Chapter 10
Learning from Evolution for Evolution

Stefan Kögel, Matthias Tichy, Abhishek Chakraborty, Alexander Fay, Birgit Vogel-Heuser, Christopher Haubeck, Gabriele Taentzer, Timo Kehrer, Jan Ladiges, Lars Grunske, Mattias Ulbrich, Safa Bougouffa, Sinem Getir, Suhyun Cha, Udo Kelter, Winfried Lamersdorf, Kiana Busch, Robert Heinrich, and Sandro Koch

S. Kögel · M. Tichy (✉)
Institut für Softwaretechnik und Programmiersprachen, Universität Ulm, Ulm, Germany
e-mail: stefan.koegel@uni-ulm.de; matthias.tichy@uni-ulm.de

A. Chakraborty · A. Fay · J. Ladiges
Helmut-Schmidt-Universität, Fakultät für Maschinenbau, Professur für Automatisierungstechnik, Hamburg, Germany
e-mail: chakraba@hsu-hh.de; alexander.fay@hsu-hh.de; jan.ladiges@hsu-hh.de

B. Vogel-Heuser · S. Bougouffa · S. Cha
Technische Universität München, Lehrstuhl für Automatisierung und Informationssysteme, Garching, Germany
e-mail: vogel-heuser@tum.de; safa.bougouffa@tum.de; suhyun.cha@tum.de

C. Haubeck · W. Lamersdorf
Universität Hamburg, MIN-Fakultät, Fachbereich Informatik, Hamburg, Germany
e-mail: haubeck@informatik.uni-hamburg.de; lamersd@informatik.uni-hamburg.de

G. Taentzer
Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Marburg, Germany
e-mail: taentzer@informatik.uni-marburg.de

T. Kehrer
Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany
e-mail: timo.kehrer@informatik.hu-berlin.de

L. Grunske · S. Getir
Institut für Informatik, Humboldt-Universität zu Berlin, Johann-von-Neumann-Haus, Berlin, Germany
e-mail: grunske@informatik.hu-berlin.de; getir@informatik.hu-berlin.de

M. Ulbrich
Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: ulbrich@kit.edu

U. Kelter
Praktische Informatik/Softwaretechnik, Fachbereich Elektrotechnik und Informatik, Universität - GH - Siegen, Siegen, Germany
e-mail: kelter@informatik.uni-siegen.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_10

Missing knowledge about the system is often one of the root causes of failed software evolution in practice. For example, the well-known failed maiden launch of the Ariane 5 rocket [Dow97] can be attributed partly to missing knowledge about the behaviour of a software system reused from the Ariane 4 rocket. The old software was integrated into the Ariane 5 rocket, which, however, had a different flight trajectory compared to the Ariane 4 rocket. That integration problem led to a value conversion error eventually causing the self-destruction of the rocket.

This very costly error serves as an illustrative example of the effects of missing knowledge during the evolution of systems. One of the focus areas of the priority program aims at providing "Knowledge Carrying Software", that is avoiding missing knowledge in the first place. The other focus areas, "Methods and Processes" and "Platforms and Environments for Evolution", enable knowledge-carrying software.

For successful evolution, knowledge not only about a piece of software but also about its environment, hardware, network, other software, libraries, and ecosystem is needed. Furthermore, users are an important part of the environment, and thus knowledge about the number of users, their different roles as stakeholders in the software, and their behaviour is equally important. For the software itself, knowledge about its structure, that is architecture and design, and its behaviour, is required.

Based on the joint automation case study, the Pick and Place Unit (PPU) (see Chap. 4), several projects in the priority program address the process of acquiring such missing knowledge, that is they support the learning of missing knowledge as a prerequisite for successful software evolution.

Those projects provide different approaches to learning. One group of approaches takes a look at *past* evolutions of the software. This enables, for example, the identification of typical evolution steps or the understanding of how the evolution of one part of the software triggers changes in another software part.

Another group of approaches addresses the *present* state of the software and the impact of software evolution. Particularly, those approaches enable understanding and assessing how a planned software evolution affects the satisfaction of requirements. A complementary aspect is to learn about the behaviour of the system, including its environment, to ensure that non-functional requirements are satisfied.

The final group of approaches addresses *future* evolutions. For example, future evolutions can be semi-automatically predicted based on the learned knowledge, for example, about past evolutions done by engineers on the same system or similar systems.

In summary, the approaches support engineers to understand the *past* evolution of a system, assess *present* evolution scenarios, as well as recommend *future* evolution scenarios. Hence, they enable *learning from evolutions for evolutions*.

K. Busch · R. Heinrich · S. Koch
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: kiana.busch@kit.edu; robert.heinrich@kit.edu; sandro.koch@kit.edu

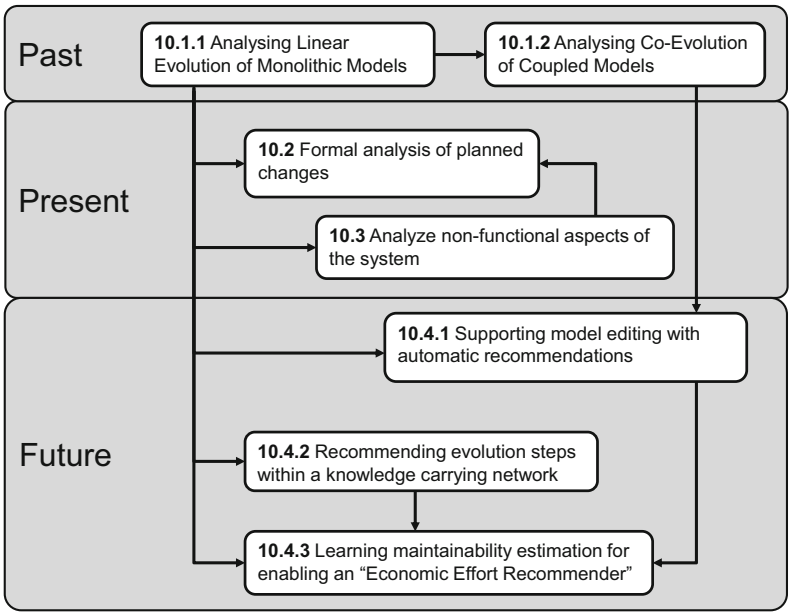


Fig. 10.1 Relations between the subchapters

Figure 10.1 shows these approaches and their relations. Section 10.1 covers the approaches analysing *past evolutions*, Sects. 10.2 and 10.3 cover *present evolutions*, and Sect. 10.4 covers *future evolutions*.

Past The foundation of the other presented approaches is the detailed analysis of historical changes. The approach of Sect. 10.1.1 analyses two versions of models to derive partially ordered sets of detailed changes reflecting the editing semantics of a particular modelling domain. The approach presented in Sect. 10.1.2 draws on this work to analyse the co-evolution of multiple models. Particularly, it supports the computation of co-evolution metrics, which show how much changes in one model result in changes in other models.

Present Two categories of approaches support the evolution of the current system. In Sect. 10.2, functional properties of the current system for small changes are verified using formal verification approaches. Section 10.3 contains two approaches to analyse the non-functional behaviour of systems. On the one hand, timed Petri Nets are learned from behavioural traces of the running system in order to analyse performance and flexibility. On the other hand, Markov chains are learned based on the running system under an evolving environment to continuously check the satisfaction of reliability requirements.

Future Finally, three approaches for assessing and recommending changes for future system evolutions are covered by Sect. 10.4. The first approach in Sect. 10.4.1 supports recommending model evolutions based on the approach for analysing past

evolutions presented in Sect. 10.1. Similarly, evolution steps of one system can be applied in future evolutions to other similar systems using the approach presented in Sect. 10.4.2. Third, economic aspects in the recommendations of evolutions are covered by the approach discussed in Sect. 10.4.3.

10.1 Detailed Analysis of Past Evolutions of Models

Model-based software engineering is the main focus of the SPP projects analysing past evolutions. Model-based software engineering has become a widespread approach for developing software in many application domains, for example for embedded systems in the automotive domain, and is one of the cornerstones to effectively manage the evolution of long-living software systems. In this section, we present techniques that have been developed in the SPP as a fundamental basis for analysing evolutions of model-based systems. In contrast to previous approaches, our achievements particularly enable analyses exploiting fine-grained yet precise and meaningful information about model changes. We consider two different kinds of model evolution. First, in Sect. 10.1.1, we consider the *linear evolution of monolithic models*, that is the chronological evolution of models that are treated as self-contained development documents. As a second kind of evolution, we address the *co-evolution of coupled models* in Sect. 10.1.2, that is the parallel evolution of models, which represent different views or aspects of a system but that are logically and/or physically interrelated. Related work is considered in Sect. 10.1.3 before we conclude in Sect. 10.1.4, along with some pointers for further reading in Sect. 10.1.5.

10.1.1 Analysing Linear Evolution of Monolithic Models

Precise and meaningful descriptions of changes between revisions of models of long-living software systems are of utmost importance in understanding and analysing the evolution of a model-based system and can be considered as a basic form of knowledge about a software (see Chap. 1). However, during model evolution, model modifications are often conducted without proper documentation, for example by recording somehow the model changes that are applied. Even if so, the recorded changes are often of minor quality, or they get lost or unusable when models are exchanged across tool boundaries in a model-based development tool chain [Kü08, Ruh+14b, Keh15]. Moreover, revisions of a model may not be created by manual editing at all but, for example, by reverse engineering from other implementation artefacts [Keh+13a] or by observing and learning from the behaviour of an actual running system [Pie+18] (see also Sect. 10.3). Thus, descriptions of model changes often can only be reconstructed by comparing the different versions of a model with each other, that is by using model differencing techniques. Thus, the calculation of a *difference* (also referred to as *delta* in Chap. 2) between two models is one of the most basic operations for supporting

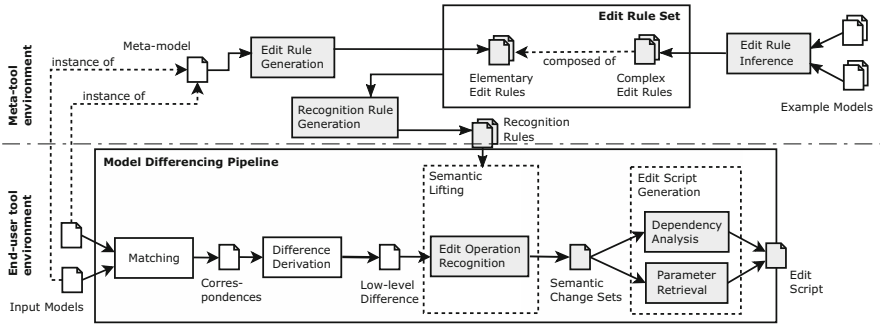


Fig. 10.2 Overview of the MOCA model differencing pipeline and related meta-tool chains

various kinds of model evolution analyses. In this section, we present the model differencing techniques that have been developed in the modular content archives (MOCA) project and that are implemented in the model differencing framework SiLift [Keh+12b], a set of Eclipse plug-ins realised on top of the widely used Eclipse Modeling Framework (EMF) and publicly available from the SiLift website.[1] In the remainder of this section, we first motivate our main technical research goals before we present an overview of our approach for achieving these goals. Finally, we give a selected set of example applications using these techniques for various kinds of evolution analyses and other development tasks in the context of model version and variant management.

Motivation and Goals

Traditional differencing techniques work on textual representations of documents and present document changes in terms of additions and deletions of lines of text. While this produces satisfactory results for source code and other kinds of textual documents, it is commonly agreed that comparing textual representations of models does not produce usable results [BE08, Ema12] and that models should be compared on the basis of graph-based representations. To that end, the internal structure of a model is typically considered as a typed, attributed, partly ordered graph, which is known as the *abstract syntax graph* (ASG) of this model. A meta-model such as, for example, the Unified Modeling Language (UML) meta-model, defines the allowed node and edge types, as well as additional well-formedness rules.

The usual processing pipeline employed by standard model differencing tools consists of the first two steps, depicted in the lower part of Fig. 10.2 (white coloured), referred to as matching and difference derivation: Initially, given two input models that are to be compared, a matching procedure [Kol+09] searches for pairs of corresponding ASG elements, which are considered the same in

[1] http://pi.informatik.uni-siegen.de/projects/SiLift.

both models. Subsequently, a difference is derived as follows: ASG elements not involved in a correspondence are considered to be deleted or created. However, describing model changes based on such primitive graph operations leads to *low-level differences*, which are hard to understand for tool users who are not familiar with the ASG-based representation of models and the related types of nodes and edges defined by a meta-model. Such "meta-model-based difference reports" are not intuitive, are confusing and Are of minor quality for many kinds of model evolution analyses [Alt+09, KKT12].

The main goal of the MOCA project is to lift model differencing techniques to the level of *edit operations* that tool users are familiar with and that capture the true nature of model changes. Elementary operations are the smallest edit operations from a user's point of view, that is they cannot be split into smaller operations, being applicable to a model in a meaningful way. In principle, one can construct arbitrarily many complex edit operations from elementary ones. An important criterion is that a complex edit operation is easier to understand than the single contained elementary edit operations. This includes complex edit operations such as model refactorings and other kinds of evolutionary edit operations that increase the understandability of model changes and that incorporate language-specific editing semantics. Concrete examples may be found later in this chapter in Sect. 10.4. Besides increasing the quality of model differences, the calculation of such high-level differences should scale up to real-world models comprising several thousands of model elements. Finally, since meaningful edit operations are highly specific for a given modelling language, the developed techniques shall be adaptable to domain-specific modelling languages with moderate effort.

Overview of the Approach

An overview of the model differencing approach and techniques developed in the MOCA project is illustrated in Fig. 10.2. As shown in the bottom part of the figure, the differencing of models takes place in several steps. The first two steps, referred to as *matching* and *difference derivation* (white-coloured boxes), constitute the usual model differencing pipeline, as described above. In brief, the matching step identifies the corresponding ASG elements in two models, while the difference derivation step derives a low-level difference in terms of creations and deletions of single ASG elements. The extended differencing pipeline, as developed in the MOCA project, comprises two further steps, which we refer to as *semantic lifting of low-level model differences* and the *generation of edit scripts*, respectively. To be adaptable to a given modelling language, both steps take the edit operations available for this language as configuration input. We use Henshin [Are+10, Str+17], a model transformation language and system based on graph transformation concepts [Ehr+06], in order to specify edit operations (a.k.a. *edit rules*) in a precise, declarative, and rule-based manner. Several meta-tools have been developed in the MOCA project in order to support the development of sets of both elementary and complex edit rules (upper part of Fig. 10.2).

Semantic Lifting of Low-Level Model Differences The goal of this step is to group a potentially large and unstructured set of low-level changes in such a way that model differences are explained in terms of edit operations. To that end, the low-level difference derived from a matching needs to be further processed by a semantic lifting component that identifies sets of low-level changes (called *semantic change sets*) that represent the effect of an edit operation.

In [KKT11], we present a technique for designing tool components that can semantically lift model differences. In our approach, difference information is structurally represented on the level of the ASG, and each edit operation leads to a characteristic change pattern in this difference representation. Thus, finding groups of related low-level changes is basically a pattern matching problem. We use the matching engine of the Henshin interpreter in order to solve this problem. The main task to adapt a semantic lifting component to a given modelling language is to provide a set of so-called *recognition rules*, which find groups of related low-level changes and which annotate these groups accordingly. We automatically derive these recognition rules from their corresponding edit rules.

The recognition rule application algorithm is efficient in the sense that it runs without backtracking. All recognition rules are applied in parallel, which is possible due to the parallel and sequential independence of recognition rules. Since this rule application strategy can lead to too many change sets, that is there can be several alternatives, of which one needs to be chosen eventually, the initial set of semantic change sets must be post-processed in order to obtain a partitioning of the overall set of semantic change sets. This is basically an optimisation problem with respect to some notion of quality of model differences, for example a minimal number of semantic change sets. In [KKT11], we present an efficient heuristic that aims at producing a partitioning comprising a minimal number of change sets. If the set of edit rules used as configuration input is complete in the sense that every possible model difference can be expressed without producing so-called transient effects, it is guaranteed that a partitioning can always be found. Practically, this means that there is an editing sequence in which the effect of every edit operation applied in that sequence is either removed completely by a later operation or is entirely preserved.

Experimental results obtained from different subjects show that the number of editing steps contained in a difference can be drastically compressed by semantic lifting. The compression rates vary depending on the model type and test series. For UML, compression factors of up to 18.0 were measured [KKT11]. Furthermore, results from stress testing our prototypical implementation of the semantic lifting engine show that the approach also scales for models of realistic size.

