Supporting Semi-Automatic Co-Evolution of Architecture and Fault Tree Models

Sinem Getir[a], Lars Grunske[a], André van Hoorn[b], Timo Kehrer[c], Yannic Noller[a], Matthias Tichy[d]

[a] *Software Engineering, Humboldt-University of Berlin*
email: {getir,grunske,noller}@informatik.hu-berlin.de
[b] *Institute of Software Technology, University of Stuttgart*
email: van.hoorn@informatik.uni-stuttgart.de
[c] *Model-driven Software Development, Humboldt-University of Berlin*
email: timo.kehrer@informatik.hu-berlin.de
[d] *Institute of Software Engineering and Programming Languages, University of Ulm*
email: matthias.tichy@uni-ulm.de

Abstract

During the whole life-cycle of software-intensive systems in safety-critical domains, system models must consistently co-evolve with quality evaluation models like fault trees. However, performing these co-evolution steps is a cumbersome and often manual task. To understand this problem in detail, we have analyzed the evolution and mined common changes of architecture and fault tree models for a set of evolution scenarios of a part of a factory automation system called Pick and Place Unit. On the other hand, we designed a set of intra- and inter-model transformation rules which fully cover the evolution scenarios of the case study and which offer the potential to semi-automate the co-evolution process. In particular, we validated these rules with respect to completeness and evaluated them by a comparison to typical visual editor operations. Our results show a significant reduction of the amount of required user interactions in order to realize the co-evolution.

Keywords: system architecture, fault trees, safety, model co-evolution, model transformation

1. Introduction

Quality of service (QoS) attributes such as safety, reliability and performance are crucial for software-intensive systems, e.g., in safety-critical or

automation systems. Such systems (e.g. aircraft and automation systems, robotics) are not tolarable with an errogoneus design since they are playing fundamental roles in human lives. Therefore quality evaluation process during the development of systems from design time to runtime is inevitable. A rigorous quality evaluation is among the key methods for the dependable engineering of such systems. To that end, model-based approaches have been proposed which construct quality evaluation models from system models to gain knowledge about the quality of a system by checking these models against formally specified quality requirements [37].

In model-based quality evaluation, the consistency of the involved models is of utmost importance. For example, the failures of an architectural component must be adequately considered in an associated fault tree model. While this consistency requirement can be reasonably met for a particular snapshot of a system, quality evaluation models typically become outdated when the system evolves, i.e., quality evaluation models and system models evolve in an inconsistent way. As a consequence, quality evaluation leads to unexpected and highly improper results. An example in the context of hazard analysis of component-based embedded systems is the addition of a new port for a sensor of a component without a corresponding addition of the sensor failures in the relevant fault trees. This will clearly lead to wrong hazard analysis results. Hence, loosely inter-related models such as architectural models and quality evaluation models should consistently evolve in parallel, a phenomenon to which we refer to as (consistent) *model co-evolution* in the remainder of this paper.

Since loosely inter-related models are typically changed in isolation of each other, one adequate approach to support developers is *model synchronization*, i.e., the task of adapting a model in response to changes in one of its inter-related counterparts in order to achieve consistent co-evolution. In general, however, achieving this kind of model co-evolution is not a straightforward task [61]: quality evaluation models cannot be fully generated from system models, and most relations between the elements of the different models are not simple one-to-one correspondences. In other words, achieving consistent co-evolution cannot be fully automated as usually assumed by existing approaches to model synchronization (see Section 2 for a more detailed discussion of related work in this area). At best, developers may be supported by *recommending* possible synchronization actions, e.g. as in the model-based (co-)evolution framework known as CoWolf [29]. To achieve consistent co-evolution, CoWolf follows a rule-based approach where incre-

mental model transformations are used to recommend both intra- and inter-model change actions. However, since the adequacy of the recommendations strongly depends on the transformation rules being used by the tool, the evolution problem is shifted to the engineering of proper transformation rules. These should capture evolution and co-evolution steps being considered as useful by the developers using the tool.

In this paper, we tackle this problem of engineering proper transformation rule sets for an important class of models in the context of model-based hazard analysis of software-intensive systems, namely system architecture models and fault tree models. We extend our previous work [30] on the evolution of the so-called Pick and Place Unit (PPU) [58], a case study from the automation engineering domain which is commonly used in the German priority program "Design for Future – Managed Software Evolution" [33]. To study co-evolution in terms of the PPU, we created consistent software architecture and fault tree models for all safety relevant evolution scenarios. The contributions of this paper are:

1. A thorough quantitative analysis of the evolution scenarios with respect to the co-evolution of the models, i.e., how changes in one model affect changes in the other model. We show that the models do not co-evolve in a systematic, automatable way and instead expertise of the developer is required to achieve co-evolution. This is a minor contribution which, while not being exceedingly surprising, confirms the findings of previous research in this context.
2. The major contribution is a set of model transformation rules for 1) the independent evolution of software architecture and fault tree models and 2) synchronization of one model based on changes in another model ensuring a correct co-evolution of both models. In the evaluation of the rules, we show that the presented set of model transformations is *complete*, i.e., it supports performing all co-evolutions of the case study scenarios, and improves the *task efficiency* (cf. Quality in Use [1]) by reducing the amount of required model transformation applications to realize the co-evolution by, on average, 52% compared to visual editing operations and 85% compared to atomic model changes. Additionally, we implemented these rules in the tool CoWolf [29] to enable the co-evolution of fault trees and software architecture models.

To enable reproducibility of our results, we make all models for the scenarios, the set of model transformation rules as well as the code for the

evaluation publicly available.[1]

The remainder of this paper is structured as follows. In the next section, we discuss related work in the areas of safety evaluation models and their automatic generation as well as different approaches to achieve consistent co-evolution of inter-related models. Section 3 briefly sketches the used modeling and model transformation languages. Thereafter, we present the results of the quantitative analysis of the co-evolution of the models of the case study in Section 4. Section 5 contains a description of the developed inter- and intra-model transformation rules for architecture and fault tree models. We evaluate this set of model transformation rules in Section 6. Finally, we conclude and present an outlook on planned future work in incremental analysis of quality models and supporting the developer selecting model transformations by analyzing historic developer decisions on co-evolutions. The appendix contains a thorough description of all evolution scenarios and their impact on system architecture and fault tree models.

