disciplines are also considered in the scenarios (e.g., installing novel sensors in the PPU in Sc13 [87] up to replacing the control platform in Sc12d [46]). Up to now, all scenarios are based on a single CPU, which represents a centralized architecture of the automation system (platform).

For the PPU scenarios, both structure and behavior are documented using SysML [87]: structural decompositions are documented as Block Definition Diagrams and the behavior is documented within State Charts. The documentation is available online – both as Eclipse Papyrus[4] models and as textual documentations [87]. In addition, automation software code is available for each scenario implemented in CODESYS[5] (industrial development tool for PLC software). A MATLAB/Simulink simulation for each scenario is available to make testing possible. The documentation of the PPU is constantly being developed, improved and placed at the disposal of the PPU community[6].

In the following, we focus on changes at design time, exemplifying the challenges that arise within the aPS domain when changes are performed at the model and on the code level. Therein, we illustrate the challenges using a conveyor system (Scenario 12 of the PPU), which is intended to push work pieces (WPs) into respective slides. A change scenario is exemplified that incorporates the changes necessary to implement a self-healing mode (SHM), which is able to identify and resolve work piece jams in the conveyor system (Scenario 12f of the PPU).

1) *Changes at the model level and mechanical changes*

In Scenario 12 of the PPU, the software is not able to identify a WP jam precisely (Fig. 3) and therefore triggers *Alarm104* if a product does not reach the slide within the expected time (timer

TABLE I. MECHANICAL CHANGES IN COMPONENT LIST

Machine group	Device number	Device	Function	location	Device/ Signal type	Power supply [V]	Remarks
310	M1	sorting line	sort/push WP in slide	pusher	pneum. valve DO	24V	mand.
310	B1.1	sorting line	pusher is extended	pusher	reed switch, DI	24V	mand.
310	B1.2	sorting line	pusher is retracted	pusher	reed switch, DI	24V	mand.
310	B1.3	sorting line	WP detected in slide	slide	opt. sensor, DI	24V	mand.
310	B1.4	sorting line	position of pusher	pusher	distance sensor, AI	0-24V	SHM
…	…	…	…	…	…	…	…
300	B1	sorting line	pressure sensor / detect leakage for all pushers	valve node	pressure sensor, AI	0-24V	SHM

[3] http://www.dfg-spp1593.de/cocome, retrieved on 7/20/2015.

[4] http://eclipse.org/papyrus/, retrieved on 7/20/2015.

[5] http://www.codesys.com, retrieved on 7/20/2015.

[6] http://www.ppu-demonstrator.org, retrieved on 7/20/2015.

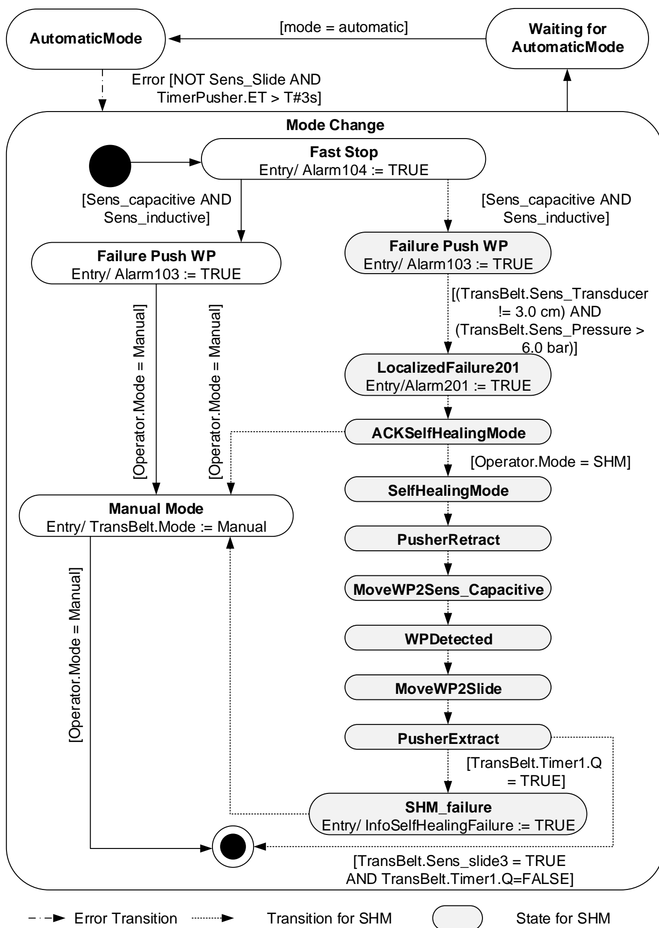


Figure 3. Change from manual mode (Sc12) to Self-Healing Mode (SHM) as a State Chart at the model level (Sc12f) according to [46]