Generation of Edit Scripts Semantic lifting addresses the question on how to recognise the executions of edit operations in a given low-level difference. However, this technique identifies edit operation executions, also referred to as edit steps, only. This is useful for better understanding changes but not sufficient for some kinds of analyses or for replaying model differences in change propagation scenarios. Actually, two further details are required to generate executable differences, namely the actual *parameters* used as arguments of edit steps, as well as *dependencies*

between edit steps. In [KKT13], we introduce an extended kind of model difference, which we refer to as *edit script*. Technically, an edit script is a complex data structure that contains (1) representations of the detected edit operations, including mappings of the parameters to objects in the low-level difference, and (2) representations of dependencies between these edit steps. Each dependency is annotated with information about its reason, for example one step produces model elements used by a later step. From a conceptual point of view, an edit script is a partially ordered set of edit steps.

Techniques for realising the *parameter retrieval* and *dependency analysis* steps in our differencing pipeline have been presented in [KKT13]. A particular challenge is to provide an efficient implementation of the dependency analysis. In general, two edit steps depend on one another if they can be executed in one order and not in the other order or lead to a different effect if executed in the reverse order, that is they do not commute. Obviously, testing this condition for every pair of edit steps is infeasible if model differences get large. To reduce the set of candidates for dependencies that have to be checked, all pairs of edit rules are statically analysed for potential dependencies using *critical pairs* [Ehr+06], which demonstrate a potential dependency between edit rules in a minimal context. Roughly speaking, a potential dependency between a critical pair of rules is an actual one between the applications of these rules if the minimal rule application of a critical pair can be embedded into the actual model changes [KKT13].

Adaptability of the Tool Environment The adaptability of the tool environment to specific (domain-specific) modelling languages has been addressed by various *meta-tools*, as indicated in the upper part of Fig. 10.2. Although being an exchangeable component in our differencing pipeline, our own model matching engine, which is known as the SiDiff model matcher, is adaptable by a dedicated *language for specifying matching strategies* and algorithms [Keh+12a] (not shown in Fig. 10.2. The configuration of the lifting algorithms is supported by an *edit rule generator*, which generates, for a given meta-model, a complete set of elementary edit operations as Henshin rules [Keh+13b, Keh+16] (see upper part of Fig. 10.2). Following the principle of model transformation by example [Kap+12], more complex edit rules may be deduced from example models using the inference techniques presented in [KAH17]. Finally, Henshin has been extended by an optimised static analysis of potential conflicts and dependencies between transformation rules [Bor+17]. New theoretical results on the critical pair analysis for amalgamated graph transformations [TG15] pave the ground for the analysis of complex edit operations.

Example Applications

The concepts and tools developed in the MOCA project have been applied and evaluated in several collaborations with other research projects. In all these collaborations, configurations of our generic components for specific modelling languages and other kinds of structural documents have been developed. The results of

applying our tools were used for a variety of purposes, notably to solve concrete problems that occurred in other projects of the priority program. In the remainder of this section, we give a selected set of example applications.

In a research collaboration between the projects IMoTEP and MOCA, an approach and supporting tool for reasoning about software product-line evolution using differences on feature models has been developed [Bür+16]. A specific contribution of this collaboration is complex edit operations, whose semantic impact on the set of valid feature configurations can be classified semantically as refactoring, generalisation or specialisation of a feature model. We applied this differencing approach to the evolution of a feature model of the PPU and showed that it is possible to semantically classify feature model differences by using a structural model comparison approach instead of a solver-based solution. More details and further results of this research collaboration can be found in Chap. 7.

In a cooperation between the projects SecVolution and MOCA, a specific component for lifting low-level differences between ontologies has been developed. The general goal was the same as in the cooperation with IMoTEP, namely to get a more abstract, meaningful view on ontology changes by using complex edit operations. This more abstract view enables more adequate planning for changes in other dependent models, such as the security maintenance model [Ruh+14b].

The differencing techniques developed in the MOCA project are used to capture model changes in terms of edit operations within the methodological framework for statistically modelling the evolution of models presented in [Yaz+16]. A main motivation for, and application of, the resulting statistical models is to control the generation of realistic model histories that are intended to be used for evaluating model versioning tools [Yaz+14]. Further usages of the statistical models include various forecasting and simulation tasks. The suitability of the framework is shown by applying it to a large set of design models reverse engineered from real-world Java systems [Keh+13a].

We also evaluated the applicability of the techniques developed in the MOCA project using one of the main case studies of the SPP; the “Pick and Place Unit” (PPU) (see Chap. 4). As a contribution to this case study, we analysed the evolution of the SysML models of the 14 evolution scenarios that have been developed in the first funding period of the SPP. In this study, the developed difference calculation techniques were an effective tool to detect several inconsistencies in the SysML models. Such inconsistencies could be spotted, for example, due to low-level differences between successive evolution scenarios, which could not be entirely lifted to the abstraction level of edit operations. Based on the analysis results, all inconsistencies could be resolved in a collaboration between the projects MoDEMAS and MOCA.

Finally, high-level descriptions of model differences can also assist developers in better understanding the formal specifications of the behavioural differences between two software revisions, for example in the context of the regression verification approach presented in Sect. 10.2.

10.1.2 Analysing Co-evolution of Coupled Models

The development of complex software systems is typically supported by a set of interrelated (a.k.a. coupled) models specifying the system from different viewpoints. In sum, all of these views must reflect a consistent description of the system. Since it is natural that each of these models may change autonomously over time, model-based software evolution is also concerned with model co-evolution. Developers are faced with the problem of consistently co-evolving the different views of the system; that is, the different views must yield a consistent overall description of the system. A basic prerequisite to support developers in achieving such a consistent co-evolution is to understand the co-evolution of coupled models in a particular domain. Getting such an understanding of typical model co-evolutions is the main motivation for analysing the historical co-evolution of coupled models. To that end, an adaptable co-evolution analysis framework has been developed in a joined work by the projects ENSURE and MOCA. The framework draws on the techniques presented in Sect. 10.1.1 and enables statistical analyses of co-evolving models on top of the calculated changes. We present an overview of the analysis capabilities and summarise our results of applying the analysis framework to co-evolving software architecture and quality of service models of the PPU to get a deeper understanding of the nature of consistent co-evolution steps of these kinds of models.

A Generic Framework for Analysing Model Co-evolution

Our co-evolution analysis framework takes a version history of interrelated models, referred to as M_{src} and M_{tgt} in Fig. 10.3a, as input. A pair of successive model versions $i \rightarrow i+1$ from the given history is referred to as evolution step $ev_{i \rightarrow i+1}$, and we assume that the co-evolution history is comprised of consistent evolution steps. We first calculate the model differences $diff(M_i, M_{i+1})$ between successive model versions M_i and M_{i+1} of each evolution step. Thereupon, two kinds of quantitative analyses are supported by our co-evolution analysis framework [GRK14].

For the first analysis, we compute a *correlation* between the kinds of changes of the interrelated models, where the kinds of changes are formally captured by the sets of available edit operations for the source and target model type, respectively. Basically, we count the applied changes for each evolution step in both the source and the target model and compute the Pearson correlation coefficient [LL89] for all combinations of the different kinds of changes to assess the dependencies between the respective edit operations.

The correlation analysis has the advantage of only requiring the source and target models and the respective model changes as input. Thus, this approach can also be applied to study the co-evolution history in cases where no explicit trace links between the observed source and target model exist. However, a correlation between the observable changes does not imply causality.

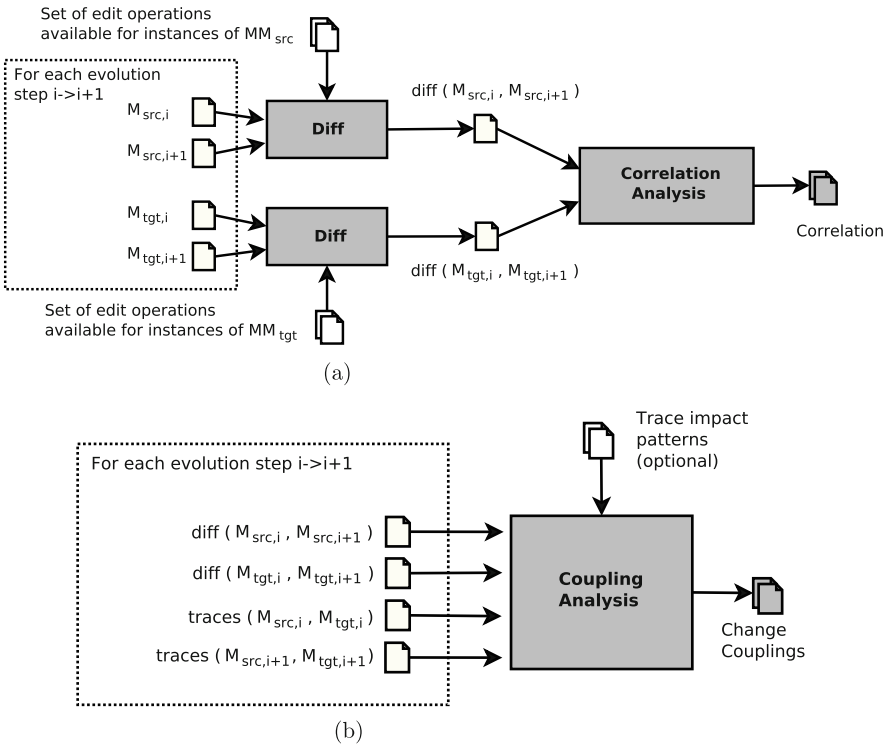


Fig. 10.3 Analysing co-evolutions. (**a**) Correlation analysis. (**b**) Coupling analysis

Hence, we provide a second analysis function referred to as *coupling* analysis, which allows us to identify so-called coupled changes. A coupled change is actually a pair of changes that happened in the same evolution step and where the affected model elements in the interrelated models are connected either directly or indirectly via trace links. In other words, the elements were not just coincidentally changed in the same evolution step. Connectivity of model elements may be formally specified by a graph pattern that relates the model elements of the source and target model, which are connected by trace links. We refer to these graph patterns as trace impact patterns. They are provided as additional domain-specific input parameters to guide the coupling analysis (see Fig. 10.4b).

Case Study on Architecture and Fault Tree Co-evolution

In a first exploratory step of our research on model co-evolution, we worked on understanding the co-evolution between architectural models and quality evaluation models (which we informally define here as models exposing a well-defined semantics amenable to various kinds of reasoning techniques). For that, we manually

developed architectural and fault tree models [Ves+81] of the PPU for every safety-relevant evolution scenario. Based on these models, we first performed a qualitative analysis of the co-evolution that showed that the relation between change operations (e.g. addition/deletion of a component or an event) between fault trees and architectural models is not straightforward and that user interactions are needed for some cases [Get+13].

Figure 10.4a and b show the results of the quantitative analyses on the architectural and fault tree models for the evolution steps of the PPU using our co-evolution analysis framework. For example, there exists a high linear correlation (0.96) for additions of component types and additions of error types. However, looking at the results of the coupling analysis in Fig. 10.3b, we can trace only 17% of the addition of component types to additions of an error type. Instead, in 39% of the cases, the added component type is traced to an existing error type, that is one that was created in a previous scenario.

Without going into the details of the analysis results presented in Fig. 10.4a and b, the most important result is that both the correlation and the coupling analysis confirm our qualitative results [Get+13] for this case study that no simple and straightforward co-evolution of fault tree and architectural models exists that could be automated. However, the analysis results could be exploited in a co-evolution framework supporting model co-evolution as a recommender system (see Sect. 10.1.5).

10.1.3 Related Work

Valuable information can be uncovered by analysing the evolution of software systems and generalising the analysis results, as actively pursued by the Mining Software Repositories research community (see, e.g., [DAm+08]) with the ultimate goal of improving software engineering techniques, methods, and processes. Traditional approaches to software evolution analysis focus on classical code-centric software development where source code files are used as the primary development artefacts [KCM07]. However, software and system models are another kind of primary development artefact that are an integral part of a model-based system. Models typically have several characteristics that are substantially different from those of traditional source code documents, which demands for new methods and techniques in the context of software evolution analysis [BE08, Ema12, Alt+09].

Since the advent of model-driven engineering, model differencing has been addressed by a large number of publications; surveys can be found in [FW07, Sel07]. One class of approaches is based on logging [HK10]; that is, editing processes are logged at the level of user commands or lower levels. Thus, the problem addressed by our model differencing approach disappears. However, logging-based approaches require closed environments and do not work with independently created models; thus, they are not a general solution of the problem. Most state-based approaches have a similar processing structure like the basic differencing pipeline

				Failure Model							Fault Tree1& 2				
				+ErrorType	-ErrorType	+ErrorInstance	-ErrorInstance	+FailureType	+FailureInstance		+BasicEvent	-BasicEvent	+Gate	-Gate	+IntermEvent
		Σ		7	0	53	11	7	11		52	13	17	1	11
			#Scn.	3	0	11	3	4	6		11	3	7	1	6
System Architecture	+CpType	19	9	0,96	-	0,27	-0,25	0,80	0,84		0,22	-0,25	0,43	-0,09	0,84
	-CpType	1	1	-0,13	-	-0,32	0,01	-0,18	-0,23		-0,31	-0,01	-0,27	-0,09	-0,23
	+CpInstance	5	3	0,92	-	0,35	-0,24	0,92	0,85		0,28	-0,24	0,48	-0,15	0,85
	-CpInstance	1	1	0,09	-	-0,13	-0,15	0,45	0,28		-0,12	-0,15	0,11	-0,09	0,28
	+SCpInstance	49	12	0,75	-	0,70	-0,17	0,69	0,77		0,64	-0,16	0,61	-0,25	0,77
	-SCpInstance	11	3	-0,21	-	-0,24	1,00	-0,30	-0,38		-0,16	1,00	-0,44	0,67	-0,38

(a)

				Error Model									Fault Tree1& 2			
				+ErrorType	-ErrorType	=ErrorType	+Error instance	-Error instance	=ErrorInstance	+Failure type	=FailureType	+FailureInstance	+BasicEvent	-BasicEvent	=BasicEvent	+IntermEvent
		Σ		7	0	-	53	11	-	7	-	11	52	13	-	11
			#Scn.	3	0	-	11	3	-	4	-	6	11	3	-	6
System Architecture	+CpType	19	9	0,17	-	0,39	0,06	-	0,00	0,07	0,11	0,18	0,56	0,00	-	0,18
	-CpType	1	1	-	0,00	1,00	-	1,00	0,00	-	0,00	-		1,00	0,00	-
	+CpInstance	5	3	0,11	-	0,00	0,11	-	0,00	0,33	0,00	0,33	0,11	0,00	-	0,33
	-CpInstance	1	1	-	0,00	1,00	-	0,00	1,00	-	0,00	-	-	0,00	1,00	-
	+SCpInstance	49	12	0,11	-	0,68	0,75	-	0,03	0,01	0,03	0,03	0,71	0,03	-	0,03
	-SCpInstance	11	3	-	0,00	1,00	-	1,00	0,00	-	0,00	-	-	1,00	0,00	-

(b)

Fig. 10.4 Analyses results for the PPU case study. (**a**) Correlation between change elements of fault trees and architectural models. (**b**) Coupling analysis between change elements of fault trees and architectural models

shown in Fig. 10.2. They concentrate on the matching step of the differencing calculation [Kol+09], while they have in common the fact that they deliver only low-level model differences. The semantic lifting of model differences, as pursued by the MOCA approach, has been addressed by only a few approaches, however with different goals and assumptions compared to ours. Among them, the one presented by Langer et al. [Lan+13] is the most similar to ours. They use a custom format of edit operation specifications for detecting complex operations in differences that are obtained from EMF Compare. This approach does not intend to produce executable edit scripts that are being used as patches. Consequently, the identification of arguments, dependencies between operation invocations, etc. are not directly addressed.

10.1.4 Conclusion

The specification and calculation of model changes is a cross-cutting concern for supporting the evolution of long-living model-based systems and a fundamental basis for various kinds of evolution analyses. In Sect. 10.1.1, we presented the differencing techniques that have been developed in the MOCA project, along with a summary of selected applications for analysing the linear evolution of monolithic models. Most of the applications have been developed in cooperation with other research projects of the priority program. The presented case studies show that the information that may be extracted from high-level differences generated by our approach are much richer than for conventional model differencing or text-based difference tools. Thus, significant improvements could be achieved for many kinds of model evolution analyses.