2. Related Work

The work presented in this paper draws from two separate research areas, namely the work on automatic/semi-automatic generation of safety evaluation models in the architectural design phase and general approaches that aim at keeping dependent models consistent while individual models evolve.

Safety evaluation models and their automatic generation: Several evaluation models have been proposed to facilitate a quantitative safety analysis based on architectural specifications [37]. According to the current standards for the development of safe systems in different application domains [16, 44, 45], common fault trees [3, 10, 11, 15, 21, 32, 35, 38, 46, 65, 66, 69, 81] are widely used as evaluation models. In-line with these industrial needs we focus our research in this paper on common fault trees as the main safety artifact. Alternative safety evaluation models being used in academia are Dynamic Fault Trees (DFTs) [4, 8, 22, 28], Generalized Stochastic Petri Nets (GSPNs) [73], State-Event Fault Trees (SEFTs) [39, 48, 49] or Markov models [12, 13]. In contrast to pure quantitative safety evaluation models also FMEA (Failure Mode and Effect Analysis) tables could be considered and are constructed from architecture specifications [18, 19, 20, 67]. To construct

[1] https://www.informatik.hu-berlin.de/en/forschung-en/gebiete/se/research/ongoingprojects/ensure/coevolution/

the safety artifacts, the system architecture models are often annotated with failure propagation models [24, 26, 35, 50, 48, 66]. These failure propagation models are commonly combinatorial in nature [66, 68] thus producing static fault trees.

Beside annotating an architecture specification, there are also approaches to construct a safety artifact via model checking techniques [36, 40, 41, 42, 59]. To keep the models consistent with the architectural models, safety evaluation models are usually generated with generative techniques rather than using (co-)evolution techniques. Such generative techniques are commonly unidirectional and generate a safety evaluation model from the architectural specifications manually [35] or quasi-automatic [31, 32, 66, 68] if the required annotations are present in the architectural model. In contrast to all the above mentioned approaches, this work focuses on co-evolution of architecture specification and safety evaluation models, and thus tackles the problem from a different angle.

Model synchronization and co-evolution approaches: Complementary to the generative approaches in the safety domain, co-evolution of multiple models, and specifically ensuring their consistency, has been named as one of the challenges of software evolution [75]. Since then, several approaches have been developed which address the problem of how to achieve a consistent co-evolution of multiple inter-related MDE artifacts.

Many approaches which aim at achieving consistent co-evolution of multiple inter-related models render the problem as a synchronization problem. Existing work on model synchronization typically focuses on fully automatic approaches using model transformation languages like Triple Graph Grammars (TGGs)[34, 76], PMT [84], Atlas Transformation Language (ATL) [47], Groove [70], Query/View/Transformation (QVT) [64], and the Janus Transformation Language (JTL) [17] (see also [77] for a recent special issue on current model transformation approaches). Bergmann et al. [9] present a novel type of model transformation to which they refer to as change-driven transformations. Change-driven transformations are triggered by model changes in a source model and can be utilized to incrementally synchronize inter-related target models. Similar approaches are presented in [60, 62, 87]. Madari et al. [60] emphasize the explicit maintenance of trace models between inter-related models in order to facilitate incremental model synchronization when source and target models originate from different domains. We draw on available MDE technologies by using the Henshin model transformation language [5, 80] as one of the technical foundations of our approach. However,

in contrast to ours, existing approaches and do not enable the user to influence the synchronization and are therefore not applicable for problems where co-evolution and thus synchronization is not deterministic as in our case. An exception of this is the approach proposed by Milovanovic et al. [62] which comprises some interactive elements when adapting database schemata in response to changes in object-oriented data models. User interactions are utilized to improve the accuracy of the difference calculation when determining the source model changes. The actual synchronization, however, is fully automated, which is rather straight forward in their scenario due to the structural similarity of object-oriented and relational data models.

Another class of approach for dealing with model inconsistencies in interrelated models is generally known as model repair. Several approaches have been developed which deal with inconsistencies by constructing repair actions [25, 27, 63]. They address the problem of consistency preservation in the context of user induced changes. However, these repair actions are restricted to small changes and do not enable complex transformations which are supported by our approach.

Finally, the co-evolution of different kinds of MDE artifacts has been studied in the literature. An example is the work of Ruhroth and Wehrheim [74] for supporting the co-evolution of models of the same modeling language where one model is the refinement of another. Moreover, several approaches support the co-evolution of meta models and related artifacts, such as the migration of instance models [14, 82], model transformations [75], or syntactic and semantic constraints [23] in response to meta model changes.

3. Background

Our work is based on model-driven software engineering, particularly, modeling software architectures and fault trees as well as specifying model transformations. Therefore, we briefly introduce our two modeling languages including their meta-models as well as the Henshin transformation language [5] which we use to specify model transformations.

3.1. Modeling Languages

This paper studies the co-evolution of two types of models, detailed in the Sections 3.1.1 and 3.1.2: *i.) system architecture (SA)* models, focusing on a structural system decomposition into components and their connections,

as well as *ii.) fault trees (FT)*, which are used to analyze the causes of undesired system states. In both cases, well-known concepts from architecture description languages (ADLs) [83] and fault tree modeling [85], respectively, are used. We use the Eclipse Modeling Framework (EMF) [79] as a technical foundation.

3.1.1. Architectural Modeling

Similar to common ADLs, the core entities provided by our SA language for describing system architectures are components, ports, and connectors. Figure 1 depicts the SA meta-classes and their relations. The SA distinguishes between type and instance level of components and ports, i.e., two meta-classes exist for each of these elements (ComponentType and ComponentInstance; PortType and PortInstance). Component types are further distinguished between hardware and software (HardwareComponent, SoftwareComponent); hardware components may be electronic (ElectronicDevice, e.g., a Sensor) or mechanical (MechanicalDevice, e.g., an Actuator). Components may be composite structures of other interconnected components. A component type contains a set of ports (PortType); on the instance level, connectors (Connector) are used to assemble component instances via ports (PortInstance). A set of intra-model constraints (not included in Figure 1), expressed in OCL, completes the specification of the SA meta-model.