t > 3 seconds). Note that this is a drastically simplified failure model that serves the purpose of illustrating the changes that need to be performed to the PPU. The additionally installed sensors in Scenario 12f provide the necessary information to detect a WP jam more precisely. Therefore, additional sensors (mechanical change) need to be implemented to enable a self-healing mode (SHM, see Table I). The component list (Table I) represents these necessary mechanical changes; the additional analog distance sensor *310 B1.4* as well as the additional pressure sensor *320 B1* are shown in Fig. 4. This failure can be exactly determined when (1) the distance sensor contained within the respective pusher indicates that it is not extended completely (*Sens_Transducer, 310 B1.4*), (2) the pressure of the pneumatic system is not as expected (*Sens_Pressure, 300 B1*) and (3) no WP entered the slide (*Sens_Slide, 320 B1.3*). The self-healing software functionality is realized as an additional functionality to the automatic operation mode. After detecting the fault, a sequence of operations for fault recovery, i.e., the removal of the WP jam and the subsequent correct sorting of the WP, is executed (cp. Fig. 3).

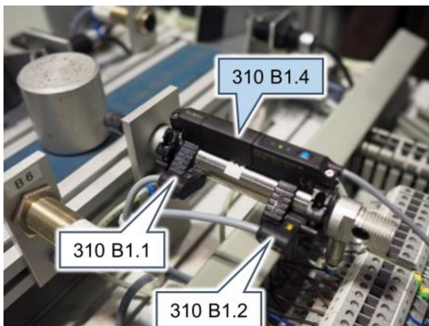
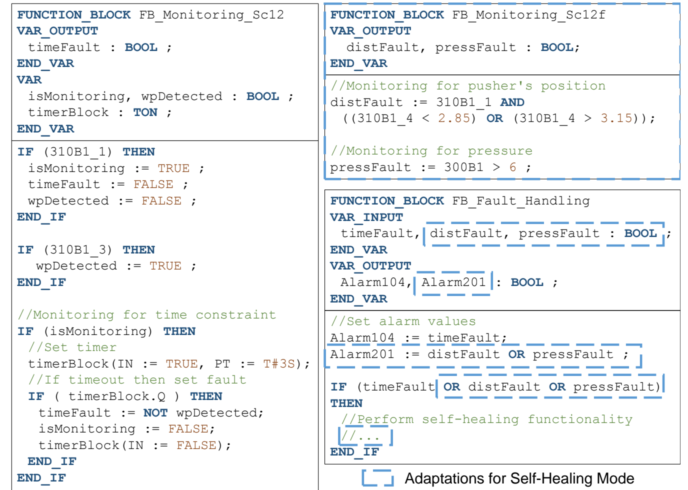


Figure 4. Additional analog sensor (310 B1.4) in Self-Healing Mode (SHM)

2) *Changes at the code level*

Aside from the changes at the model level, code-level changes need to be performed to enable the intended self-healing mode. Besides others, self-healing mode (Scenario Sc12f, Fig. 3) requires the identification of faults, e.g., violation of the time constraint for a WP to arrive at the slide (*Alarm104* in Fig. 5) and faulty pressure or distance (*Alarm201* in Fig. 5).