In addition to the analysis of the linear evolution of monolithic models, we presented quantitative techniques for the analysis of model co-evolution, that is how changes in one model affect changes in the other model, in Sect. 10.1.2. The analysis techniques have been integrated into a general co-evolution analysis framework, which has been implemented on top of the differencing techniques presented in Sect. 10.1.1. The overall goal of the framework is to assist domain engineers in finding proper co-evolution rules, which can be finally used to support developers in the co-evolution process. The analysis framework has been instantiated to perform a thorough quantitative analysis of co-evolving architectural models and fault trees of the PPU. We show that the models do not co-evolve in a systematic, automatable way, and instead the expertise of the developer is required to achieve consistent co-evolution, confirming the findings of previous research in this context. Nonetheless, the results could be finally exploited to develop a set of model transformation rules supporting model co-evolution through a recommender system for model co-evolution (see Sect. 10.1.5).

10.1.5 Further Reading

In addition to the evolution analyses discussed in Sect. 10.1.1, exact and meaningful specifications of model changes, as provided by the MOCA approach, are an indispensable basis for many further software evolution activities (see Sect. 2.1). In particular, our model differencing approach has been used as a fundamental basis for many techniques supporting model configuration management, an inevitable discipline to manage the evolution of long-living software systems (see Sect. 2.3). Tools and tool functions that have been developed in the MOCA project include, for example, tools for updating local workspace copies [KKR14] and for propagating changes between variants in a product family [KKT14] and a complete environment for developing delta-oriented model-based software product lines [Pie+15, Pie+17].

Recently, we have started to apply our evolution analysis techniques to the field of automated model repair, inspired by the manual inconsistency resolutions as performed in the context of the PPU case study (see example applications in Sect. 10.1.1) The basic assumption is that inconsistencies are introduced by incomplete editing processes, and the idea is to automatically propose the necessary complementing edits to resolve an inconsistency [Tae+17, Ohr+18]. However, further research is needed to localise the origin of an inconsistency in the development history of a model, to develop strategies of how to deal with other conflicting edits that have been performed in the meantime, and to also offer (partial) undo operations as another repair alternative. Another promising line of research is to mine existing model histories for typical model repairs that have been actually performed by developers. The idea of history-based model repair recommendations can be generalised to develop a much broader set of *recommender systems* supporting developers in a variety of model evolution tasks. We will present more general notion of a recommender system for model-driven development later in this chapter in Sect. 10.4.1. The ultimate goal is to speed up modelling by automating repetitive tasks and to warn developers when they make atypical changes to a model. As we will see in Sect. 10.4.1, the methods and techniques developed within the SPP are a suitable basis for further research into this direction.

Concerning the co-evolution of models, understanding the nature of co-evolution in a particular domain is only the first step to reach the overall goal of supporting developers in achieving consistent model co-evolution. This is particularly challenging when co-evolution is not straightforward, as in the case of architectural and quality evaluation models. To that end, we developed the CoWolf tool suite [Get+15b], a generic yet adaptable framework that implements the idea of using a recommender system in order to support model co-evolution, focusing on the co-evolution of architectural and quality evaluation models. CoWolf currently supports seven different types of models: state charts, component diagrams, and sequence diagrams as architectural models, and discrete time Markov chains (DTMC), continuous time Markov chains (CTMC), fault trees, and layered queuing networks (LQN) as quality evaluation models. In fact, the results of a co-evolution analysis may be exploited by CoWolf to propose the most suitable co-evolutions when one of the interrelated models undergoes structural changes; that is, the co-evolution analysis serves as a machine learning approach to improve the effectiveness of the recommendations. Based on our analysis results presented, we tailored a CoWolf instantiation towards architectural models and fault trees, and research results obtained from an experiment based on the PPU case study show that the approach indeed has the potential to significantly reduce the manual effort for consistently evolving the respective kinds of models [Get+18].

10.2 Formal Analysis of Planned Changes

Automated Production Systems are long-living multi-disciplinary systems that are often operated for several decades. Therefore, automated Production Systems (aPS) often faces change requests due to various reasons, for example to fix bugs, to implement additional functionality, or to come up with some technology trends. For this *present* evolution, as introduced in the introduction of this chapter, see Fig. 10.1; analysis of the implementation and evaluation of the evolved system regarding functional and non-functional aspects confirms the outcome of the change. In principle, there are two categories of solutions for ensuring software correctness: testing and formal verification. Testing is widely used for fault detection because of its usability. However, the guarantees that it can provide is naturally limited to the specified test cases, which in most cases cannot cover the entire system behaviour. On the other hand, formal verification can provide proof that a system conforms to its specification in all possible situations. In spite of this nice coverage property, formal methods are not commonly used for quality assurance since they require a high level of expertise to be able to come up with suitable models and specifications.

Within the two SPP1593 projects, MoDEMMiCAS (Model-Driven Evolution Management for Microscopic Changes in Automation Systems) and Improve APS (Regression Verification in a User-Centered Software Development Process for Evolving Automated Production Systems), we want to enable the engineer to incorporate formal techniques into their development process. We will achieve this via two different directions: one is the model-based verification of interdisciplinary aPS combined with the incremental verification of their evolution, and the other is user-centred regression verification for the software.

The development of aPS is a complex task as it brings together at least three different disciplines in one system: mechanical, electrical, and software engineering. To date, each of the involved engineers works independently on a solution, and only then that the results of the engineering of each are finally integrated into one system. However, high-quality designs can be achieved by considering all the disciplines simultaneously [BFB14], and furthermore, shorter development cycle is required for a shortened time for marketing in order to handle ever-changing customer needs [KVD01]. Consequently, the challenges of integrating multi-discipline artefacts and ensuring the correctness of the system arise.

The other characteristic of the aPS is that the software often changes and the scale of the change is usually small. In other words, under the assumption that the previous version of the system confirms previous specification, Regression Verification [Bec+15] overcomes the need for function specification by letting the old version of the software serve as a partial specification of the new revisions where the preserved behaviour is expected. The developer profits from both versions in that they gain additional knowledge about the system.

As a possible situation, a developer from one of the disciplines wants to change the model. Through the development process, the application engineer is informed about how the implemented change on a specific domain model affects the overall

system in view of the quality of the system. Another situation we can regard is that the application engineer revises the code of the automated system. During this code revision, the developer can execute the verification of the preserved and changed functionality and correct the code by using the information from the verification tool chain. Both approaches are based on the characteristic of aPS that changes are usually small variations, and this characteristic facilitates the inclusion of formal verification within the development process. In Sect. 10.2.1, we consider a model-driven evolution management, and Sect. 10.2.2 considers a user-centred software development using regression verification (see Chap. 11 for the detailed techniques of the formal verification used in the approaches)

10.2.1 Using Model-Based Verification of Interdisciplinary Models

In almost every engineering discipline, models are used to cope with system complexity. In addition, they are used as reusable and analysable artefacts to bridge the conceptual gap between requirements and target system implementations. A variety of sophisticated approaches exists for visually representing interdisciplinary models [SW10, Bas+11, Thr13]. However, our focus is on a formal model-based approach to manage the evolution of aPS, which integrates suitable means for quality assessment and automatic verification assuring model correctness in an interdisciplinary way.

In the MoDEMMiCAS project, we model the aPS based on the FOCUS theory [BS10], which is based on a (formal) foundation and provides well-elaborated notions of the system components and their interfaces, composition, and refinement. By that, formally verifying the functional conformance of the aPS' behaviour based on the composition of components, that is the system's architecture, is possible. Furthermore, the approach is fundamentally viewpoint based, providing three different viewpoints onto the aPS, which refer to different engineering concerns: requirement, process, and system. The system viewpoint consists of three scopes (see Fig. 10.5):

- **Context Scope** mainly contains information about the geometry of mechanical components.
- **Platform Scope** comprises everything from the sensor/actuator interface with the context scope to the programmable controller's variables available interfacing the software scope.
- **Software Scope** contains a model of the software architecture, including its components and their behaviour.

In order to ensure system correctness regarding the availability requirement as a part of the quality assurance of the aPS, the system behaviour model is extended with deviation models, which represent fault occurrences and effects of

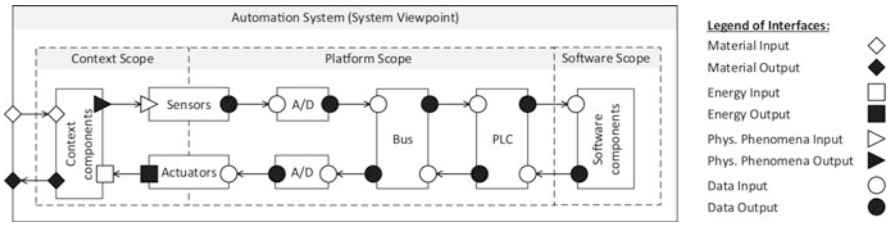


Fig. 10.5 Schematic overview of system control loop

the system components [Mun+17]. Within the presented approach, our focus was on the availability as a degree of correctness rather than binary distinction (i.e. correct or incorrect). Based on the extended specifications (i.e. deviation models), the actual behaviour of the plant is verified by means of failure definition and aggregation models and finally compared to requirements based on models of availability metrics. This is realised by translating the model for the probabilistic model-checker PRISM. The deviation model consists of three components, namely an input and output filter to model the altered behaviour of the component under consideration and an activation function (act) that represents the (de-)activation of those filters. In an aPS, faults may occur in the system's software, platform, or mechanical context. The exemplary case using extended Pick and Place Unit (xPPU) and the detailed verification procedure of this approach is described in Chap. 11.

Towards incremental model change verification, we consider the integration of verification technique (i.e. model checking) into a continuous integration (CI) environment. Maintaining high quality is the most important piece of delivery in a CI environment. Most development frameworks improve software efficiency by implementing automated processes, as well as imposing checks on the model. CI takes it a step further by finding any model integration issues by compiling the model as soon as the developer checks it in and running unit tests. Build status and unit test catch the initial errors in the model during compiling. However, those tests are insufficient to catch the deployment- and integration-related errors. Furthermore, continuous build and test systems today are used as part of the development activities to control and improve the process of software creation in software engineering domain [DMG07] and for the software of the aPS [Vog15]. However, adapting CI for other disciplines of aPS is challenging [Vog15].

Within the CI environment, incremental changes are applied on a regular basis to the aPS model and are integrated continuously. To ensure a conformance level of the overall system quality, continuous verification techniques that provide support for incremental and compositional verification are required [Vog+15c]. In other words, instead of checking the entire system model each time a change is applied, incremental and compositional verification is limited to the changed parts and the affected parts of the system model, as well as the environment model. Within CI, verification is organised along an increasing verification scope, starting with verifying the software in isolation (Fig. 10.6). In case the additional efforts for

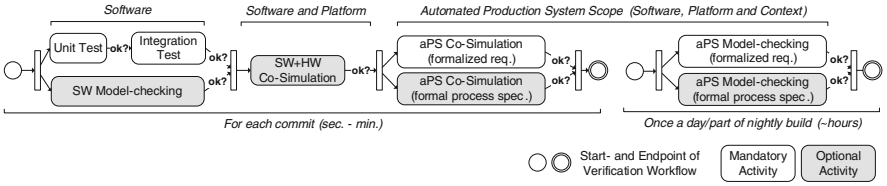


Fig. 10.6 Verification workflow as part of a CI approach [Mun+18]

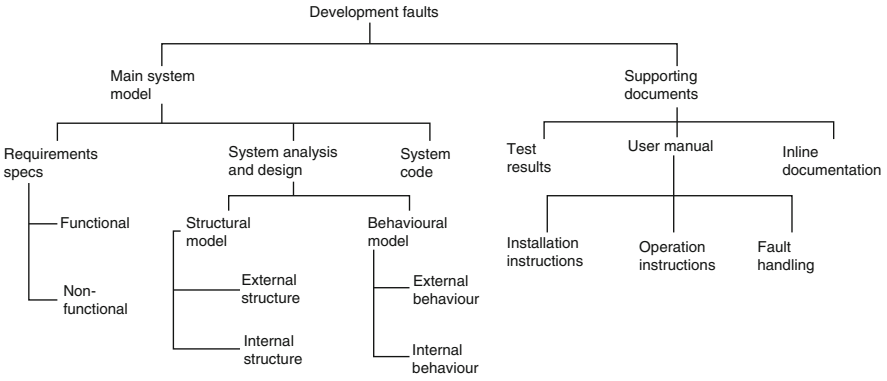


Fig. 10.7 Taxonomy of faults arising from incremental changes [BV18]

specifying logical constraints at the software level are acceptable, exhaustive model checking of the control software can be done in parallel to testing. After the software system is verified at the software scope, we use co-simulation to verify if it also has the desired effects on the automation hardware and, ultimately, achieves the desired plant behaviour. Finally, the continuous verification procedure suggests performing a co-simulation against formalised requirements, a formal specification of the technical process, or both. In case an automated hardware setup exists, this step can also be replaced by hardware in loop verification.

Based on an analysis of the incremental changes throughout the entire aPS life cycle, we develop a fault taxonomy [BV18]. This taxonomy builds on [Lau99], where faults are classified based on their cause into either physical faults due to changes in the underlying technical process or human faults associated with changes before and after commissioning. The taxonomy presented here is for analysing the effect of incremental changes associated with CI and hence focuses only on human faults occurring before commissioning. As illustrated in Fig. 10.7, development faults resulting from incremental changes to aPS span the different activities of the development life cycle. Due to the differences in their influencing effects on the overall aPS quality, development activities contributing to generating artefacts modelling the system are distinguished from those resulting in supporting documentation. Where faults in the former category may have an impact on the entire set of quality criteria, faults in the latter basically affect criteria such as

usability, maintainability, and portability. For example, a change in the user manual in the section regarding the operational instructions may harm operability, which is a sub-criterion of usability.

A fault in the system model may occur during requirements specifications, system analysis and design, or system coding. A change in the system's requirements is associated with faults either in the functional or in the non-functional requirements. With the help of data associated with the changed artefact, a more precise prediction of the impacted quality criteria can be derived. For instance, a change of a non-functional requirement that indicates that the system's uptime shall be 99% is deemed to influence the availability criterion of the system.

Artefacts generated during analysis and design are decomposed according to their purpose into artefacts constituting the structural model and those forming the behavioural model. Each of these models is further decomposed out of the adopted point of view into external and internal. A fault in the external models basically affects the interrelationship between the aPS and the external actors, including the system user, as well as any interacting legacy systems. Accordingly, faults in the external categories are related to quality sub-criteria such as operability, interoperability, and coexistence. On the other hand, faults belonging to the internal structural and behavioural models need further investigation based on more specific data about the associated context to derive their expected impact on the quality criteria.

10.2.2 Using Regression Verification for Small Evolution Steps in PLC-Code

Formal verification proves the implementation correctness mathematically and exhaustively with respect to the formal specification. In spite of the full coverage, formal verification is not commonly used in aPS engineering for the functional implementation of the system. One of the barriers is achieving formal specification to use the formal verification. In our approach, we apply regression verification methods that do not require full functional specifications and minimise the complexity of the verification problem.

Verifying of the PLC program with respect to temporal specifications, for example especially safety and liveness, has been the subject of research in the automation field [Bec+15]. Formalisation of the PLC program [YF03a] and its validation [Lam+99a] have already been discussed, and various approaches to verify the behaviours of industrial machines using formal methods are suggested, for example [Wit+06]. Also, various transformation methods from IEC61131-3 languages into model checking available languages are suggested, for example from Sequential Function Chart (SFC) into Process Meta Language (PROMELA) for SPIN model checker [BM00] or into timed Computational Tree Logic (tCTL) for UPPAAL [Bau+04b] and from all IEC61131-3 languages into symbolic model

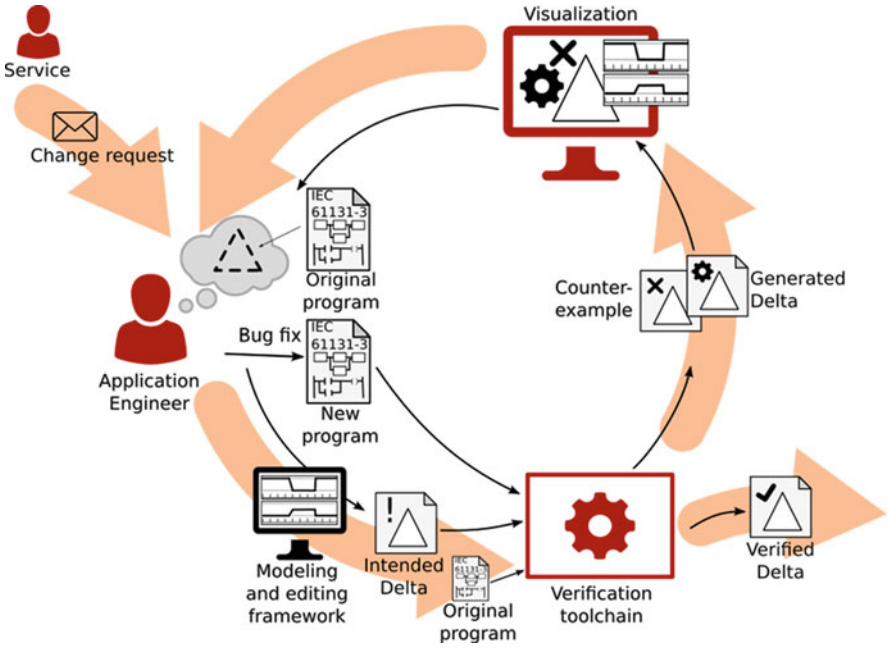


Fig. 10.8 Overview of user-centred regression verification process

for Cadence-Symbolic Model Verifier (SMV) [De +00]. However, for applying verification, the complete formal specification is still the bottleneck even though the system model can be obtained using the transformation methods.