3.1.2. Failure Model and Fault Trees

Figure 2 depicts the FT meta-classes and their relations. The FT language allows the definition of a failure model and a set of corresponding fault trees. A failure model includes the definition of ErrorTypes, ErrorInstances, FailureTypes and FailureInstances, based on Avižienis et al.'s taxonomy [6]. To exemplify the difference between instance- and type-level, a sensor error is an ErrorType, while the error of a specific sensor is an ErrorInstance. The core (abstract) entities of a fault tree are events (Event) and gates (Gate). A gate is a boolean function (AND, OR, XOR, etc.) that combines multiple input events into a single output event. Three different concrete types of events exist: *i.)* the top event (Hazard)—a FT's root element—corresponds to the undesired real-life hazard whose causes are analyzed using the FT; *ii.)* basic events (BasicEvent)—a FT's leaf elements—with associated probabilities of occurrence correspond to an ErrorInstance, which is not further decomposed; *iii.)* intermediate events (IntermediateEvent) are all other events in a FT, i.e., they are both outputs and inputs of gates. Like for the SA meta-model,

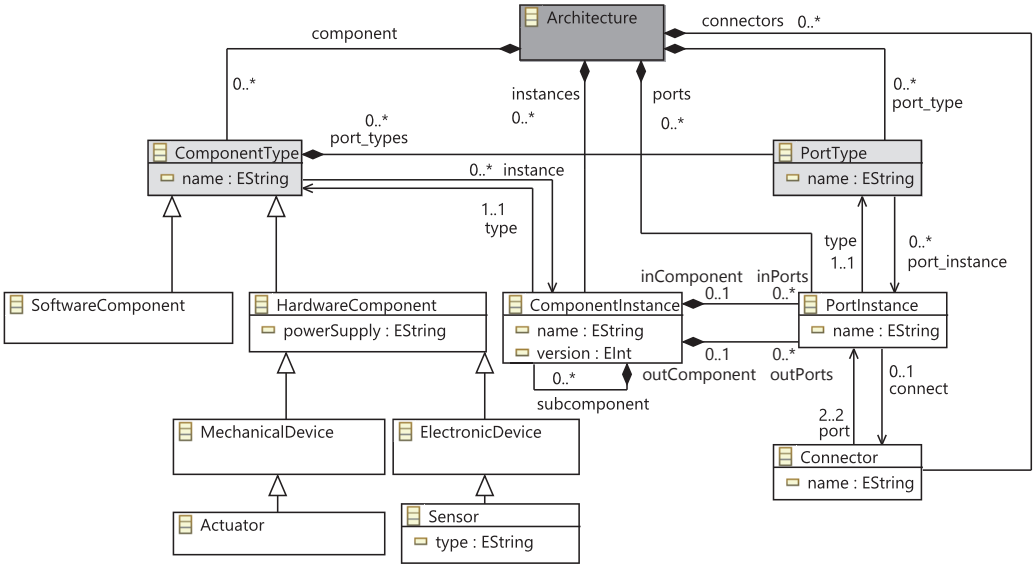


Figure 1: Classes and relations in the SA meta-model

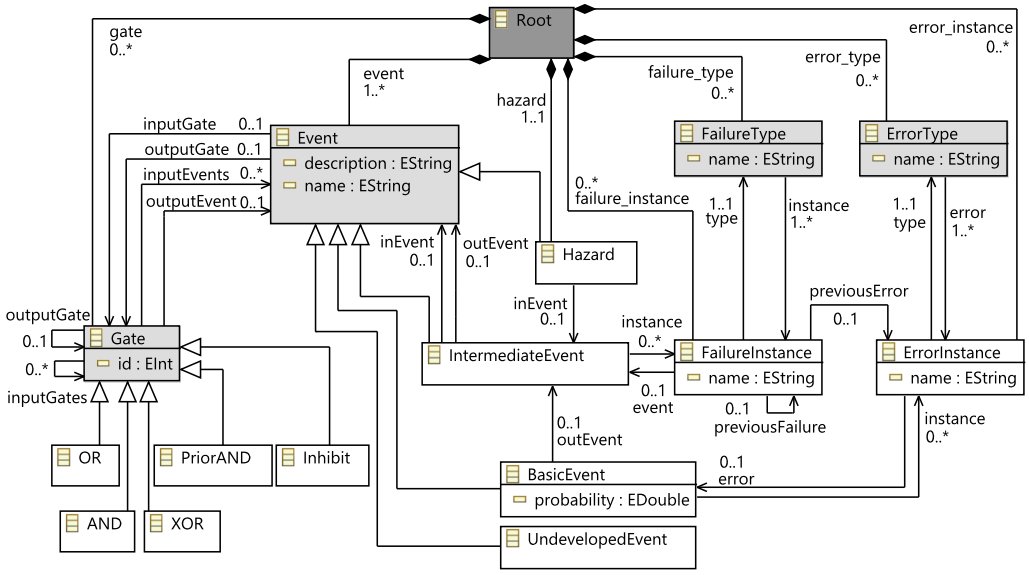


Figure 2: Classes and relations in the FT meta-model

the FT meta-model is completed by a set of OCL constraints (not shown in Figure 2).

3.1.3. Modeling of SA/FT Interrelations

Generic trace elements are used to connect component instances in an SA model to failure and error instances in a corresponding FT model. Such trace elements are represented as instances of the generic trace meta-model provided by Henshin (shown in Figure 3). As we can see in Figure 3, a Trace instance may be used to relate arbitrary model elements of type EObject, the common generic base type of any model element in EMF. Our convention is to use component instances in an SA model as source elements while failure and error instances in a corresponding FT model are used as target elements of Trace instances.