The fault handling requires evolution of the initial FBs as discussed in the following: *FB_Monitoring_Sc12* covers the monitoring of the time constraint (*TimerPusher* in Fig. 3) for a WP to be detected by the digital sensor input (*Sens_Slide* in Fig. 3). This is represented by the function block *FB_Monitoring_Sc12* in Fig. 5. The added function block *FB_Monitoring_Sc12f* handles the newly added analog sensors (*Sens_Transducer* and *Sens_Pressure* in Fig. 3). Moreover, function block *FB_Fault_Handling* needs to be adapted accordingly to trigger *Alarm201* and to solve the WP jam based on the identified faults. The alarm signals are forwarded to the handling of modes of operation and to the technical process.



```
FUNCTION_BLOCK FB_Monitoring_Sc12
VAR_OUTPUT
 timeFault : BOOL ;
END_VAR
VAR
 isMonitoring, wpDetected : BOOL ;
 timerBlock : TON ;
END_VAR

IF (310B1_1) THEN
 isMonitoring := TRUE ;
 timeFault := FALSE ;
 wpDetected := FALSE ;
END_IF

IF (310B1_3) THEN
  wpDetected := TRUE ;
END_IF

//Monitoring for time constraint
IF (isMonitoring) THEN
 //Set timer
 timerBlock(IN := TRUE, PT := T#3S);
 //If timeout then set fault
 IF ( timerBlock.Q ) THEN
  timeFault := NOT wpDetected;
  isMonitoring := FALSE;
  timerBlock(IN := FALSE);
 END_IF
END_IF
```

```
FUNCTION_BLOCK FB_Monitoring_Sc12f
VAR_OUTPUT
  distFault, pressFault : BOOL;
END_VAR

//Monitoring for pusher's position
distFault := 310B1_1 AND
  ((310B1_4 < 2.85) OR (310B1_4 > 3.15));

//Monitoring for pressure
pressFault := 300B1 > 6 ;
```

```
FUNCTION_BLOCK FB_Fault_Handling
VAR_INPUT
  timeFault, distFault, pressFault : BOOL;
END_VAR
VAR_OUTPUT
  Alarm104, Alarm201 : BOOL ;
END_VAR

//Set alarm values
Alarm104 := timeFault;
Alarm201 := distFault OR pressFault ;

IF (timeFault OR distFault OR pressFault)
THEN
  //Perform self-healing functionality
  //...
END_IF
```

Adaptations for Self-Healing Mode

Figure 5. Maintenance tasks on (structural) code level (Structured Text, IEC 61131-3) (blue – changes for scenario Sc12f)

V. CONCLUSIONS AND OUTLOOK

To enable a domain-spanning maintainability estimation of IS and aPS, the approaches existent in the domain of IS must be adapted and applied to the aPS domain. Therefore, a deeper investigation of these techniques for aPS is necessary.

Existing approaches to maintainability and change effort estimation often do not have sufficient architecture descriptions or are limited to software development without taking management tasks into account. Work on task-based project planning uses the architecture only in a very coarse-grained

manner, if at all. Also approaches to architecture-based project planning and architecture-based software evolution neither offer automated change impact analysis nor support derivation of change activities or change effort estimation based on a given architecture. Scenario-based architecture analysis approaches that focus on software development activities and artefacts, lack a formalized architecture model and seldom come along with tool support. Appropriate tools for managing changes within designs of aPS, especially focusing on the control system, are not yet sufficiently available. We target developing a procedure to analyze maintainability of software-intensive systems by deriving maintenance task lists from architecture descriptions. The lists are applied to make estimations about maintainability and to compare design alternatives. This procedure is expected to consider the interrelations between processes and the system as well as their effects on maintainability estimation while regarding all system life-phases.

ACKNOWLEDGMENT

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution.

REFERENCES

[1] ZVEI Automation Division, "Life-Cycle-Management for Automation Products and Systems," 2012. [Online]. Available: http://www.zvei.org/-Publikationen/Guideline_LifeCycle_en.pdf

[2] H.-P. Wiendahl, *et al.*, "Changeable Manufacturing – Classification, Design and Operation," *CIRP Ann. – Manuf. Tech.*, vol. 56, no. 2, pp. 783–809, 2007.

[3] *Software Engineering – Product Quality – Part 1: Quality Model*, IEC Std. ISO/IEC 9126-1, 2001.

[4] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std., 1990.

[5] A. Avizienis, *et al.*, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[6] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. Academic Press Professional, 1985.

[7] N. H. Madhavji, J. Fernandez-Ramil, and D. Perry, *Software Evolution and Feedback: Theory and Practice*. John Wiley & Sons, 2006.

[8] A. M. Aerts, *et al.*, "Architectures in Context: On the Evolution of Business, Application Software, and ICT Platform Architectures," *Informat. & Manage.*, vol. 41, no. 6, pp. 781–794, 2004.

[9] R. Heinrich, *et al.*, "Integrating Business Process Simulation and Information System Simulation for Performance Prediction," *Softw. Syst. Modeling*, 2015. DOI: 10.1007/s10270-015-0457-1.

[10] J. A. O'Brien and G. Marakas, *Introduction to Information Systems*. McGraw-Hill, 2010.

[11] R. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability Prediction for Component-based Software Architectures," *J. Syst. Software*, vol. 66, no. 3, pp. 241–252, 2003.