In the Improve APS project, we deal with regression verification in a user-centred software development process for evolving aPS [Bec+15] (Fig. 10.8) by including the user into the process loop with more human-understandable notations and visualisations. A major prerequisite of this project is that the requested change is implemented on an already-implemented and being-operated system after the acceptance test or at least after the system is approved by customers. Change requests happen due to either the necessity of the different functionality or faults in the system. Once a change request is issued to the application engineer, the new change is implemented based on the existing code version to fulfil the change request. When the modification is done, the engineer verifies the regression of the code behaviour according to the original program and its deviation (i.e. intended delta). If it is verified that the new program satisfies the change request, the code block can be launched to the system. However, if it is not, the verification tool generates a counterexample and also actual differences between original and new, that is generated delta. (Follow the loopback path in Fig. 10.8.) This counterexample is provided to the engineer together with the delta information.

This overall process is suggested as a verification-supported evolution tool chain for automation software application engineers [Ule+16] with three phases. First,

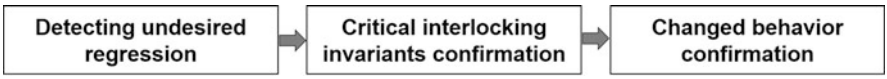


Fig. 10.9 Overview of aPS verification process

#	Input			Output			Dur.
	A	B	C	X	Y	Z	
0	1	1	2	0	0	5	1
1	0	3	3	6	6	5	7
2	1	4	2	2	8	5	2

(a)

#	Input			Output			Dur.
	A	B	C	X	Y	Z	
0	1	1	2	0	0	-	1
1	-	p	p	$2*p$	X	$Z[-1]$	>5
2	-	$p+1$	-	$[0,p]$	$>Y[-1]$	$2*Z>Y$	*

(b)

Fig. 10.10 An example of a concrete test table and a generalised test table from [Wei+17]

regression verification checks whether the retained behaviours in the new program is equivalent to the old version for all cases where no change in behaviour is intended. Second, the critical properties which need to hold also in the new software system against the interlocking invariant properties, for example safety properties. Third, the new behaviours are verified against how the program is expected to behave differently based on the properties describing the new situation (Fig. 10.9).

The precise verification technique in this project is explained in Chap. 11. However, even after setting aside the verification techniques, there arise user interaction issues. To verify a program, the application engineer should provide the verifier (model checker) a piece of software or its model and the property that needs to be verified in the target program. Also, the engineer is required to be able to understand the verification results, which consist of a success/fail verdict and a counterexample in the event of a failed case. Since formal notations are one of the barriers for using formal verification [Pak+16] and the counterexample of the verifier is usually a simple series of the values for a trace, this interacting information, which is provided by and provided to the user, needs to be represented in an easier way for the users than existing formal expressions.

As one possible user-friendly specification, we have extended the concept of test tables, which the software application engineers use for reactive system testing [UV15]. In [Wei+17] and [Bec+17a], we suggested an approach to support quality assurance by generalising the test tables such that they can be used for formal verification purposes in addition to testing.

An example of test table is shown in Fig. 10.10. A test table consists of three parts of expected input, its corresponding output, and allowed duration of the appearance of input-output pair with fixed values (Fig. 10.10a). Rows of the table describe the execution progress of the system under test. The example represents three input variables (A, B, and C) and three output variables (X, Y, and Z) during the duration of 10 s. The generalised test table represents all possible test table cases of the sort by generalising the cells with constraint expressions, while the test table just represents

one specific or concrete test sequence. In the generalised test tables (GTTs), three generalisation concepts are used, such as below (Fig. 10.10b):

- Abstraction using constraint expressions, for example mathematical formulas (as in cell $(1, X)$), interval expressions (as in cell $(2, X)$, which means "a value in the interval from 0 to p", and $(2, Z)$, which means "the value which satisfies $2 * Z > Y$"), or *don't care* (as seen as "-")
- Relating cells using existing input/output variable names, together with relative cycle index (as in cell $(1, Y)$ for current value of the other cell and $(1, Z)$ for a value one cycle ago) or new variable symbols (as in cell $(1, B)$ and $(2, B)$)
- Various duration using interval expressions (as in the second row, which means "the row must be repeated more than 5 s") or *don't care*, which is equivalent to $>= 0$ (as seen as "$*$")

Using this GTTs, the developers can describe the specification of the function more easily and effectively since it is an already accepted form. And this specification is used by the verifier to prove the function behaviour. Together with these concepts, a graphical interface, that is Structured Text Verification Studio (STVS), is implemented for hands-down automation software proof against the GTTs, which is developed in IEC 61131-3 Structure Text (ST) (Fig. 10.11). Once the ST code and the GTT are typed in, the model checker called from STVS (nuXmv) verifies the program against the specification in GTTs. If not, counterexample in the form of concrete test table is displayed that violates the specification, that is GTT.

We also presented a monitoring block generation method based on the GTTs [Cha+17]. Since GTTs describe allowed behaviour sequences, a violation of this table means either a violation of the expected output for the given input or a violation of the expected input. The code of the monitoring block can be generated by converting the GTT systematically (see [Cha+17]). This block checks the input-output pair sequentially according to the defined description in GTT. This monitoring block can interact with the user by raising appropriate signals depending on its decision results: (1) in the case of *warning*, which means the output violation for the given input, the function block needs to be adjusted in order to execute as the specification defines and (2) in the case of *unknown*, which means the input violation for the specific sequence, the specification needs to be adjusted, that is revised or added, to cover the situation that occurred. This monitoring is applied during the machine execution. User interaction using the monitoring block decision results can be implemented in various ways, such as logging, displaying on human machine interface (HMI) system, and so on.

10.2.3 Conclusion and Outlook

In this section, we explored quality assurance approaches for aPS. Formal verification is applied to the incremental changes of aPS, and by introducing formal methods into the aPS engineering process, we can achieve higher quality aPS by

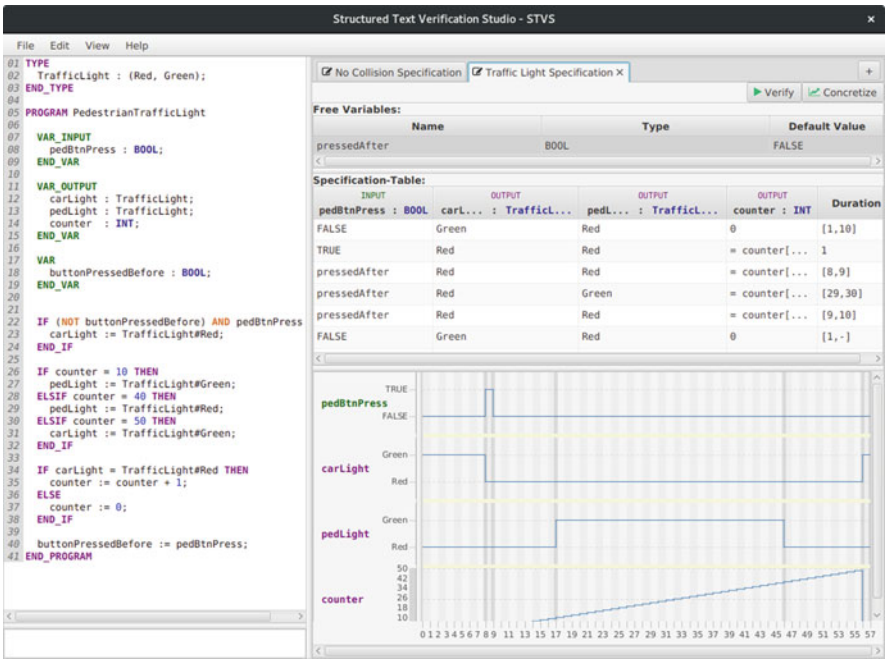


Fig. 10.11 Structured Test Verification Studio (STVS)

verifying them exhaustively in the model level, as well as in the code level. For model verification, we suggest a quality assurance (QA) model to verify the system in the CI approach capturing incremental changes. For code verification, regression verification is applied to verify system behaviour according to specification, as well as the new specification method suggested.

As a future work, we intend to enrich fault taxonomy model by analysing industrial practical cases to make it more generalised. Moreover, the applicability of the prototypical tool chains on the real machines needs to be measured, for example CI framework, by bonding the solutions with suggested bridges between them, or STVS plug-ins for IEC 61131-3 development environments, such as CODESYS and TwinCAT.

10.3 Analyse Non-functional Aspects of the System

Complementary to approaches which target a formal analysis of the functional aspects of software quality, the methods presented in this section consider non-functional quality aspects such as performance and robustness. Non-functional quality aspects are very crucial to ensure the quality of complex systems [II11]. However, due to tight time and cost restrictions in practice, evolution is often

performed without pre- or post-evaluation of its influence on non-functional quality aspects [LFL16]. Reasons for that mainly include lack of good requirement engineering practice, cf. [MMW98, PR11]. One method to overcome this lack of quality based evolution evaluation is an automated analysis of the actual behaviour which is constantly performed during the evolution [Vog+15a]. To do so, models reflecting the current system behaviour and capturing the non-functional properties are needed. However, building such models manually is complex, tedious, and error prone, which contradicts aforementioned tight time and cost restrictions in practice [CM09]. One way to overcome this drawback is to enhance the models with runtime information like observed signal events [Epi+09] or even to completely generate models out of runtime data gathered from current system behaviour. To support evolution, it is shown in the following how different types of models capturing a variety of non-functional properties can be developed and learned from observation to analyse the system quality during the evolution of the system. Examples of the models and their automated generation are shown on the (x)PPU case study plant.

Firstly, Petri Nets are automatically derived for software-controlled manufacturing systems by the observation of input/output events [Lad+15a, Lad+15c]. The models are suitable for the performance and flexibility analysis of the interdisciplinary system. The Petri Nets are defined in a way that they reflect those non-functional properties. In production systems, these are typically metrics defined for each manufacturing plant, such as the throughput rate, the allocation ratio, or the utilisation efficiency [Lad+13b]. Secondly, Markov chains are used to analyse the system's reliability. A novel approach [FGL15] is presented that learns the Markov chains from the running system. The Markov chains can then later be used for probabilistic model checking [KNP04].

10.3.1 Learning and Analysing the Machine States and Material Flow of Evolving Manufacturing Systems

production automation systems have a strong focus on non-functional properties of the underlying production process. This production process is defined in the production plant's specification and inherently implemented in the plant's structure and behaviour. But when considering evolution, a systematic re-engineering for adapting production plants is often omitted and changes are directly implemented based on the informal requirements [FL00, Vog+15a, BS10]. Accordingly, this results in a gap between the actual production process and its specification [Hau+13]. Therefore, no (formal) up-to-date models of the plant are present, and especially evolution, including small and unanticipated changes, usually lacks appropriate documentation and evaluation [Vog+15a].

As a result, non-functional properties may be unsatisfied and unnecessary weaknesses are even not recognised by the staff. One approach to overcome these deficiencies is using automated model learning for creating formal behaviour

models of the plant's production process, which is suitable for both automatic recognition of changes and their analysis regarding non-functional properties of the process. Contrary to the reliability approach for learning Markow chains, this learning approach concentrates on the structure of a production process and the order in which signal events occur in the plant. Therefore, this approach deals with generating Petri Nets from observations gathered at discrete inputs and outputs of *Programmable Logic Controllers (PLCs)* controlling production systems.

Three-Phase Evolution Support Process

In the following, an approach is presented that aims at automatically generating models from externally observable events of production systems. Figure 10.12 shows the process of the approach in three phases. In the first, phase knowledge about the underlying production process is gathered by observing input and output signals of the production system. In the second phase, an evolution cycle is established by learning observed behaviour in models at runtime and using them to detect changes that must be relearned. In a third phase, the evolution is assessed by deriving non-functional properties out of the learned models. Each phase is further described in more detail.

The first phase is data *acquisition*, in which data in terms of event traces are recorded. In order to lift the data to knowledge in terms of non-functional property values, the data have to be made semantically interpretable. Therefore, further meaning is added to the events by information modelling. An information model is intended to provide semantics of the event generating signals in order to be able to generate meaningful behaviour models. However, manual effort for creating an information model has to be kept to a minimum to meet the requirements regarding tight time and cost restrictions. Therefore, in contrast to a complete manual modelling of release and consumption of events in the plant, the semantics provided by the information model only contain the assignment of signals to the plant topology and the types of recorded signals [LFL16]. Types of signals are, for example, workpiece detection (e.g. stemming from a light barrier), workpiece

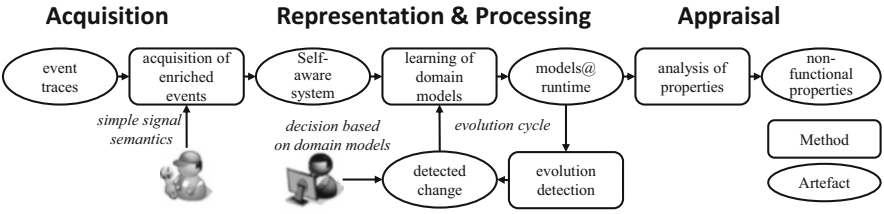


Fig. 10.12 Acquisition, representation and processing and appraisal to determine non-functional properties

modification (e.g. actuator turning on a drilling machine), or state detection signals (e.g. motor on/off). As a result, semantically enriched event traces are generated.

The second phase is called *representation and processing*. In this phase, the behaviour of the plant is represented in models that can be used and changed at runtime. Two different model types have been developed to apply the approach on manufacturing systems: Machine State Petri Nets (MSPN) and Material Flow Petri Nets. MSPNs represent the full behaviour of every single technical resource in the plant, and MFPNs capture the routing of workpieces of the whole plant. The models on one hand are used to detect evolutionary changes by using anomaly detection mechanisms (see Sect. 9.6) and on the other hand are used for analysing them regarding non-functional properties, as shown in this section. An overview of the model types, their automatic generation, as well as their analysis, is given in the following sections. More details can be found for MSPNs in [Lad+15a] and for MFPNs in [Lad+15c].

Both models are based on the state of the system, which is given by a vector of input/output binary sensor and actuator signals. The models are based on the recorded event traces with the semantics of the acquisition phase. One advantage of using the added signal semantics is the possibility of dividing the input/output vector of the plant into subvectors that reflect specific parts or specific aspects of the plant. For example, the event trace can be filtered by signals assigned to just one considered resource or reflect a specific action in the system.

Formally, both models are represented with Place/Transition (P/T) Petri Nets $\langle P, T, F \rangle$ in which transitions are annotated with signal names. T is the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the set of directed arcs between places and transitions.

First, for each technical resource in the plant, one MSPN is generated from data observations reflecting the resource's behaviour by representing its full language in terms of all input/output events, as well as their timing dependencies. To do so, a preprocessing step combines events in a defined time interval to avoid an increase of model complexity due to imprecise and scattered timestamps. The actual learning is based on the causality specification of Lefebvre and Leclercq [LL11]. It uses an event propagation matrix that contains all direct successors of events. This matrix is used to calculate an incidence matrix, which replicates each successor relation with its firing sequences. In a last step, this matrix is reduced in complexity and its initial marking that represents the current state of the system is calculated. The permittivity of the MSPN is further reduced by exploiting the fact that binary signals can just toggle between their two states. Furthermore, timings are annotated as meta-information that is used to find timing anomalies and to derive performance properties.