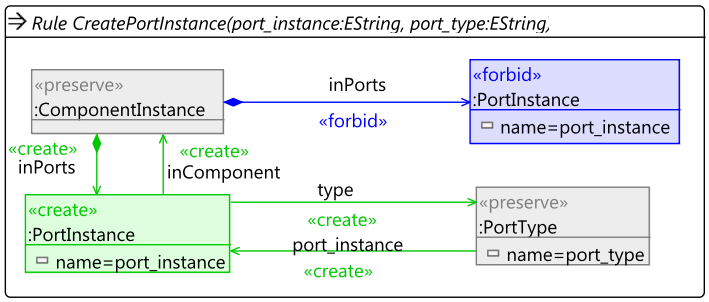


Figure 4: Henshin rule specifying the creation of a PortInstance

3.2. Henshin Model Transformations

Henshin [5] is a high-level graph rewriting and model transformation language and tool targeting models defined in the Eclipse Modeling Framework (EMF) [79]. Henshin is based on the foundations of algebraic graph transformation [72]. It provides a powerful modeling formalism including multi-rules, control flow and higher order transformations. Furthermore, it supports execution by interpretation, state space generation and an API to execute the model transformations in normal programs.

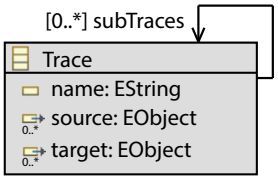


Figure 3: Trace metamodel [2] used for modeling SA/FT interrelations

Figure 4 shows a simple Henshin transformation rule, typed over the previously presented SA metamodel, for the creation of a port instance. We present the rule using the visual syntax of the Henshin transformation language in which the left- and right-hand sides of a rule are merged into one graph, indicating the model patterns to be found, to be created and being forbidden by the rule. The rule *CreatePortInstance* is applicable if 1) a component instance and a port type with a dedicated name given as rule parameter *port_type* exist in the model, and 2) a port instance with the same name as the name of the port instance to be created, given by parameter *port_instance*, does not already exist connected to the component instance. If the rule can be executed, a port instance is created with the given name and connected to the port type and the component instance.

More generally, apart from the example illustrated in Figure 4, the left-hand side of a rule comprises all model elements stereotyped by *delete* and

preserve. The right-hand side contains all model elements annotated by *preserve* and *create*. Negative application conditions, i.e., existing model patterns preventing a rule from being executed, are stereotyped by *forbid* and rendered in blue color.

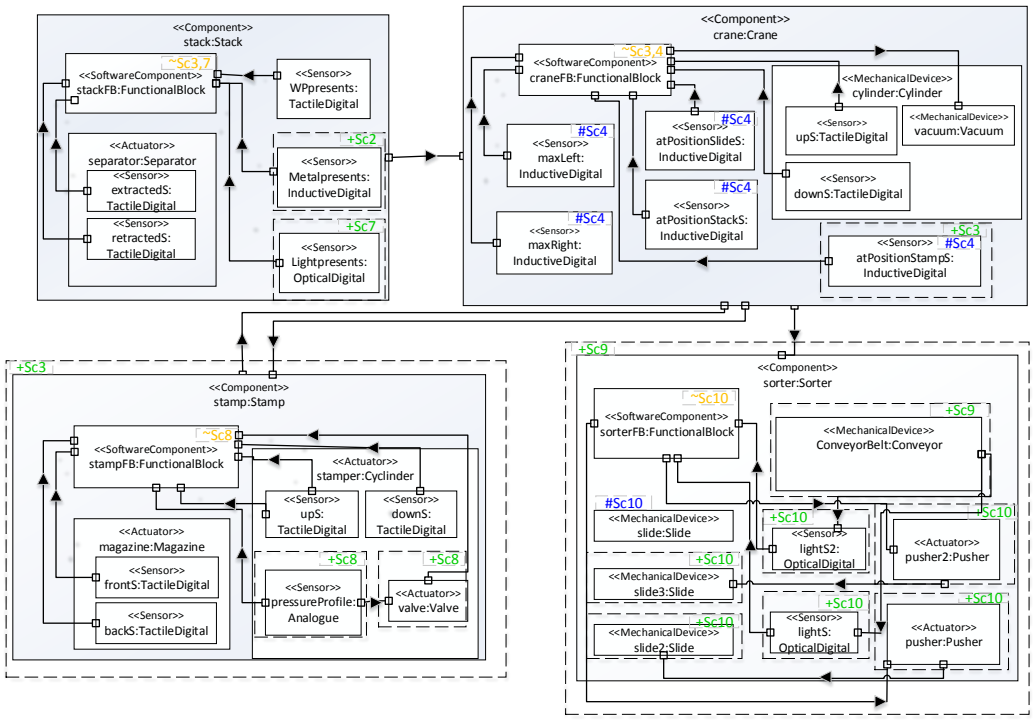
4. Case Study on Architecture and Fault Tree Co-Evolution

The system we studied is an industrial production plant, called pick&place unit (PPU). The PPU is a factory automation system that mimics an industrial robot that moves work pieces (WPs) between different working positions where they are stored or processed. Such systems are an interesting case for evolution since they contain mechanical parts, electrical parts, and software parts. All these parts can be evolved individually or in combination. Additionally, these systems are also typically safety-critical.