[12] B. Vogel-Heuser, *et al.*, "Challenges for Software Engineering in Automation," *J. Softw. Eng. Applicat.*, vol. 7, no. 5, pp. 440–451, 2014.

[13] *Programmable Logic Controllers – Part 3: Programming Languages*, Std. IEC 61131-3, 2013.

[14] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.*, vol. 11, no. 3, pp. 259–266, 1984.

[15] C. R. Maga, N. Jazdi, and P. Göhner, "Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity," in *IFAC World Congress*, 2011, pp. 9145–9150.

[16] C. Szyperski, *Component Software: Beyond Object-oriented Programming*. Addison Wesley, 1998.

[17] C. R. Maga, N. Jazdi, and P. Göhner, "Requirements on Engineering Tools for Increasing Reuse in Industrial Automation," in *IEEE Int. Conf. Emerging Tech. Fact. Autom.*, 2011.

[18] U. Katzke, K. Fischer, and B. Vogel-Heuser, "Development and Evaluation of a Model for Modular Automation in Plant Manufacturing," in *Int. Conf. Informat. Syst. Anal. Synth.*, 2004.

[19] T. Kehrer, *et al.*, "Understanding Model Evolution Through Semantically Lifting Model Differences with SiLift," in *IEEE Int. Conf. on Software Maintenance*, 2012.

[20] R. El Hadj Khalaf, B. Agard, and B. Penz, "Module Selection and Supply Chain Optimization for Customized Product Families Using Redundancy and Standardization," *IEEE Trans. Autom. Sci. Eng.*, vol. 8, no. 1, pp. 118–129, 2011.

[21] K. Thramboulidis, "The 3+1 SysML View-Model in Model Integrated Mechatronics," *J. Softw. Eng. Applicat.*, vol. 3, no. 2, pp. 109–118, 2010.

[22] S. Feldmann, J. Fuchs, and B. Vogel-Heuser, "Modularity, Variant and Version Management in Plant Automation – Future Challenges and State of the Art," in *Int. Design Conf.*, 2012, pp. 1689–1698.

[23] K.-K. Lau and Z. Wang, "A Taxonomy of Software Component Models," in *EUROMICRO Conf. on Software Engineering and Advanced Applications*, 2005, pp. 88–95.

[24] N. Medvidovic and R. N. Tayklor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[25] S. Becker, *et al.*, "The Palladio Component Model for Model-driven Performance Prediction," *J. Syst. Software*, vol. 82, no. 1, pp. 3–22, 2009.

[26] B. Vogel-Heuser, *et al.*, "Model-driven Engineering of Manufacturing Automation Software Projects – A SysML-based Approach," *Mech.*, vol. 24, no. 7, pp. 883–897, 2014.

[27] P. Leitão and F. Restivo, "ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control," *Comput. Ind.*, vol. 57, no. 2, pp. 121–130, 2006.

[28] M. Bonfè, C. Fantuzzi, and C. Secchi, "Design patterns for model-based automation software design and implementation," *Contr. Eng. Practice*, vol. 21, no. 11, pp. 1608–1619, 2013.

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

[30] *Function Blocks – Part 1: Architecture*, IEC Std. IEC 61499-1, 2011.

[31] K. Thramboulidis, "Overcoming Mechatronic Design Challenges: the 3+1 SysML-view Model," *J. Comput. Sci. Tech.*, vol. 1, no. 1, pp. 6–14, 2013.

[32] OMG, "Unified Modeling Language (UML) V2.4.1," 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[33] OMG, "Systems Modeling Language (SysML) V1.3," 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[34] L. Bassi, *et al.*, "A SysML-Based Methodology for Manufacturing Machinery Modeling and Design," *IEEE/ASME Trans. Mech.*, vol. 16, no. 6, pp. 1049–1062, 2011.

[35] D. Witsch and B. Vogel-Heuser, "PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking," in *IFAC World Congr.*, 2011.

[36] M. Obermeier, *et al.*, "Fundamental Aspects Concerning the Usability Evaluation of Model-Driven Object Oriented Programming Approaches in Machine and Plant Automation," in *Int. Conf. on Human-Computer Interaction*, 2011.

[37] S. She, *et al.*, "Reverse Engineering Feature Models," in *Int. Conf. on Software Eng.*, 2011.

[38] T. Thüm, *et al.*, "FeatureIDE: An Extensible Framework for Feature-oriented Software Development," *Sci. Comput. Programming*, vol. 79, pp. 70–85, 2014.

[39] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag, 2005.