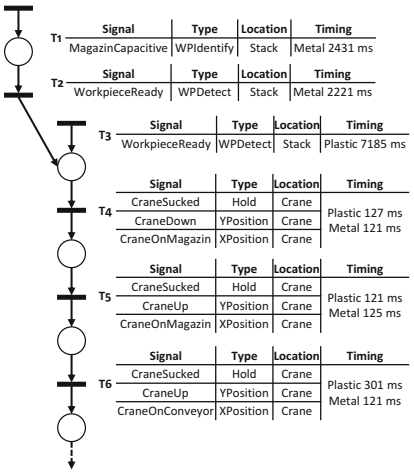
In MFPN, the material flow is modelled, whereby tokens represent workpieces. In an MFPN, transitions represent events when one sensor or a combination of sensors detects a workpiece. Further, a place represents a region in which the workpiece is not detected by a sensor. As an example, consider a running conveyor belt that transports workpieces and a light barrier that detects workpieces on this belt. When a workpiece passes the light barrier, the binary sensor of the light barrier

indicates the position of the workpiece, which in the MFPN is represented by the firing of a corresponding transition. When the light barrier is passed, the workpiece is in a region between two sensors, and therefore in the MFPN the token is situated in the place after the fired transition. To learn the MFPN first, the trace is separated according to the events' affiliation to the equipment in the information model, and all non-relevant events are filtered out. The actual learning is done by analysing the time stability between events in order to determine which events are triggered by the same workpieces during transportation through the plant. Subsequently, in a first step, the events are assigned to workpiece instances. If the same events are triggered several times within a stable time difference, they are assigned to the same workpiece instance. By using all instances for each workpiece type, a place-transition chain is generated. These chains are combined in such a way that the number of places and transitions is minimised and the net still fulfils the definition given above. As a last step, similar to MSPNs, timings, as well as identifications for workpiece types, are annotated.

The models are suitable for analysis regarding non-functional properties to evaluate changed behaviour of the underlying production process. Such an analysis takes place in the third phase of the process of Fig. 10.12, which is called knowledge *assessment*. The properties of interest here are mainly *Key Performance Indicators (KPIs)*, as defined for production systems in the ISO 22400-2 [DIS12]. These properties have to be operationalised in order to be measured directly at the production machine. An extensive literature research of such properties was done in [Lad+13b]. As an illustration, the throughput rate of a manufacturing plant describes the number of produced goods per unit time [DIS12]. Accordingly, it can be calculated by determining the number of produced goods during a specific production time. To determine the production duration, the time difference between the first event and the last event in the event trace is calculated. By summing up all mean transportation durations of each workpiece type in the MFPN, the mean production duration for one product of each type can be calculated.

Application on xPPU Case Study

For evaluation, an implementation of the knowledge-carrying software has been built based on a service-component architecture [Hau+14a]. It uses event source endpoints in a 1-to-1 mapping according to the available entry points of the production process. Each part of the physical plant hierarchy is mapped to self-contained representation components, which are in charge of implementing a consistent system state. The models are implemented as runtime artefacts that are decoupled from the real-time plant by using event statements that are learned and evaluated at runtime [Hau+17]. The software cannot decide if a performed change and its influences on its non-functional properties are intended (or at least acceptable); therefore, Fig. 10.12 includes a cycle with a "user in the loop" to decide whether an intended evolution has taken place (in which case the model has to be re-learned) or an undesired change in the production system occurred (which should



	Sc 1	Sc 3	Sc 5	Sc 10	Sc 11	Sc 12
$MFPN\ (T)$	5	12	12	18	17	15
$MFPN\ (P)$	5	11	11	16	16	14
$MSPN_{Stack}\ (T)$	11	13	19	28	22	22
$MSPN_{Stack}\ (P)$	8	9	11	15	13	12
$MSPN_{Crane}\ (T)$	34	30	37	45	41	54
$MSPN_{Crane}\ (P)$	34	30	37	45	41	54
$MSPN_{Stamp}\ (T)$	-	13	16	18	18	16
$MSPN_{Stamp}\ (P)$	-	12	13	13	14	13
$MSPN_{Conveyor}\ (T)$	-	-	-	15	24	32
$MSPN_{Conveyor}\ (P)$	-	-	-	11	19	23
$Utilisation_{Stack}$	49%	73%	80%	92%	92%	94%
$Utilisation_{Crane}$	100%	69%	87%	85%	82%	93%
$Utilisation_{Stamp}$	-	10%	37%	40%	18%	24%
$Utilisation_{Conveyor}$	-	-	-	38%	38%	35%
$Makespan_{White}$	-	-	-	20s	20s	28s
$Makespan_{Black}$	7s	10s	12s	11s	11s	10s
$Makespan_{Metal}$	-	15s	22s	26s	21s	23s

Fig. 10.13 (Left) Part of the resulting MFPN with semantics of signals. (Right) Utilisation and Make Span properties of some PPU scenarios

be fixed by an operator). This process is implemented by a goal-based management component that is based on the Belief-Desire-Intention architecture [Pok+14].

The MFPN learning algorithm has been evaluated on the Pick and Place Unit (in the state as described in [Vog+14b]). Figure 10.13 shows on the left side a part of the learned MFPN of Scenario 3. It shows the part where the workpieces are introduced into the PPU by the stack and picked up by the crane. The MFPN has two starting points, because the plastic workpieces are first detected by the *WorkpieceReady* sensor, which is a switch at the stack slide, and metallic workpieces are first detected by the *MagazinCapacitive* Sensor, which is mounted at the slide. Plastic workpieces do not show the *MagazinCapacitive* event since the sensor remains in its 0 state. Nonetheless, both workpieces are transported by the crane. Also, the semantic information is shown for the signals. For example, the crane has a combined signal for transition *T4*. This signal contains the *CraneSucked* signal, which indicates that a workpiece is in the crane, and two position signals of the crane (*CraneDown* and *CraneOnMagazin*). Also, the timing, distinguished by their type, for the combined signal is given.

Further on the right side of Fig. 10.13, an overview of the learned models is given in a table. It is indicated how complex the models are in terms of learned places and transitions. As a summary regarding the PPU case study, it can be stated that the PPU is steadily extended, which results in more complex models. Further, the crane is the most complex resource.

The resulting MFPN allows calculating relevant process-related non-functional properties, as described in the assessment phase. As examples, we show the makespan and utilisation on the right side of Fig. 10.13. The utilisation is measured on individual resources of the PPU, and the makespan is defined for workpieces.

Utilisation describes the amount of time a resource has operated on a given workpiece within the frame of the total time that the entire plant has taken to produce the given workpiece from beginning till the end. Hence, the value of utilisation of a particular given production module can have the maximum value of 100% if it has been in operation the entire time. One other property that we explored is the makespan for an individual module. Makespan is defined as the time that elapses from the beginning till the end of an operation performed on a given workpiece. For example, a workpiece is being shifted from point A to point B with the aid of a crane. In this case, the makespan of the crane would be the time taken for the crane to move the workpiece from point A to point B.

One exemplary finding out of these non-functional properties is that the optimisation of the crane behaviour of Scenario 5 indeed increases the utilisation of the crane (69% $\rightarrow$ 87%) but that the makespans of the products are not improved by the optimisation as intended (10 s $\rightarrow$ 12 s and 15 s $\rightarrow$ 22 s). This results from the long time the crane needs to turn between the stamp and the stack. Therefore, the "optimisation" of Scenario 5 is not advisable as an evolutionary change.

10.3.2 Learning and Analysing DTMCs for Reliability Evaluation

The learned Petri Nets from the previous approach are mainly used for performance evaluation. For reliability and safety, based on the results of the joined MOCA and ENSURE research on model-driven co-evolution (Sect. 10.1), we know how to co-evolve architectural models and probabilistic quality evaluation models, such as fault trees and Markov models or queuing networks@runtime. Markov models like Discrete-Time Markov Chains (DTMCs) are mainly used for reliability evaluation.

Since the behaviour of a system may change at runtime, for example user request rates for web systems, we also require a time-efficient, robust, and accurate learning algorithms to keep the parameters, for example transition probabilities, of a probabilistic model continuously updated. Most of the available approaches [CJR11, Cal+14, Epi+09, EGT10, ZWL08] achieve only one of different goals (time efficiency, robustness, and accuracy) at the price of the other.

Before we can go into the details of learning transition probabilities for DTMCs, we first need to define what a DTMC is. A DTMC is a state-transition system where the choices among successor states are governed by probability distribution. Formally, a DTMC is a tuple (S, s_0, P, L, AP) [BK08], where S is a (finite) set of states, $s_0 \in S$ is the initial state, $P : S \times S \rightarrow [0, 1]$ is a stochastic matrix, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labelling function that associates to each state the set of atomic propositions that are true in that state. An element p_{ij} of the Matrix P represents the transition probability from state s_i to state s_j, that is the probability of going from state s_i to state s_j in exactly one step.

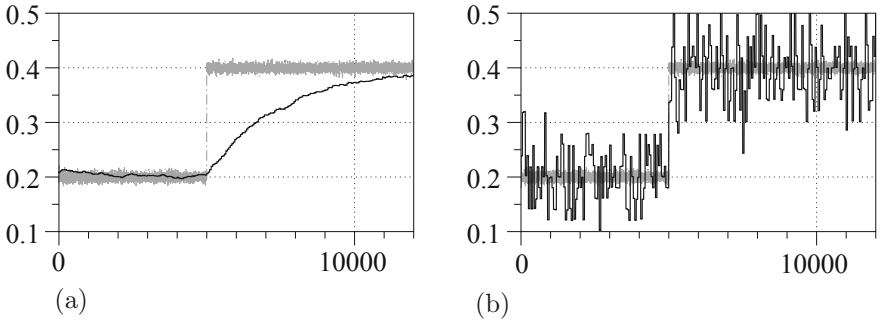


Fig. 10.14 Examples for behaviour with different a parameter values. (**a**) Noise filtering ($a = 0.98$). (**b**) Fast change tracking ($a = 0.02$)

The probability of moving from s_i to s_j in exactly two steps can be computed as $\sum_{s_x \in S} p_{ix} \cdot p_{xj}$, that is the sum of the probabilities of all the paths originating in s_i, ending in s_j, and having exactly one intermediate state. The previous sum is, by definition, the entry (i, j) of the power matrix P^2. Similarly, the probability of reaching s_j from s_i in exactly k steps is the entry (i, j) of matrix P^k. As a natural generalisation, the matrix $P^0 \equiv I$ represents the probability of moving from state s_i to state s_j in zero steps, that is 1 if $s_i = s_j$, 0 otherwise.

As time-efficient and accurate learning probabilities for DTMCs, the following lightweight adaptive filter (LAF) has been developed in the ENSURE project:

$$\begin{cases} e(k) = |p^m(k) - \widehat{p}(k-1)| \\ a(k) = a_0 + \Delta a \cdot f_a(e(k) - e_{thr}) \\ \widehat{p}(k) = a(k) \cdot \widehat{p}(k-1) + (1 - a(k)) \cdot p^m(k-1) \end{cases} \tag{10.1}$$

This filter, as shown in Eq. (10.1), is an extended version of a unity-gain, first-order discrete-time filter [Lev10]: $\widehat{p}(k) = a \cdot \widehat{p}(k-1) + (1-a) \cdot p^m(k-1)$, where a is a tuning parameter $0 < a < 1$. A value of a close to 1 provides a really good noise filtering, as can be seen in Fig. 10.14a, whereas a value of a close to 0 allows the filter to track changes really fast. To have a good performance, we have extended the filter (see Eq. (10.1)) to dynamically adapt the parameter a based on the characteristics of the data p^m we measure. This is done with the two functions $a(k)$ and $e(k)$. For details, we refer to the paper [FGL15]. The filter can be used to learn individual transition probabilities p_{ij} from observed system traces. However, the obtained estimates for each row of P would most likely not constitute correct categorical distributions (sum of the probabilities of the outgoing transitions of each state s_i is not 1). Consequently, [FGL15] presents a procedure to use these learned probabilities to update the provability matrix P of the discrete-time Markov chain (DTMC) via a convenient "correction" procedure. Furthermore, all existing procedure [CJR11, Cal+14, Epi+09, EGT10, ZWL08], including LAF [FGL15], estimate the probabilities based on the observed data; for further studies,

Table 10.1 MARE results for the six patterns

	LAF (%)	Kalman (%)	Cove (%)
Noisy	4.41	4.54	11.70
Step	4.19	8.51	8.65
Ramp	4.28	7.04	8.27
Square	6.54	21.47	8.50
Triangle	7.79	12.84	8.16
Outlier	3.68	3.78	8.56

it would be also interesting to apply forecasting procedures [AGC12, ACG12] to estimate the future development of the individual transition probabilities p_{ij}.

In the original paper [FGL15], two experiments have been performed to evaluate the quality of LAF. The first experiment evaluates the accuracy via the Mean Average Relative Error (MARE) and data that are generated to follow six common patterns: Noisy, Step, Ramp, Square, Triangle, Outlier. The results, as shown in Table 10.1, indicate that the developed algorithm (LAF) outperforms in terms of accuracy (MARE) the state-of-the-art approaches Cove [Cal+14] and Kalman filtering [ZWL08].

For the second experiment, [FGL15] uses a realistic and large-scale dataset of the users' browsing behaviour for the WorldCup98 website [WCL]. The logs of the browsing behaviour spread over a period of 3 months, and the website is composed of a total of over 32,000 pages. As a result, we show that LAF is able to scale to problems of a realistic size and complexity.

As mentioned before, in Sect. 10.1, we could obtain the structure of DTMC models. Together with LAF approach, learned models can be synthesised and verified using model checking techniques (e.g. Prism) specified with probabilistic computational tree logic (PCTL). Hence, we can reason about the reliability of large-scale systems by using probabilistic model checking.

Related Work

Many approaches using automated model generation from the observation of production systems stem from the domain of fault detection and isolation (FDI). Generated models are continuous models [Ise06, AA13], discrete event models [HKW03, LL11, RLL10], or hybrid models [Vod+11]. Approaches for continuous systems use machine learning techniques to, for example, parametrise differential equations, train neural networks, estimate initial states for observers, or estimate parameters for fuzzy models; cf. [Ise06, AA13]. Some approaches learn hybrid models of systems composed of both discrete and continuous dynamics; cf. [NF15]. Such approaches are suitable for process plants, that is plants dealing with continuous product flows (e.g. liquids, gases, or granules). The state of such plants is given by both continuous measurements (such as temperatures, pressures, or flow rates) and discrete measurements (such as open/close states of valves or on/off states of pumps).

The states of discrete manufacturing systems (i.e. production systems producing piecewise goods) are usually described by purely discrete state variables. These states are consisted of binary measurements (such as "workpiece present/not present" or "conveyor running/not running"). For discrete event systems, learning algorithms mostly generate automata (cf. [RLL10]) or Petri Nets (cf. [LL11]). However, manufacturing processes are usually highly concurrent, resulting in a huge amount of possible states. The cyclic control and difficult real-time conditions render it problematic to weave code for monitoring into the controlling software. Therefore, only the event traces, observed by monitoring the sensor and actuator signal changes, are available as a data source for monitoring approaches. Unfortunately, models generated from signal traces tend to be highly complex and are on a low level of abstraction compared to non-functional properties. The algorithm presented in [RLL10] reduces model complexity by automatically dividing the gathered data into subsets and creating partial automata. However, without any semantics of the data, an interpretation on non-functional property level is still rarely possible. Other approaches, for example, [AT12, HA05, Hus+06] use static a priori information. This includes modelling manually, which event consumes or releases resources for these processes. This allows an interpretation of generated models on a higher level of abstraction. But an automated analysis is not in the focus of those approaches, and therefore these approaches are, like most presented models for production systems, by design not suitable for automatically deriving non-functional properties during evolution.

Further Reading

A more detailed overview of possible non-functional requirements that can be derived is given in Ladiges [Lad+13b] and shown for the PPU, besides a more detailed description of the overall approach in Lagides [LFL16]. Anomaly detection to identify evolution changes is given in Chap. 5. How the models are handled within a knowledge-carrying software is shown in Haubeck [Hau+14a] and Haubeck [HLF18]. Further, Sect. 10.4.2 builds this knowledge-carrying software and the learned models by identifying evolution steps that can be exchanged in order to recommend changes by comparing slightly different but similar systems.

10.3.3 Conclusion and Outlook

This section has shown two approaches using learning algorithms for automatic model generation or model parametrisation. The resulting models are suitable for analysing current system behaviour regarding non-functional properties. The first approach aims at generating models of the machine state [Lad+15a] and material

flow [Lad+15c] reflecting the behaviour of manufacturing systems. The approach contains different levels of knowledge. The lowest level is raw data. By adding the information model, that is the signal semantics, the data are made interpretable. Therefore, the considered events are first filtered and separated along the signal semantics, and then the models are learned out of traces. The presented models of the machine state and material flow, as well as the properties resulting from their analysis, are the top levels of contained knowledge and allow conclusions regarding evolution.