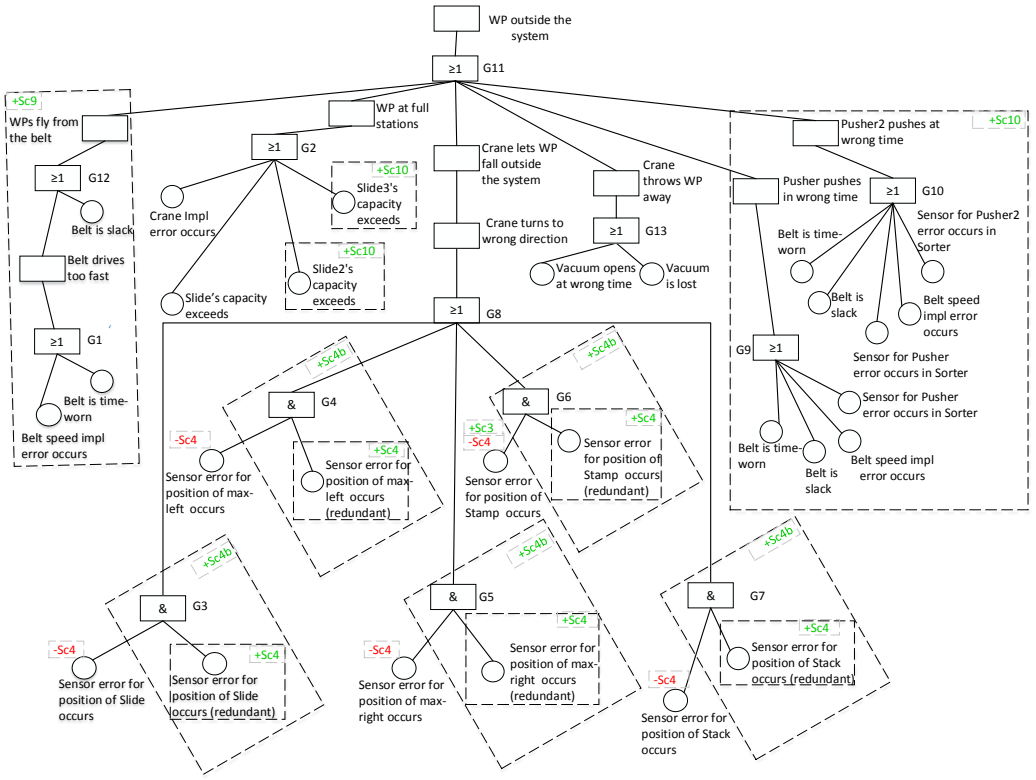
The evolution scenarios for the PPU have been described in [58] and we selected a subset of 12 scenarios for our study [30] out of these 14 evolution scenarios —those that were identified as including system changes affecting the system's safety properties. For each scenario, we manually created SA and FT models, conforming to the meta-models described in Section 3.1. To reliably identify model elements over time, successive model versions in the historical evolution have been created as revisions of each other, and model elements have been equipped with persistent yet universally unique identifiers [52, 56]. Section 4.1 briefly summarizes the different scenarios and the changes they implied to the software architecture and fault tree models. Section 4.2 includes the co-evolution analysis.

4.1. Evolution Scenarios

Section 4.1.1 includes a compact description of the PPU's initial scenario (SC0), including the corresponding SA and FT model. A brief summary of the changes in the subsequent evolution scenarios is provided by Section 4.1.2. Note that in the latter, changes to the SA and FT models—as detailed in Section Appendix A—are not covered for the sake of brevity. Figure 5 summarizes the SA and FT models for the scenarios SC0–10, including changes. Additional models can be found in the detailed description of the scenarios in Appendix A (Figures A.13–A.15). Note that a similar description of the PPU scenarios is also provided in [58] (without considering SA and FT models) and [30]. However, we decided to include it to make this paper self-contained. Table 1 provides a quantitative summary



(a) SA component model of the PPU



(b) FT model until scenario SC10 for the hazard that a WP is outside the system (FT1)

Figure 5: SA and FT models for the PPU scenarios SC0–10.
(Legend for change operations: + addition, - deletion, # replacement by other implementation, ∼ new version of implementation.)

	System Architecture						Failure Model								Fault Tree 1&2					
	+CpType	−CpType	+CpInstance	−CpInstance	+SCpInstance	−SCpInstance	+ErrorType	−ErrorType	+ErrorInstance	−ErrorInstance	+FailureType	−FailureType	+FailureInstance	−FailureInstance	+BasicEvent	−BasicEvent	+Gate	−Gate	+IntermEvent	−IntermEvent
SC0	8	0	3	0	14	0	5	0	8	0	3	0	4	0	8	0	4	0	4	0
SC2	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SC3	2	0	1	0	8	0	0	0	7	0	1	0	1	0	4	0	1	0	1	0
SC4a	0	0	0	0	5	5	0	0	5	5	0	0	0	0	6	6	0	0	0	0
SC4b	0	0	0	0	5	0	0	0	5	0	0	0	0	0	6	0	5	0	0	0
SC7	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
SC8	2	0	0	0	2	0	1	0	4	0	0	0	1	0	4	0	2	0	1	0
SC9	2	0	1	1	3	0	1	0	3	0	2	0	2	0	3	0	2	0	2	0
SC10	1	0	0	0	6	0	0	0	11	0	0	0	2	0	11	0	2	0	2	0
SC11	0	0	0	0	2	0	0	0	7	0	1	0	1	0	7	0	1	0	1	0
SC13	1	0	0	0	1	5	0	0	1	5	0	0	0	0	1	6	0	1	0	0
SC14	1	1	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	0	0	0
$\sum$	19	1	5	1	49	11	7	0	53	11	7	0	11	0	52	13	17	1	11	0
#Scn.	9	1	3	1	12	3	3	0	11	3	4	0	6	0	11	3	7	1	6	0

Table 1: Number of changes (by type) per scenario

of selected model changes to the SA and FT models in the different evolution scenarios. The column headers indicate the considered types of SA and FT model changes, the latter are conceptually classified into failure model changes and changes in the actual fault trees called Fault Tree 1 and Fault Tree 2, respectively. For example the abbreviation *+CpType* denotes the addition of a ComponentType element, and *-SCpInstance* denotes the removal of an ComponentInstance element of type SoftwareComponent. The first 12 rows describe for each individual scenario the number of applied additions and removals of specific elements in the considered models. The second to last row provides the total number of occurrences for each addition or removal over all scenarios, and the last row summarizes the number of scenarios, in which each type of change was involved. Overall, this table gives an intuition of how many changes occurred in each scenario, and how often each change occurred during the (co-)evolution.

4.1.1. Initial Scenario (SC0)

In the initial scenario, the PPU consists of a stack, a crane, and a slide. The crane places a WP at the slide, which serves as the output storage. The PPU includes nine sensors, e.g., to detect the presence of a WP at the pick up position of the stack and to detect the position of the crane. The PPU

processes only one kind of WPs (metallic).