[40] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Int. Conf. on Software Product Lines*, 2005, pp. 26–29.

[41] S. Apel, *et al.*, “Detection of Feature Interactions Using Feature-aware Verification,” in *IEEE/ACM Int. Conf. Autom. Software Eng.*, 2011, pp. 372–375.

[42] D. Dhungana, P. Grünbacher, and R. Rabiser, “The DOPLER Meta-tool for Decision-oriented Variability Modeling: A Multiple Case Study,” *Autom. Softw. Eng.*, vol. 18, no. 1, pp. 77–114, 2011.

[43] D. Dhungana, *et al.*, “Automated Verification of Interactive Rule-based Configuration Systems,” in *IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2013.

[44] J. Guo, *et al.*, “Variability-aware Performance Prediction: A Statistical Learning Approach,” in *IEEE/ACM Int. Conf. Autom. Software Eng.*, 2013.

[45] R. Pohl, V. Stricker, and K. Pohl, “Measuring the Structural Complexity of Feature Models,” in *IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2013.

[46] B. Vogel-Heuser, *et al.*, “Challenges of Parallel Evolution in Production Automation Focusing on Requirements Specification and Fault Handling,” *at – Automatisierungstechnik*, vol. 62, no. 11, pp. 758–770, 2014.

[47] G. Dauenhauer, T. Aschauer, and W. Pree, “Variability in Automation System Models,” in *Int. Conf. on Software Reuse*, 2009, pp. 116–125.

[48] S. Feldmann, C. Legat, and B. Vogel-Heuser, “Engineering support in the machine and plant manufacturing domain through interdisciplinary product lines: An applicability analysis,” in *IFAC Symposium on Information Control in Manufacturing*, 2015.

[49] C. R. Maga and N. Jazdi, “An Approach for Modeling Variants of Industrial Automation Systems,” in *IEEE Int. Conf. on Automation, Quality and Testing, Robotics*, 2010.

[50] R. Froschauer, D. Dhungana, and P. Grünbacher, “Managing the Life-cycle of Industrial Automation Systems with Product Line Variability Models,” in *EUROMICRO Conf. Softw. Eng. Adv. Applicat.*, 2008.

[51] A. Polzer, S. Kowalewski, and G. Botterweck, “Applying Software Product Line Techniques in Model-based Embedded Systems Engineering,” in *ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009, pp. 2–10.

[52] N. Papakonstantinou and S. Sierla, “Generating an Object Oriented IEC 61131-3 software product line architecture from SysML,” in *IEEE Int. Conf. Emerging Tech. Fact. Autom.*, 2013.

[53] U. Ryssel, J. Ploennigs, and K. Kabitzsch, “Reasoning of Feature Models from Derived Features,” in *Int. Conf. on Generative Programming and Component Engineering*. ACM Press, 2012, p. 21.

[54] S. Feldmann, C. Legat, and B. Vogel-Heuser, “An Analysis of Challenges and State of the Art for Modular Engineering in the Machine and Plant Manufacturing Domain,” in *IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control*, 2015.

[55] B. Kirwan and L. K. Ainsworth, *A Guide To Task Analysis*. London: Taylor & Francis, 1992.

[56] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Taylor & Francis, 1983.

[57] S. K. Card, T. P. Moran, and A. Newell, “The Keystroke-level Model for User Performance Time with Interactive Systems,” *Commun. ACM*, vol. 23, no. 7, pp. 396–410, 1980.

[58] J. B. Dreger, *Function Point Analysis*. Prent, 1989.

[59] B. Boehm and E. Harrowitz, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[60] M. Conway, “How do committees invent?” *Datamination*, vol. 14, pp. 28–31, 1968.

[61] D. J. Paulish and L. Bass, *Architecture-Centric Software Project Management: A Practical Guide.* Addison Wesley, 2001.

[62] R. Carbon, *et al.*, *Architecture-Centric Software Producibility Analysis*. Fraunhofer IRB, 2012.

[63] D. Garlan, *et al.*, “Evolution Styles: Foundations and Tool Support for Software Architecture Evolution,” in *IEEE/IFIP Conf. on Software Architecture and European Conf. on Software Architecture*, 2009, pp. 131–140.

[64] M. Naab, “Enhancing Architecture Design Methods for Improved Flexibility in Long-living Information Systems,” Ph.D. dissertation, Fraunhofer IESE, 2012.

[65] P. C. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley, 2001.