The second approach aims to efficiently learn the transition probabilities of a Discrete Time Markov Chain based on the observed traces from the actual running system [FGL15], and based on this model quantitative verification techniques can be applied. These quantitative verification techniques check whether the model satisfies quantitative properties, for example in PCTL (Probabilistic Computation Tree Logic) [HJ94], that can be defined via a specification pattern system [Gru08] and a controlled natural language [Aut+15].

The models of the presented approaches can be used to derive differences, as shown in Sect. 10.1.1. This allows identifying differences that result in changes of non-functional quality aspects, which were derived by the two approaches presented here. Further, runtime monitoring of non-functional quality aspects has synergies with model-driven development that deals with requirements in an interdisciplinary model (see Sect. 10.2.1), because monitoring allows the detection and evaluation of changes in the running system that can be transferred back to the model-driven models to use the advantages of both approaches (see [Hau+14b] for details).

10.4 Recommend and Assess Future Changes Based on Past Changes

The previous sections have proposed approaches for a wide range of evolution problems considering changing systems. One thing that is common in these approaches is that they provide valuable knowledge about past changes, which can be utilised to support future evolution processes. Therefore, this section presents three approaches on how the current state and past changes can be used to recommend and evaluate future changes. (1) The first approach concentrates on finding recurring changes on different model versions and meta-models. By using heuristics, atomic changes on the user level are combined to complex, recurring changes cutting across different models. (2) The second approach focuses on managing knowledge by establishing a knowledge-carrying network. This network exchanges experiences of past changes between similar systems characterised by their behaviour and context. (3) The third approach derives maintenance tasks from a certain change request and for a given architecture. These maintenance tasks can be used to identify the effort for changing the system architecture, realising an “Economic Recommender”.

10.4.1 Supporting Model Editing with Automatic Recommendations

Models are key artefacts in model-driven software engineering, and software engineers typically spend much time creating and evolving models. Software engineers may even define new domain-specific models. Hence, good tool support is needed to efficiently work with models.

IDEs for source code offer many tools that make recommendations for working with text-based programming languages, for example auto completions, quick fixes, refactorings, and code templates. All major text-based programming languages can be defined via context-free grammars (e.g. EBNF), which simplifies the implementation of these source code recommenders. Also, there are few widespread text-based programming languages, which makes it feasible to develop or extend these recommenders manually.

Our goal is to improve the modelling speed and quality of model changes by recommending model changes to the engineer based on current changes and historical changes. Figure 10.15a shows an example recommendation for an Ecore meta-model. The original version of the model contains two classes with duplicated attributes. When a user adds a common superclass (green), the recommender systems notice that this change fits the pattern of the common *Pull-up-Attribute* (see Fig. 10.15b) refactoring and recommends the remaining steps, that is moving one attribute up (yellow) and deleting the other one (red). Accepting the recommendation automates several manual modelling steps.

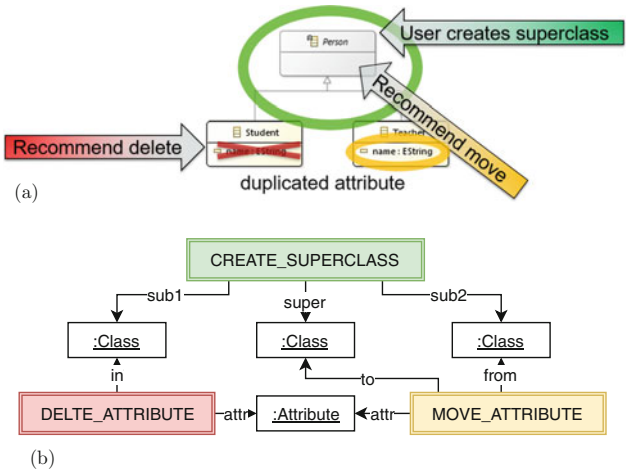


Fig. 10.15 Example of *Pull-up-Attribute* recommendation and blueprint. (**a**) Example *Pull-up-Attribute* recommendation (red and yellow) after a user creates a superclass (green). (**b**) Example blueprint for recommendation in **a**

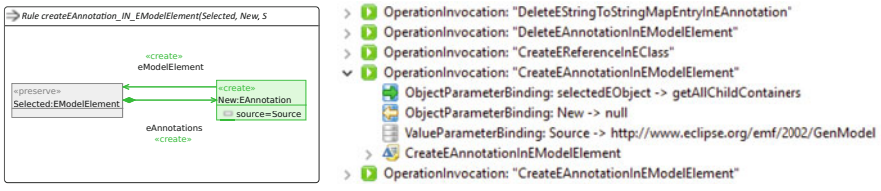
The main challenge with this approach is that for most types of models, there is no list of common refactorings or frequent changes that could be used as a basis for a recommendation. This is especially true for proprietary domain-specific models. For example, Eclipse Projects hosts over 300 projects related to the Eclipse Modeling Framework. These Projects contain over 30,000 models; most of these are Ecore meta-models and UML models for which common refactorings are known. But there are also 260 types of models with only 10 to 100 instances. It is not realistic to expect that there is any documentation about common refactorings on these less common types of models.

Foundation

Our approach is based on the SiLift-Tool [Keh+12b, KKT13] from the MOCA project, which is also described in Sect. 10.1.1. This tool computes the differences between versions of meta- and instance models. We represent differences as sets of partially ordered model transformation invocations, where an invocation consists of model transformations and their parameter bindings. Model transformations are represented by Henshin rules [Are+10] that can take multiple parameters.

Figure 10.16a shows a simple Henshin rule with two parameters: *Selected* and *New*. The Henshin rule matches the *EModelElement* (grey) to the *Selected* parameter. If this match is successful, the rule creates a new *EAnnotation* (green) and matches it to the *New* parameter. A parameter binding can be, for example, the *id* of an element in a concrete model. The Henshin rules can be generated for every type of model for which an Ecore meta-model is available, using the SiLift tool Sect. 10.1.1.

Figure 10.16b is a screenshot of a difference computed by SiLift. It contains five Henshin rule invocations. One invocation with two parameter bindings (*selectedEObject* and *New*) is expanded. The corresponding Henshin rule is shown in Fig. 10.16a. This rule matches an *EModelElement* called *getAllChildContainers* and creates a new *EAnnotation* for it.



(a) (b)

Fig. 10.16 Example model difference. (**a**) Henshin rule for creating an *EAnnotation*. (**b**) Example model difference computed by SiLift

Analysis of Change Histories

The ENSURE 2 approach is to automatically infer the information that is required for making useful recommendations from the change histories of models. This information can be thought of as blueprints for recommendations that represent common refactorings or frequent changes to a type of model.

Figure 10.15b shows an example blueprint for the *Pull-up-Attribute* refactoring. Blueprints consist of model transformations (depicted as boxes with two borders) and parameters that may be shared between these transformations (depicted as boxes with single borders). The transformation boxes with two borders contain the name of the Henshin rule, and their outgoing edges are labelled with the names of their parameters. The parameter boxes with single borders contain the type of parameter. If a parameter is shared between transformations, it is only shown once.

The blueprint in Fig. 10.15b describes the creation of a new superclass with two subclasses, where the subclasses contain an attribute with the same name. The blueprint further describes that for one of the subclasses, the attributes have to be moved to the superclass, while it has to be deleted in the other subclass.

This approach has several advantages: No expert knowledge is required, recommendations can be specific to a model type and developer or development team, and this technique can be applied to private code bases without publishing the results. Note that it is still possible to integrate expert knowledge into the approach by manually creating additional blueprints and adding them to the automatically generated blueprints. Also note that we cannot automatically infer descriptive names for the blueprints.

Inferring recommendations from a model history allows the recommendations to be tailored to the model. This has the added benefit that users can influence future recommendations automatically with their current actions. A further benefit is that users don't have to do additional work to get recommendations, as long as a history of previous versions is available.

Inferring Blueprints

The history of model changes is very noisy. A commit can contain many changes that may not be related at all. Before we can automatically infer blueprints, we have to define metrics to identify good blueprints.

In a first step, we identified the following metrics for determining the quality of a blueprint:

Occurrences This metric counts how often a blueprint appears in the change history. *Occurrences* is a simple metric that prefers small blueprints, that is blueprints consisting of few Henshin rules, because these occur more often in the history. A higher number of *occurrences* is better because it means that the blueprint was applied more often in the history of the model.

Shared Parameters Henshin rules inside of a blueprint can have *shared parameters*. Thus, a blueprint can have fewer parameters than the sum of the parameters of its Henshin rules. The number of *shared parameters* is a measure of the blueprint's cohesion; thus, a high number of *shared parameters* is better. As the number of *shared parameters* goes up, the number of *occurrences* may decrease (but not increase).

Diversity We represent *diversity* as a partial order between blueprints. If a blueprint A contains a subset of the rules in a blueprint B, then A is less diverse than B. This metric prefers bigger blueprints, as opposed to the *Occurrences* metric. More *diverse* blueprints are better because they encode more complex refactorings.

We generate blueprints by evolving a population of blueprints using a genetic algorithm. Since we have three different metrics, we must solve a multi-objective optimisation problem. Thus, we use a Pareto front of our three metrics to choose the member of our population that is taken into the next generation.

We defined two mutation operators that evolve our blueprints: *extension* and *specialisation*.

Extension tries to add a new Henshin rule to a blueprint in such a way that the new rule shares one parameter with one of the existing rules. *Specialisation* tries to share two previously unshared parameters within the blueprint. Note that both mutation can never increase the *occurrences* of a blueprint, while *extension* will increase *shared parameters* and *diversity*. Note also that the graph of Henshin rules connected via shared parameters will stay connected after the application of a mutation.

For our initial population, we create all feasible blueprints that contain a single Henshin rule.

Evaluation

Figure 10.17 depicts a blueprint learned by our genetic algorithm from the PPU Fault Tree data set [Get+13]. The blueprint has an *occurrence* of 8 and 5 *shared parameters* (we do not count the shared *Root* element). Our algorithm did not find any more *diverse* blueprints in this data set.

In [KGT16] we performed an evaluation of a prototype recommender as a proof of concept that did not yet use blueprints. This early prototype created recommendations based on a single user change in version V_x and all changes from earlier versions $V_{x-1}..V_0$. If all recommended changes based on the history up to V_{x-1} were present in version V_x, then we counted this as a true positive, else it was a false positive. Changes in version V_x that were never recommended were counted as false negatives.

For the evaluation, we used the GMF data set from Herrmannsdoerfer et al. [HRW09] because it contained models with many versions, and it was also used by other researchers. The data set consists of three models with 10–110

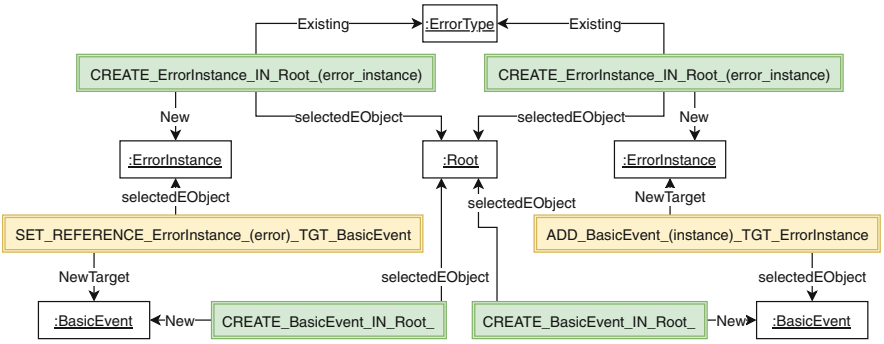


Fig. 10.17 Blueprint learned by our genetic algorithm from the PPU fault tree data set

Table 10.2
Recommendation results for different models

Name	TP	FP	Precision
gmfgen	554	217	0.72
gmfgraph	25	33	0.43
Mappings	36	8	0.82
Total	615	258	–
Average	205	86	–

Table 10.3
Recommendation results for PPU fault trees

Name	True positives	False positives	Precision
faulttree1	95	115	0.45
faulttree2	71	36	0.66

versions, and our prototype achieved a precision between 43% and 82%. This shows that it is possible to generate recommendations from model histories. The results are depicted in Table 10.2.

We also performed an evaluation on two fault trees created from the PPU evolution scenarios. These results are depicted in Table 10.3. We will also perform a further evaluation on the Petri Nets learned in Sect. 10.4.2.

Related Work

There already exist recommender systems for model-driven software development that are capable of performing the refactoring from Fig. 10.15a. Kuschke et al. [KMR13] translate common refactorings on models into constraint systems and use these to identify partially applied refactorings. They use a constraint solver to generate a recommendation based on a partial refactoring. Another approach from Taentzer et al. [Tae+17] focuses on model repair. They search for partially applied refactorings in models and recommend the remaining parts of the refactoring. This approach also works for inter-model refactorings.

Both of these approaches require expert knowledge about how common refactorings for certain types of models look like. But this knowledge is not generally available for all types of domain-specific modelling languages.

Note that our work is also related to model transformation by example (e.g. [Var06]). But this approach also requires an expert to generate clear examples of transformations for every type of model (i.e. pairs of models before and after the transformation).

Another approach by Sen et al. [SBV10] uses meta-models as graph grammars, these grammars are then used to compute possible additions to instances of these meta-models, so that the computed instances are valid w.r.t. the meta-models.

This approach can be applied to an instance model when its meta-model is available and its results are similar auto completions for text-based programming languages. Our approach, in contrast, aims to recommend refactorings that are more complex than auto completions.

Other related works are source code recommender systems [MKM13, BMM09, Muş+12]; these make use of different techniques for generating and prioritising recommendations based on previous changes to source code. All major text-based programming languages can be defined via context-free grammars (e.g. EBNF), which simplifies the implementation of these source code recommenders. Also, there are only few very widespread text-based programming languages, which makes it feasible to develop or extend these recommenders manually. Thus, this work is related, but the techniques cannot be directly applied to the more diverse and complex realm of domain-specific models.

Conclusion

Existing recommenders for model-driven software development are mostly model-type specific and require expert knowledge. Our vision was to automatically generate recommendations based on model change histories. For this we inferred blueprints of recommendations from model change histories. This work has also been presented in [Kög17].

Our blueprints and our methods to generate these blueprints will be useful for developing model agnostic recommender systems.

10.4.2 Recommending Evolution Steps Within a Knowledge Carrying Network

Contrary to the previous approach of recommending changes of one model, the following method targets horizontally integrated systems. Cyber Physical Production Systems (CPPS) are characterised by the interconnection of software and hardware components that are horizontally and vertically integrated. These CPPS cause the need to document their evolution systematically to support their maintainability

over time [Vog+15a]. To handle this task, a collective memory for identifying and perceiving historical artefacts is needed to allow collaborative work in which not only the machine's behaviour arising from the dynamics of its hardware and software components is considered but also the evolution of the whole CPPS is considered holistically [GG08]. Here, the horizontal integration of machines provides new potentials to support evolution [Hau+17] because it allows for a comparison of one individual machine status among other machines by pushing information about their status in the networked cyber level of a CPPS [LBK14]. These statuses can be seen as steps of the evolution performed during their semi-automated operation. The recommendation of evolution has a strong focus on these steps, which goes along with the core idea of evolution that focuses the process of changes rather than how a machine looks like after the change. Therefore, in the following section, model-based evolution steps are showcased as first-class entities that form an evolution process in a peer-to-peer network. Recommendations are given to engineers on the basis of this evolution process by providing evolution steps of other similar systems that were already extracted during the evolution process. In this way, the vision of a marketplace for evolution steps is followed that allows exchange of already performed evolution steps [Hau+18c].

Express Evolution Steps of a System

To describe evolution within a horizontally integrated network, evolution should be fully expressed by the sequential execution of evolution steps. For the description, a step-based approach similar to the delta-modelling of software product lines is used. The idea of steps is to adapt to evolution by representing an evolution in a core step, which is the initial development, and apply evolution steps over time. Evolution steps represent changes that are following Buckley et al. [Buc+05] characterised in the form of questions. Evolution steps are described as answers to the six questions what, why, who, when, where, and how.

To understand the learned behaviour as evolution steps, it must be specified *what* is changed. In order to capture historical behaviour, runtime models are learned according to the signal behaviour of a machine, as presented in Sect. 10.3.1. The following approach is therefore explained and demonstrated on material flow models of production systems. These models are Petri Nets, which have signals as transitions. What is changed is expressed in accordance with these signals in lifted (edit) operations. To utilise the differencing pipeline of lifted (edit) operations, as explained in Sect. 10.1.1, a matching strategy of transitions and therefore signals is needed. The used matching strategy measures the similarity of signals by exploiting their additional context information. This is done in order to identify similar or even the same production sections, which are reflected in similar patterns of the signals and their context information (timing, type, and topology). As a similarity metric, the algorithm uses a Null Hypothesis for the timing according to a specified confidence and a relation matrix for the type and topology context. To find the matching, strongly related signals are identified by aligning different sequences of

signals to each other. The best alignment is identified by an extended Needleman-Wunsch Algorithm, and the signals are matched following this alignment. Further details about the matching can be found in [Cha+18].