In the SA model (Figure 5(a)), the PPU is decomposed into three top-level component instances for stack, crane, and slide (depicted as part of the sorter introduced in SC10)—with a dedicated component type for each. The stack and the crane are further decomposed including the software components responsible for their control. The FT model for this scenario includes five error types (e.g., software implementation error, sensor error), three failure types (e.g., position failure), as well as respective failure and error instances for the respective component instances. Figure 5(b) shows an FT, referred to as FT1, for the hazard that a WP is outside the system. Failure and error instances are related to component instances during the development of the transformations. For instance, a sensor component in Figure 5(a) called *atPositionStackS* is related to the basic event called 'Sensor error for positions of Stack occurs" in fault tree model given in Figure 5(b).

4.1.2. Evolution Scenarios (SC2–SC14)

- *SC2 (Black Plastic WPs).* The PPU is extended by a sensor in the stack in order to distinguish a new type of WPs (black plastic) from the already supported type of WPs.
- *SC3 (Stamp Module Added).* The PPU is extended by a module that is used to stamp metallic WPs.
- *SC4 (Inductive Sensors for Crane Positioning).* The crane positioning sensors are replaced by more robust devices.
- *SC4b (Increase Reliability of Crane Positioning).* As a variant of SC4 (which remains the basis for the next scenarios SC7–SC14), the new sensors are added in addition to the (now remaining) existing sensors.
- *SC7 (Additional White WPs).* White WPs are supported by adding another sensor to the stack.
- *SC8 (Different Pressure Profiles).* The PPU's stamp is extended by a proportional valve and an analogue pressure sensor to support stamping with different pressure profiles.
- *SC9 (Installation of Sorter).* A sorter is added to the PPU, which comprises a conveyor (belt) that transports WPs to the slide—now located at the end of the belt.

- *SC10 (Additional Slides and Pushers).* Two additional slides are added to both sides of the conveyor belt to increase the PPU's output storage capacity. To support this functionality, two corresponding pushers and sensors to detect WPs are added.

- *SC11 (Specific Order of Work Pieces).* The PPU's conveyor is extended by additional sensors that serve to sort WPs by type, each type of WP is transported to one of the three slides.

- *SC13 (Potentiometer at the Crane).* The crane's five individual positioning sensors are replaced by a potentiometer to increase the accuracy and to avoid spending cables and terminal blocks.

- *SC14 (Incremental Encoder at the Crane).* The crane's potentiometer is replaced by an incremental encoder to increase resistance to electromagnetic influences.

4.2. Lessons Learned from the Case Study

We described the different scenarios of the case study and the corresponding changes of the software architecture and fault tree models in the previous sections. Based on these models, we analyze how the two models co-evolve. Particularly, we investigate the research question *How high is the dependency between changes of the SA and the FT models?*

In the following, we describe two research approaches, namely correlation and mining analysis and their results to answer the research question. The first approach manually inspects the models and their changes and uses Pearson's correlation coefficient to assess the dependency between changes of both models (see Section 4.2.1). This approach has the advantage that it only requires the type and amount of changes in each scenario and as such can assess the dependency between parts of the models where no immediate connection exists. It has however the disadvantage that a correlation between changes does not mean that the changed objects are actually connected. Thus, as a complementing approach, we also mine the co-evolved models to identify those changes in both models which are corresponding in the sense that the changed objects are connected to each other (see Section 4.2.2).

				Failure Model									Fault Tree1& 2					
				+ErrorType	-ErrorType	+ErrorInstance	-ErrorInstance	+FailureType	-FailureType	+FailureInstance	-FailureInstance		+BasicEvent	-BasicEvent	+Gate	-Gate	+IntermEvent	-IntermEvent
		Σ		7	0	53	11	7	0	11	0		52	13	17	1	11	0
			#Scn.	3	0	11	3	4	0	6	0		11	3	7	1	6	0
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) Correlation between SA and FT change types

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

(b) Trace Results

Figure 6: Analysis results. (+) addition; (-) deletion

4.2.1. Correlation analysis

Both the SA and FT models of a scenario SCi+1 are a result of applying a sequence of changes to the SA and FT models of scenario SCi. We distinguish between SA and FT change types, which involve the addition and removal of entities from the respective meta-model, e.g., component types (+/-ComponentType), error instances (+/-ErrorInstance), and basic events (+/-BasicEvent). For each of the 12 evolution scenarios, we counted the number of applied changes grouped by change type. In total, we consider six different SA model change types and 14 different FT model change types. These results, which form the basis for the further correlation analysis, are listed in Figure 6(a). Each column comprises the number of how many times the change type represented by the column has been applied in the respective scenario. For example, eight component types are added in SC0. The bottom rows include the total number of applied changes per type over all scenarios and the number of scenarios in which this change type was applied. Note that the changes to FT1 and FT2 are merged. In order to quantify the linear relationship between the changes in the SA and FT models per type, we computed the well-known Pearson correlation coefficient $r_{X,Y}$ for each combination of change count vectors for SA change type $X =< x_0, x_2, \ldots, x_{14} >$ and FT change type $Y =< y_0, y_2, \ldots, y_{14} >$ over all scenarios, with x_i and y_i representing the number of SA and respectively FT changes of this type in SCi. A Pearson correlation value $r_{X,Y}$ is in the range between -1 and 1, with -1 and 1 indicating high negative/positive linear relationship, and 0 indicating no such relationship. The computed correlation coefficients for the case study are listed in Table 6(a). Note that we will omit all correlation coefficients from the further discussion, which involve change types (in either X or Y) that occur in less than three scenarios and, thus, too seldomly.

Three clusters of correlation values can be observed in the data: *i)* $-0.44 \leq r_{X,Y} \leq -0.16$, *ii)* $0.22 \leq r_{X,Y} \leq 0.48$, and *iii)* $0.61 \leq r_{X,Y} \leq 1.0$. The further discussion is limited to relationships in the latter group (bold values in Figure 6(a)), considered to reveal a high (linear) correlation. High linear correlations can be observed for additions of component types, top-level component instances, and subcomponent instances with additions of error types, failure types, failure instances, and intermediate events. The addition of subcomponent instances also shows high correlations with additions of error instances, basic events, and gates. High correlations can also be observed for deletions of subcomponent instances with deletions of error instances and

basic events. The addition of component types, component instances, and subcomponent instances roughly show similar correlation patterns, in that they show high correlations with the same set of FT change types. However, comparing even the pairs of highly correlated change types such as the addition of component and error types with the vectors of Figure 1, it can be observed that the number of changes of one type not always equals the number changes of the other type in the same scenario. The only exception is the relationship of deletion of subcomponent instances with error instances and basic events.