[66] P. Bengtsson and J. Bosch, “Architecture level prediction of software maintenance,” in *European Conf. Softw. Maintenance Reeng.*, 1999.

[67] P. Bengtsson, *et al.*, “Architecture-level Modifiability Analysis (ALMA),” *J. Syst. Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[68] N. Lassing, D. Rijsenbrij, and H. van Vliet, “Towards a Broader View on Software Architecture Analysis of Flexibility,” in *Asia Pacific Software Engineering Conf.*, 1999, pp. 238–245.

[69] K. Rostami, *et al.*, “Architecture-based Assessment and Planning of Change Requests,” in *Int. Conf. on the Quality of Software Architectures*, 2015.

[70] C. Heger and R. Heinrich, “Deriving Work Plans for Solving Performance and Scalability Problems,” in *Computer Performance Engineering, LNCS 8721*, 2014, pp. 104–118.

[71] O. Hummel and R. Heinrich, “Towards Automated Software Project Planning – Extending Palladio for the Simulation of Software Processes,” in *Symposium on Software Performance*, 2013, pp. 20–29.

[72] S. S. Chawathe, *et al.*, “Change Detection in Hierarchically Structured Information,” in *ACM SIGMOD Int. Conf. on Management of Data*, 1996.

[73] D. Kung, *et al.*, “Change Impact Identification in Object Oriented Software Maintenance,” in *IEEE Int. Conf. on Software Maintenance*, 1994, pp. 202–211.

[74] L. Briand, Y. Labiche, and L. O’Sullivan, “Impact analysis and change management of UML models,” in *IEEE Int. Conf. on Software Maintenance*, 2003, pp. 256–265.

[75] B. Hamraz, N. H. M. Caldwell, and P. J. Clarkson, “A Holistic Categorization Framework for Literature on Engineering Change Management,” *Syst. Eng.*, vol. 16, no. 4, pp. 473–505, 2013.

[76] J. Rieke, *et al.*, “Management of Cross-domain Model Consistency for Behavioral Models of Mechatronic Systems,” in *Int. Design Conf.*, 2012.

[77] M. Göring and A. Fay, “Method for the Analysis of Temporal Change of Physical Structure in the Instrumentation and Control Life-cycle,” *Nuclear Eng. Tech.*, vol. 45, no. 5, pp. 653–664, 2013.

[78] G. Huang and K. Mak, “Current Practices of Engineering Change Management in UK Manufacturing Industries,” *Int. J. Operations Prod. Manage.*, vol. 19, no. 1, pp. 21–37, 1999.

[79] G. Huang, W. Yee, and K. Mak, “Current Practice of Engineering Change Management in Hong Kong Manufacturing Industries,” *J. Materials Processing Tech.*, vol. 139, no. 1-3, pp. 481–487, 2003.

[80] T. A. W. Jarratt, C. Eckert, N. H. M. Caldwell, and P. J. Clarkson, “Engineering Change: An Overview and Perspective on the Literature,” *Res. Eng. Des.*, vol. 22, no. 2, pp. 103–124, 2010.

[81] S. Herold, *et al.*, “CoCoME – The Common Component Modeling Example,” in *The Common Component Modeling Example*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 16–53.

[82] R. Heinrich, *et al.*, “Architecture-based Analysis of Changes in Information System Evolution,” in *Workshop Software Re-engineering and Evolution (WSRE)*, 2015.

[83] R. Heinrich, *et al.*, “A Platform for Empirical Research on Information System Evolution,” in *International Conference on Software Engineering and Knowledge Engineering*, 2015.

[84] R. Heinrich, *et al.*, “Run-time architecture models for dynamic adaptation and evolution of cloud applications,” Tech. Rep. TR_1503, 2015. [Online]. Available: http://www.uni-kiel.de/journals/receive/-jportal_jparticle_00000265

[85] R. Heinrich, *et al.*, “Integrating run-time observations and design component models for cloud system analysis,” in *Int. Workshop on Models@run.time,* CEUR Vol-1270, 2014.

[86] B. Vogel-Heuser, “Usability Experiments to Evaluate UML/SysML-based Model Driven Software Engineering Notations for Logic Control in Manufacturing Automation,” *J. Softw. Eng. Applicat.*, vol. 7, no. 11, pp. 943–973, 2014.

[87] B. Vogel-Heuser, *et al.*, “Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit,” Tech. Rep. TUM-AIS-TR-01-14-02, 2014. [Online]. Available: https://-mediatum.ub.tum.de/node?id=1208973