To identify the lifted edit operations, a template for the material flow production system is used. It determines how the structure of the signals evolves in the material flow. For structure changes, two (edit) operations apply when an additional signal occurs before or after a specific signal sequence and a third if two separated signal sequences in the model merge or split. Besides structural changes, changes in the meaning of signals are captured by operations for timing changes, as well as signal type and topology changes. For example, these changes in timing occur when the latency of the signal is changed, for example due to wear and tear. Based on these operations, edit scripts are derived (see Sect. 10.1.1) that represent what is changed in an evolution step. More details are given in [Pie+18].

Further, it is questioned in an evolution step *where* the evolution takes places. This describes the condition under which the evolution happened and therefore where the operations of signals are applied. Since in the practical implementation these conditions are sensitive parameters, three *privacy policies* are introduced: First, just the edit script with no additional information besides the changed signals is provided in the step. Second, neighbour information of signals directly related to the changed signals of the edit script are included in the evolution step, and, last, the entire structure of the material flow model from which the edit script stems is included.

Why a system evolves is evaluated in terms of non-functional properties such as performance or flexibility since it enables the operator to recognise weaknesses and to ensure the quality of complex systems [LFL16]. This part relies on the non-functional properties derived directly out of the material flow models as the reason for evolution. Properties express the change in properties that was observed when the step was introduced. The extraction of these properties was introduced in Sect. 10.3.1.

The temporal properties (*when*) build up an evolution process by linking to each other, which creates a partial order in the form of a directed tree. Vertexes of this tree represent steps, and edges represent the directions in which the evolution proceeded with respect to versions and variants. Versions describe an evolution step in the same system and variants a step derived by applying a step on another system. The leaves of the tree are the currently operated versions, which can be reconstructed using involved steps retrospectively starting from the tree's root. In this way, an evolution process of the whole network of systems is built.

One interesting question, when seeing evolution in exchangeable evolution steps of a network of systems, is *who* has evolved. This ownership is reflected in an evolution step by using a digital signature of the evolved system. For evolution steps, a valid digital signature gives assurance that a step really was provided by the signing system. In addition, this allows tracing back steps for evolution support.

How evolution is supported is considered in the environment of the network. From a technical point of view, the overall evolution process is a distributed ledger of evolution steps that is stored within a distributed network. Therefore, to support

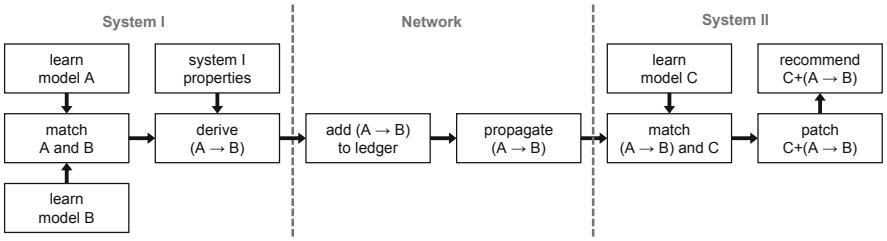


Fig. 10.18 Process of the extraction of an evolution step and its comparison to a model of another system

evolution, the steps must be propagated through the network. Figure 10.18 shows the evolution support process that provides recommendations to the engineer. The process assumes two different systems that observe and analyse models, as well as detect anomalies of the production system to extract different versions of models. This is performed by the knowledge-carrying software of the underlying learning approach that uses a model-based CPS architecture (see [Hau+18b] for details of the architecture). Whenever a change is detected, the previous model (model A) is matched with the model after the change (model B) to derive the evolution step ($A \rightarrow B$). To share the step, the network adds the step to the ledger and then propagates the step to all other nodes. Propagated evolution steps are used to recommend changes to the engineer. The shared step is matched and then patched to their own model (model C) with the help of model patching (see Sect. 10.1.1). The patched model is provided as a recommendation to the engineer to propose a behaviour change that was already positively experienced on other production systems.

To sum up, this section suggested evolution steps that use a model-based edit script (what) under a specific context condition (where) to achieve the desired effect (why). Then the evolution steps are structured by linking predecessor and inheritance as a process (when) that considers its originators (who) in a propagating network that gives recommendation to the engineer (how supported).

Application on xPPU Case Study

To evaluate the concept, a marketplace for evolution steps is envisioned (see [Hau+17]). This distributed marketplace should allow a platform to provide and request evolution steps in order to share experiences about evolution in the form of performed changes. The evolution steps were learned for Scenarios 1, 1b,[2] 2, 3, 5, and 10 of the xPPU case study. The distributed ledger of evolution steps is

[2] 1b is a modified version of the Pick-and-Place Unit (PPU) Scenario 1, which only uses metallic workpieces.

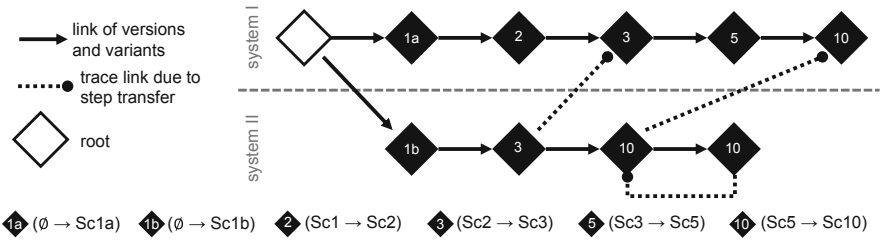


Fig. 10.19 Evolution process of performed steps over CPPS network

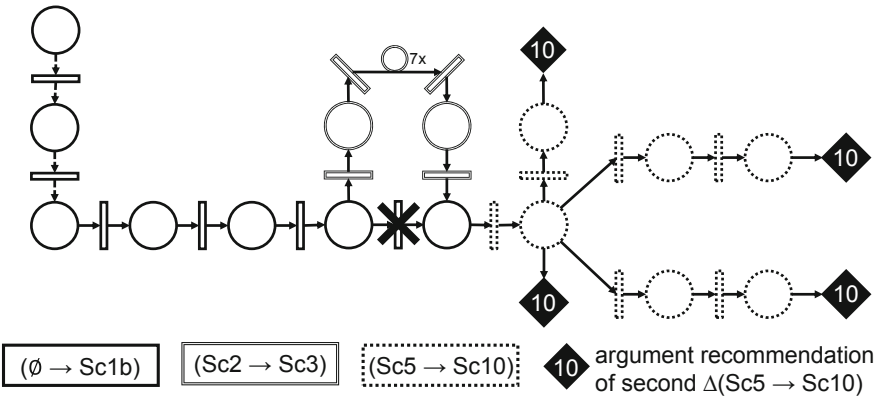


Fig. 10.20 Summary recommendation of the resulting material flows (refer: [Hau+18a])

implemented in a knowledge-carrying software managing material flow models at runtime, which includes a peer-to-peer network with blockchain functionalities to implement a distributed marketplace. The platform provides a common and secure ledger while eliminating the central authority with its peer-to-peer nature. This is done by a global consensus protocol that ensures a full copy of the blockchain available for each participant that contains every step of the build-up evolution process. The tested peer-to-peer network had one KCS for each evolution scenario of the case study and consisted of 6 (simulated) production systems.

Exemplary, Fig. 10.19 shows the resulting evolution process of two systems. It shows system I, which already underwent all scenarios from 1 to 10 with all possible evolution steps, and system II, which started as Scenario 1b with only metallic workpieces. As the figure illustrates, system II evolved through recommendation in three cases by applying the stamp ($Sc2 \rightarrow Sc3$) and two conveyors ($Sc5 \rightarrow Sc10$) without additional workpieces of Scenario 2 or the crane optimisation of Scenario 5. Furthermore, Fig. 10.20 shows a summary of the resulting recommendations given to the engineer before the last addition of a conveyor. The left-hand side of the model represents the root model with the initial step implemented on it ($\phi \rightarrow Sc1b$). It transports plastic workpieces to a ramp. The further steps are explained in details

in [Hau+18a]. It can be seen that recommendations on the model layer can have errors. The recommendation contains a transition that will never be observed in the implemented system because it indicates that a workpiece is directly transferred from the stack to the ramp. This will not be observed in the implemented system because no plastic workpieces exist in the evolved system II and therefore no direct transport is done in system II. But the patched model still provides a good recommendation, and the mentioned transition is not included in the model after the operator implemented the signal behaviour. The last patching is not reflected in Fig. 10.20. Instead, it shows alternatives identified by the matching algorithm. An additional conveyor could be added instead of one of the output ramps served by the pushers (two alternatives at the right) or instead of the ramp at the end of the existing conveyor (alternative at the top), as well as in parallel to the existing conveyor (alternative at the bottom). Each option is valid (which is not necessarily the case). This example shows the variety of recommendations that might be provided by this approach.

Related Work

In cloud and agile manufacturing, similar approaches exist but aim at virtualisation of manufacturing capabilities and resource sharing [VH13]. As an example, the approach by Lee et al. [LBK14] suggested to send information to a centralised hub for special analytics. The presented evolution support approach can be seen as an instantiation of CPPS with cloud connection for the specific functionality of a cooperative evolution support but aims for decentralised capturing and sharing of model differences. Cooperative knowledge exchange about specific topics has already proven its benefits in other application domains like distributed data race detection [KZC12]. Therefore, the approach is also related to experience management in which approaches exist for capturing the users' or developers' field experiences of software systems [He13] or systems driven by user experiences [SVS14]. Here the approach follows the view of knowledge-sharing networks that combine the idea of (a) an advanced preparation of an individual itself and (b) an evolution steps by considering the knowledge of other individuals who already made these experiences.

Direct sharing of runtime evolution steps of machines is not the focus of the research so far, but, for example, Würsch et al. [WGG13] present a query network for software evolution data in a software development tool, which shows that sharing and exchanging can answer evolution questions. In software development, past changes are often captured in software variant and versioning tools. Such a repository provides data to find changes and is in this approach transferred to CPPS in the operational phase with evolution based on generated models. Furthermore, Kolovos et al. [KPP06] have laid out different ways of implementing model transformation tools of Eclipse Modelling Framework and Graphical Modelling Framework to optimise results in order to make them beneficial in terms of productivity and maintainability. Abdallah et al. [ATW08] already have demonstrated the

applicability of model transformation aspects through different case studies. Brun and Pierantonio [BP08] have demonstrated how maintaining a database and tracking of design-level changes have made it possible to understand structural evolution processes. They have illustrated how to find differences and how to track them. They give reasons on how important it is to record differences between instances consisting of calculation and identification of such differential steps. In this section, we targeted a similar automated history of (quality) indicators on a model level in the operational phase for recommending behaviour changes.

Further Reading

More information about the model learning approach can be found in Sect. 10.3.1. Additional information about the matching used here has been published in [Cha+18] and the overall idea of a cooperative support in [Hau+17]. In [Pie+18], the use of the differencing approach for learned behaviour models of production systems is shown in more detail, and in [Hau+18c], the idea of a marketplace for evolution experience is described.

10.4.3 Learning Maintainability Estimation for Enabling an "Economical Effort Recommender"

In contrast to previously presented model learning approaches to automated identification of changes, this section deals with the automated analysis of change propagation in system models. For this purpose, we extend a change impact analysis approach in Information Systems (IS) to aPS, as IS and aPS face very similar challenges with respect to evolution. One target of the analysis of an IS or aPS with respect to its maintainability is giving an impression of the change effort initiated by a change request. Rather than identifying which changes were performed from a system alternative to another one, maintenance tasks are derived from a certain change request and for a given architecture. These maintenance tasks can be used to identify the effort for changing the system architecture, realising an "Economical Effort Recommender". In order to realise a change impact estimation approach for both IS and aPS, an architecture description in terms of meta-modelling and an architecture-based change impact identification for automating this procedure are necessary.

Challenges for Maintainability Estimation: Information Systems vs. Automated Production Systems

Both IS and aPS operate over time, often over decades, after deployment. During the runtime, they often face change requests that lead to the modifications on the system

for most of the cases, and these modifications may include corrections, improvements, or adaptations of the system environment changes [Hei+18a, Vog+17a]. Therefore, maintainability, which is defined as the ease with which a system or a component can be retained in a state where it can perform its required functions [IEE90], is an important quality aspect of the system, especially for long-living systems.

However, maintainability estimation is already a difficult task for IS [Leh80] and even harder for aPS since they are comprised of multi-discipline artefacts from mechanical, electric/electronic, and software engineering. Moreover, aPS is implemented in modularised architectures that are in their own way different from pure software systems [Vog+17b]. In the following, we discuss the state of the art for the change impact analysis in IS and aPS.

State of the Art for Change Impact Identification in Information Systems In the domain of IS, there are several approaches to change impact identification [Ros+15]. Task-based project planning focuses on building a hierarchical decomposition of tasks into sub-tasks (e.g. [KA03] or [Car+83]). However, these approaches do not use the software's architecture. Further, the scale of the estimation is coarse grained. Architecture-based project planning approaches (e.g. [PB01] or [Car12]) are based on Conway's law [Con68] stating that the communication structure of organisations plays an important role in software design. However, the approaches in this category do not estimate the change effort or do not support an automated change impact analysis. A further category is architecture-based software evolution, for example as done in [Gar+09] and [Naa12]. They consider software architecture as the main artefact during the change process. However, the works are not extended to the change effort estimation. In scenario-based architecture analysis, supplementary information is taken into account, such as maintenance scenarios [BB99] or informal software architecture description [Cle+02]. Still management tasks are missing in the aforementioned approaches.

State of the Art for Change Impact Identification in Automated Production Systems Effort estimation in automation has been purely realised by counting the results of input and output signals multiplied by the hourly effort per signal for decades [Vog14]. However, this method provides just simple results, not all the necessary tasks in detail and corresponding costs. Recently, Prähofer et al. proposed a multi-purpose feature-modelling approach by mapping feature into modules [Pra+17], and this enables the impact measurement of a change, but this is not embracing the various type of features, that is all the features from different disciplines. This approach is based on the aPS meta-models. Meta-modelling is an important topic for this project since an aPS meta-model and its properties, which lie over the disciplines, are required ultimately. For the multi-disciplinary property of aPS, domain-specific models from each domain may be maintained separately [Bro+10]. The problem is synchronising all models [Fel+15], especially when a change happens in one domain. Especially, some works (e.g. [KV13] and [FKV14]) suggest discipline-encompassing modelling methods based on SysML to check compatibilities on changes in the interface level and functionality level.

Further, representing all the necessary information within one meta-model has also been researched by [BFS05]. This approach suggests a method to combine software structure, together with physical aspects in one common model. The approach of Berardinelli et al. [Ber+16] maps AutomationML into SysML. Recently, an integrated plant modelling using AutomationML is introduced in [Win+17]. However, models or meta-models focus on specific problems respectively, and there is no existing meta-model that encompasses all different disciplines, especially with the purpose of estimating change effort by examining change propagation.

Karlsruhe Architectural Maintainability Prediction for Automated Production Systems: An Architecture-Based Change Impact Analysis Method for aPS

In the DoMain project, we proposed a systematic and automated approach for effort estimation considering all disciplines within aPS instead of manual estimation, depending on the engineers' knowledge. As IS and aPS are very similar regarding evolution, we used an existing change propagation approach from IS—Karlsruhe Architectural Maintainability Prediction (Karlsruhe Architectural Maintainability Prediction (KAMP)) [Ros+15]. KAMP aims to analyse the change propagation in the model of the software architecture for a set of change requests. For this purpose KAMP uses change propagation rules based on domain knowledge. Based on the KAMP [Ros+15] and KAMP framework [HBK18a], we developed Karlsruhe Architectural Maintainability Prediction for automated Production Systems (KAMP4aPS) [Hei+18a, Vog+17a, Koc17]. KAMP4aPS allows automated change propagation analysis starting from an initial change in the system. To solve the issues discussed in the beginning of this subsection and to allow a more realistic change effort estimation, our approach considers technical and organisational tasks during the change process as well. To support change propagation in the system's structure, as well as in the technical and organisational tasks, we provide two meta-models, which serve as input for KAMP4aPS. Using the first meta-model, the system's structure can be described. The second meta-model describes the non-structural elements in the system for the structural elements. Examples of non-structural elements are documentation and test cases, as well as technical and organisational information. Thus, non-structural meta classes reference the corresponding structural meta classes. In other words, the input of KAMP4aPS is the descriptive model of the system structure and the description of non-structural elements for the structural elements. Based on the input and the initial change in the descriptive model, KAMP4aPS automatically generates a task list containing all tasks that should be potentially carried out to implement the change. These tasks refer to changing elements in the descriptive model of the system structure and of the non-structural elements. As effort estimation may differ according to different elements (e.g. the cost of changing a specific component, module, or program), there is a need for domain knowledge to accurately estimate the effort. Thus, effort estimation has to be done manually based on the generated task list.