4.2.2. Mining Analysis

The mining approach analyses not only whether change types correlate with each other, it also considers whether the changed objects are connected to each other and if yes, how. The approach checks for each object in the software architecture model, whether it was created (denoted as "+") or deleted ("-") in a scenario. For each of those created and deleted objects, it mines in the models whether the object is connected (directly or indirectly) via a Trace element to an object in the fault tree model which is either created, deleted or unchanged ("="). In other words, we identify those changes which happened in the same scenario and the changed objects are connected to each other and were not just coincidently changed in the same scenario.

The mining, i.e., our notion of connectedness, is formalized by a set of model patterns expressed as Henshin rules (left hand side equals right hand side) which contain elements of the source and target models being connected by intermediate elements which express the concrete relation between them. Figure 7 shows the mining pattern for the connection between ComponentType and BasicEvent. In the models of the PPU case study, only ComponentInstances are connected by trace elements to elements in the error model, i.e., ComponentType and BasicEvent elements may be indirectly connected. Thus, the pattern specifies that ComponentType and BasicEvent are linked via a Trace element between their instances (ComponentInstance and ErrorInstance). Overall, we use 18 different mining patterns for all possible combinations of classes and connections.

We call those changes to connected objects *corresponding* and use $c_{src,trgt}$ for the number of corresponding changes in all scenarios for a change type src in the SA model and a change type $trgt$ in the fault tree model, i.e., src refers to a row in Figures 6(a) and 6(b) and $trgt$ refers to a column. We use n_{src} for the number of all changes for a change type src. The cells

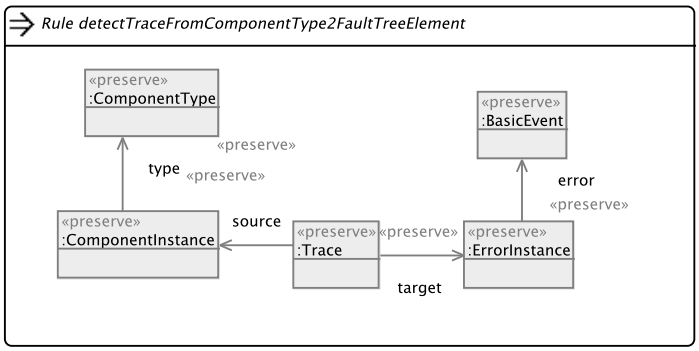


Figure 7: Mining pattern for trace elements between ComponentType and BasicEvent

in Table 6(b) contain then the fraction of corresponding changes between source and target model with respect to all changes in the source model: $f_{src,trgt} = c_{src,trgt}/n_{src}$. For example, $f_{+CpType,+ErrorType} = 0,17$ describes that 17% of the newly created component types are connected to a newly created error type, i.e., one which was created in the same scenario.

Furthermore, we also mined for those changes in the scenario where a created object (e.g., subcomponent instance +SComponentInstance) is connected to an existing object (e.g., error instance =ErrorInstance). This enables to compare whether created objects in the software architecture models are linked to created, to existing, or no objects in the error model. Similarly, we mined also for corresponding deletions, e.g., a subcomponent instance is removed -SComponentInstance and also the connected error instance is removed -ErrorInstance. The case that the error instance remains is shown as =ErrorInstance.

Only changes of subcomponent instances (+/-SComponentInstance) and additions of component types (+CpType) happen more than 10 times in order to draw some conclusion. We see that created subcomponent instances in the majority of cases are connected to *existing* error types. With respect to error instances and basic events in the fault tree, they are in contrast connected to newly *created* objects. The reverse change type, deletions of subcomponent instances, show also the reverse corresponding changes, deleting error instances and basic events and keeping error types. Created component types are in more than half of the cases connected to newly created basic events, but this is due to the fact that component types are also connected to component instances.

4.2.3. Results and Discussion

Both the correlation and mining analysis confirm our preliminary results [30] for this case study that no simple and straightforward co-evolution of SA and FT models exists that could be automated. However, both mining approaches could be exploited in a co-evolution framework that includes user interaction, by prioritizing potential co-evolutions based on the data.

We are aware of major threats to validity concerning our conclusions, namely *i.)* the limited statistical significance due to the low number of observations (conclusion validity), *ii.)* the fact that all models have been developed by the same people —while in practice they are developed by different teams (internal validity), *iii.)* the consideration of grouped changes per scenario (construct validity), as well as the fact that *iv.)* we investigated only a single case (external validity). We argue that it is a challenge to find a consistent model co-evolution case study such as the PPU scenarios used in this paper. In fact, we are only aware of one other case study on model co-evolution which is presented by Herrmanndörfer et al. in [43]. However, the study is on the co-evolution of meta-models and instance models, the co-evolution of multiple instance models is not addressed. We experienced that it is already extremely hard to consistently co-evolve the SA and FT models for this—seemingly small—case study. However, particularly w.r.t. to *ii.–iv)* we do not see that these threats have a major impact on our conclusion that co-evolution cannot be fully automated but requires user interaction.

5. Supporting Co-Evolution

In Section 4, we presented the analysis results and conclude that change actions between system architecture and fault trees are not straightforward. For this reason, developers can only be assisted by a recommender system that offers a set of meaningful transformations, which is rich enough to consistently co-evolve system architecture and fault tree models. Specifications of such transformations can be integrated into a rule-based framework such as the CoWolf tool [29] assisting developers in achieving consistent co-evolution in a step-wise manner.