Further, due to the diversity of artefacts in aPS domain due to electrical and mechanical parts, as well as software systems, some artefacts may be neglected when estimating the effort of a change implementation manually. As KAMP4aPS considers these artefacts during evolution (i.e. using the system structure model and the model of the non-structural elements), the cost of a change can be estimated more realistically. Another benefit is the way in which a change scenario can be implemented, as there are usually different ways to implement a change scenario. Using KAMP4aPS, different ways of implementation at model level can be simulated and compared with each other without implementing them in the real system [Hei+18a, Vog+17a, Koc17, Ros+15]. Thus, KAMP4aPS serves as an economical effort recommender as it allows selecting the most efficient and cost-effective implementation variant for a change scenario and estimating the change effort before implementing the change.

Overview of KAMP4aPS Approach Figure 10.21 gives an overview of KAMP4aPS. It is composed of two phases. In the first phase, the input has to be prepared. As illustrated in Fig. 10.21 and described previously, the first phase comprises (i) constructing structural models of the domain; (ii) annotating these models with further information regarding the non-structural elements, as well as the technical and organisational elements; and (iii) identifying the initial changes in the model based on a change scenario. In the second phase, KAMP4aPS automatically calculates the change propagation using a set of predefined change propagation rules. Change propagation rules are defined by the domain expert and describe from which meta class to which meta class of the system structure meta-model the change can propagate (e.g. change propagation from the sensor to its fixation). Based on the initial change, KAMP4aPS iteratively applies the change propagation rules on the system structure meta-model and identifies the affected model elements by the change. In other words, KAMP4aPS applies the change propagation rules to the model elements identified as affected in the previous iteration. In each iteration, KAMP4aPS adds the newly affected model elements of the system structure meta-

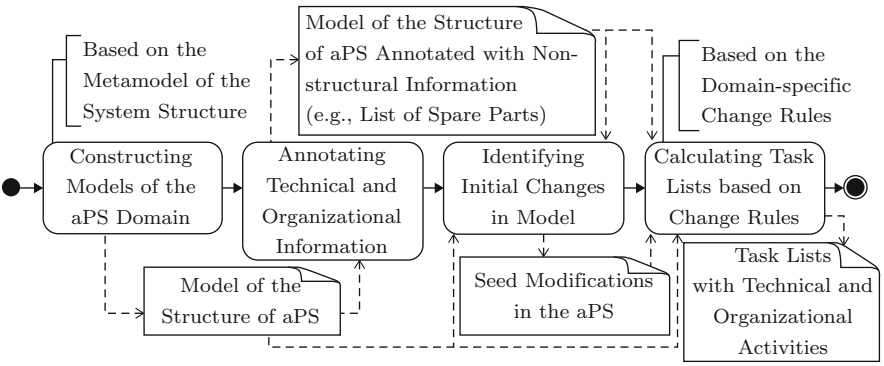


Fig. 10.21 Overview of KAMP4aPS task list derivation procedure [Hei+18a, Vog+17a]

model to a set containing all affected artefacts. The set of all affected artefacts does not contain any duplicates. Thus, KAMP4aPS terminates if no new model elements are added to the set of all affected artefacts in the last iteration. For each affected model elements of the system structure meta-model, KAMP4aPS identifies all affected model elements of the non-structural meta-model. For example, if there are test cases (i.e. model elements of the non-structural meta-model) for a component (i.e. model element of the structural meta-model) and the component is affected by a change, the test cases have also to be marked as changed. KAMP4aPS adds the affected model elements of the system structure meta-model and non-structural meta-model to a task list. The task list is the output of our approach. Each task in the task list refers to the affected elements of the system structure meta-model (e.g. changing a component), as well as technical (e.g. changing the corresponding test cases) and organisational activities (e.g. updating the list of spare parts) [Hei+18a, Ros+15, Vog+17a, Koc17].

Motivation Example: xPPU As an example, a certain component of the aPS might need to be replaced (e.g. the replacement of the microswitches within the crane module of xPPU, as in Scenario 13 [Vog+14b]). The crane module has used three microswitches to indicate the direction of the arm at stack, at stamp, and at conveyor, respectively (see the structure in Fig. 10.22a). With some reasons, for example contamination or defect of the microswitch, the accuracy is degraded and needs to be improved. For example, the crane is at the ramp position, as depicted in

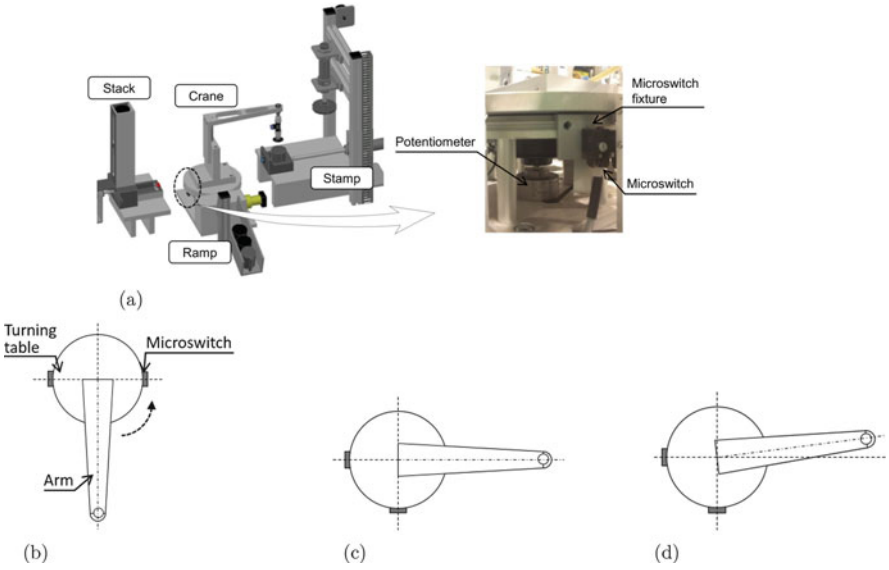


Fig. 10.22 Crane structure description and defective crane positioning. (**a**) Graphical representation and actual image of crane structure. (**b**) Crane at the ramp. (**c**) At the stamp (intent). (**d**) At the stamp (actuality) [Hei+18a]

Fig. 10.22b, and is supposed to turn to the stamp position, as depicted in Fig. 10.22c. However, the crane passes by the exact stamp position and stops at a wrong position, as illustrated in Fig. 10.22d. This performance issue leads the replacement of the microswitches with rotary potentiometer. In other words, to achieve a more accurate motion of the crane arm rotation, the engineer decides to replace all these three microswitches with one potentiometer, which provides more precise information about the rotation angle. Even with this simple example, through counting I/O, which is the conventional and common approach, this change effort can just be counted as removing three digital inputs and adding one analogue input. However, more detailed tasks (i.e. not just about the number of input/output port changes but rather about specific tasks on the relevant components) need to be done for this implementation from purchasing needed elements (i.e. potentiometer itself and additional fixture parts) to updating corresponding documents (e.g. operation instructions and stock lists) in addition to replacing the elements [Vog+14b].

As discussed previously, a change in a system propagates not only to the elements in the system structure but also to the non-structural elements (Fig. 10.23). Figure 10.24 shows an exemplary task list following the change for Scenario 13 [Vog+14b]. Tasks include not only the change activities for implementing the change but also the related activities, for example maintaining the artefacts, as seen in Fig. 10.24. Additionally, each type of the task is mapped into the specific

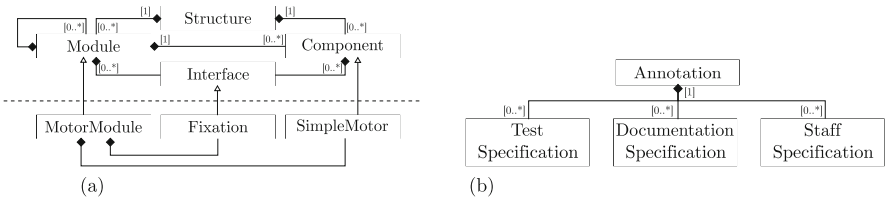


Fig. 10.23 aPS meta-models. (**a**) Structural aPS meta-model. (**b**) Non-structural aPS annotation meta-model [Hei+18a]

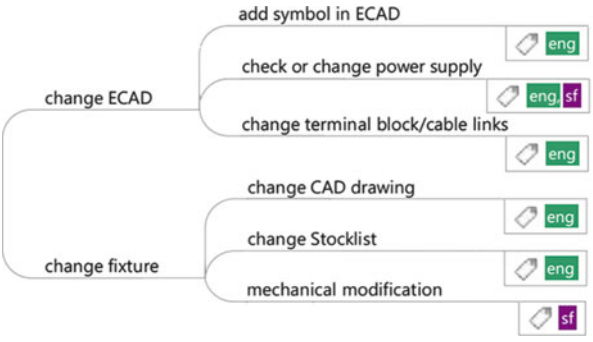


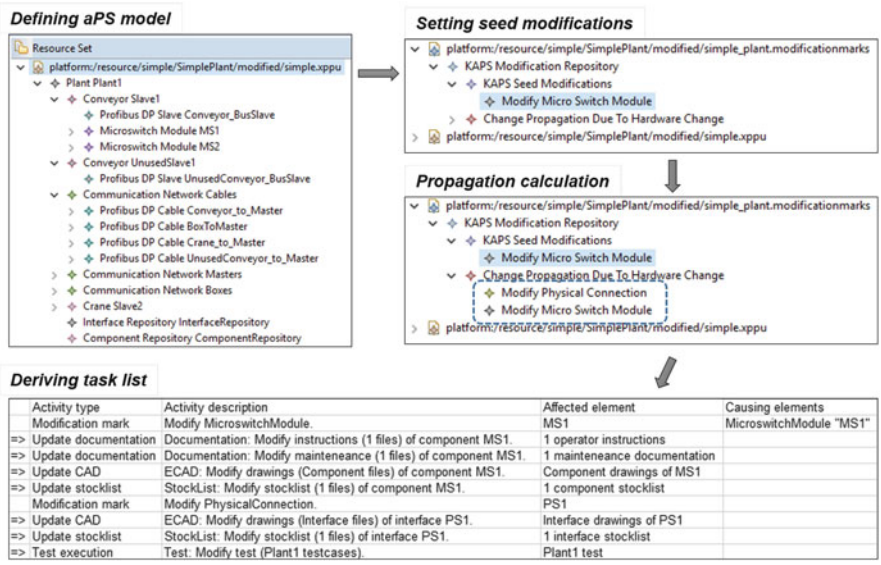
Fig. 10.24 Depiction of the task list on Scenario 13

personnel or department (as described using eng for engineering staff, m&p for material and procurement, and sf for shop-floor). Thus, the cost of the tasks can be calculated, for example, based on person-hour cost. Further, we can observe that the change causes tasks in other disciplines: removal of microswitch (i.e. an electrical component) causes mechanical modification. These mutual dependencies between disciplines are handled by a discipline-integrated meta-model. This meta-model specifies structural element descriptions of the system. The non-structural elements can be presented using a further meta-model. Each meta-class of this meta-model refers to the corresponding meta-classes of the structural elements. Figure 10.23 shows an excerpt from the structural (a) and the non-structural (b) meta-model [Hei+18a].

Application of KAMP4aPS to xPPU In order to use KAMP4aPS, the structural model of the plant has to be provided by the domain expert. A further model contains the non-structural elements of the plant, such as test cases, documentation, or ECAD. In this scenario, the seed modification is changing the three microswitches. Starting from seed modification, KAMP4aPS iteratively calculates other changing elements based on the change propagation rules. The change propagation rules highly depend on the underlying meta-model of structural elements. The domain expert can define the change propagation rules at a high abstraction level for a general plant (e.g. a change can propagate from an affected component to other components) or at a low abstraction level for a concrete plant (e.g. a change can propagate from an affected sensor to its fixture). For KAMP4aPS, we defined a set of generic change propagation rules, as well as a set of specific change propagation rules. The general change propagation rules can be extended to more concrete rules for a specific plant. If a change propagation rule specifies the change propagation from microswitch to its fixture, KAMP4aPS marks the corresponding fixture as modified. Further, it derives required task lists considering non-structural information based on predefined change propagation rules. In this way, the change propagates through the elements of the plant. Figure 10.25 describes the results of the KAMP4aPS tool regarding the change scenario introduced previously. Simplified version of the xPPU model is inserted, and the seed modification is defined as modification of microswitches. Ultimately, these tasks can be manually converted into cost. Based on this cost estimation information, better implementation, which means less costly solution, can be recommended [Hei+18a].

10.4.4 Conclusion and Outlook

This section has explored possibilities of how past changes can be used to recommend and assess future changes. Therefore, three possibilities are presented that allow a more guided evolution of software-intense systems. First, it was shown how information that is required for making useful recommendations can be inferred from the change histories of models. This information was used to



	Activity type	Activity description	Affected element	Causing elements
	Modification mark	Modify MicroswitchModule.	MS1	MicroswitchModule "MS1"
=>	Update documentation	Documentation: Modify instructions (1 files) of component MS1.	1 operator instructions	
=>	Update documentation	Documentation: Modify mainteneance (1 files) of component MS1.	1 mainteneance documentation	
=>	Update CAD	ECAD: Modify drawings (Component files) of component MS1.	Component drawings of MS1	
=>	Update stocklist	StockList: Modify stocklist (1 files) of component MS1.	1 component stocklist	
	Modification mark	Modify PhysicalConnection.	PS1	
=>	Update CAD	ECAD: Modify drawings (Interface files) of interface PS1.	Interface drawings of PS1	
=>	Update stocklist	StockList: Modify stocklist (1 files) of interface PS1.	1 interface stocklist	
=>	Test execution	Test: Modify test (Plant1 testcases).	Plant1 test	

Fig. 10.25 Prototypical KAMP4aPS running results on microswitch replacement in xPPU

provide blueprints for recommendations that represent common refactoring or frequent changes to a type of model. A second approach considers how change information that is generated out of past changes can be used to establish a knowledge-carrying network for evolution experiences by providing changes as evolution steps in a service-component architecture. These evolution steps are used in similar production systems to identify potential improvements of the system and provide recommendation in the form of high-level model description of the change and its impact. The last approach focuses on presenting tasks to the operator of a production system. Therefore, a systematic and automatic change impact analysis approach and an interdisciplinary meta-model for production systems were shown. The approach allows achieving the task list based on the architectural model considering the multiple disciplines. The task list can be used for manual effort estimation by mapping the tasks to the costs instead of coarse-grained manual estimation depending on the engineers' knowledge.

As future work, it is intended to integrate the three presented approaches. High-level changes of the monitored signal models in the knowledge-carrying network should be learned with the automatic recommendation approach for model editing. Further, the integration of both case studies establishes the economical effort recommender in a supply chain of Common Component Modeling Example (CoCoME) and xPPU so that possible tasks based on KAMP4aPS using the interdisciplinary model and exchanged evolution steps containing the learned high-level changes can be provided.

10.5 Conclusion

Software is ever changing. Hence, software evolution is a continuous process that requires knowledge about the software. The existence and quality of this knowledge have a high impact on the success of software evolutions. We presented in this chapter several approaches to learn from past software evolutions and, thus, guide future evolutions. Several approaches have been developed around the joint Pick-and-Place case study.

Analysing and understanding past evolutions is a foundation for the evolution of a software. The approaches in Sect. 10.1 support systematically analysing how single models have evolved in the past and multiple ones have co-evolved.

The approaches in Sects. 10.2 and 10.3 address functional and non-functional requirements during software evolution. Whereas the former approaches focus on functional correctness of the evolved software, the latter approaches provide learning of models from running systems in order to analyse non-functional requirements.

Finally, the approaches in Sect. 10.4 support software engineering during a software evolution by recommending future evolutions from previous software evolutions either of the same system or from other similar systems.

While all those approaches solve their respective challenges and particularly the analysis of past evolutions is used in multiple approaches, future work includes a tighter integration to holistically address software evolution.