As described in the following subsections, the creation, identification and the co-evolution of the relations between two models such as system architecture and fault trees can be achieved by means of model transformations incrementally. We use Henshin model transformations in order to capture

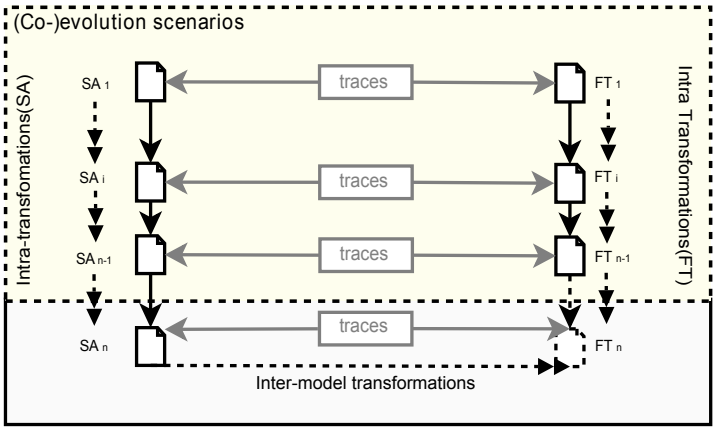


Figure 8: The role of transformations supporting model (co-)evolution

i.) co-evolution actions between SA and FT models, and *ii.)* evolution actions in the individual models. In CoWolf, the former are used to support developers in model synchronization, while the latter provide assistance in conveniently performing isolated changes in individual models.

We illustrate our concept and basic notions which are used in the remainder of this article in Figure 8. Every version SA_i in a history of co-evolving SAs and FTs has a corresponding version FT_i. We denote such a pair (SA_i,FT_i) of two corresponding models a *couple*. These couples are connected via *trace* elements that associate elements from different corresponding models with each other (see Section 3.1.3). A couple (SA_i,FT_i) represents a consistent snapshot of our sample system, i.e. SA_i and FT_i have been consistently co-evolved in each evolution step of the PPU.

We distinguish two kinds of model transformations: *Intra-transformations* and *inter-transformations* as presented in the following subsections.

5.1. Intra-model Transformations

For every type of model (here SA and FT), there are intra-model transformations that execute evolution actions internally within this type of model, i.e. without changing the corresponding model (see vertical transformations in Figure 8). This implies that every change between two versions of a model can be partially described by applications of rules from this set of intra-transformation rules.

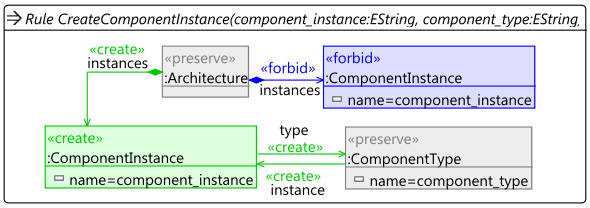
Figure 9 demonstrates two examples of SA intra-transformation rules specified in Henshin (see Section 3.2 for a brief introduction into the Henshin model transformation language). The first rule in Figure 9(a) specifies the creation of a new component in an SA model. The rule, called *CreateComponentInstance*, gets the names of the component to be created and its desired type as input parameters. The rule is applicable if such a component type exists and there is not yet a component having the same name as the one which is to be created. When being applied, it adds the created component to the architecture model and sets the desired component type. The second rule in Figure 9(b), which is more complex than the first one, specifies the creation of a connection between two components. The rule, called *CreateConnection*, gets the names of two components that shall be connected as well as the names of the ports and the connector to be created as input parameters. It creates the whole connection, actually a complex model pattern including in- and out-ports that are to be connected by the connector. Like most of our intra-model transformation rules, the rule is equipped with a negative application condition (NAC) in order to preserve internal consistency constraints of a model to which the rule is applied. Similar to the transformation rule in Figure 9(a), the NAC exposed by the transformation rule *CreateConnection* ensures the uniqueness of names.

Note that the change specified by the transformation rule *CreateConnection* could be achieved by a sequence of sub-rules, namely the creation of the required ports followed by the creation of the connector. Both sub-rules, i.e., the creation of a component port as well as the creation of a connection between existing ports, are also included in our set of SA intra-transformation rules (not presented in this paper). Nevertheless, a compact rule such as *CreateConnection* achieves this change in a single step, which demonstrates our aim of reducing the amount of manual work to achieve consistent model (co-)evolution.

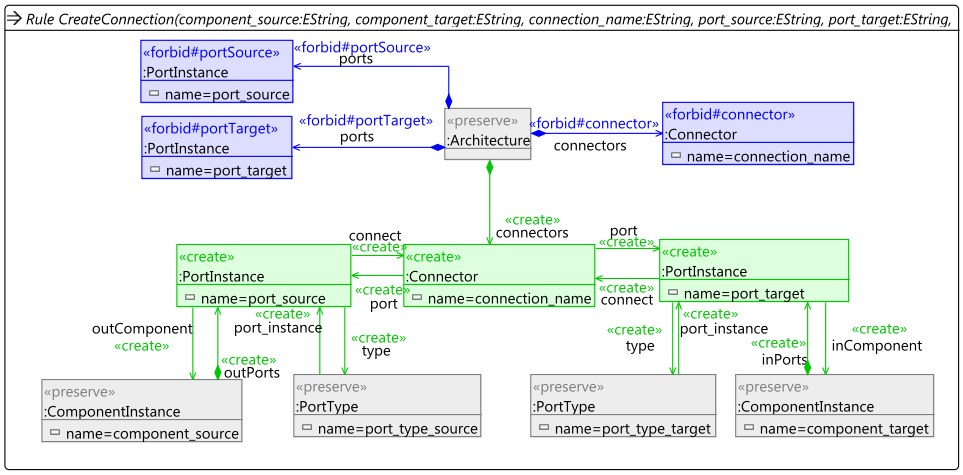
Internal transformations for FT models have been created in the same manner. As a result, we have created 42 intra-model transformations for SA and 57 intra-model transformations for FT.

5.2. Inter-model Transformations

Inter-model transformations (horizontally shown in Figure 8) have been created to describe co-evolutions. They are used to execute the corresponding changes on an FT model when an SA model undergoes changes, i.e., to effectively achieve semi-automated model synchronization through change



(a) Component instance creation



(b) Connection creation

Figure 9: Intra-model transformations for System Architecture

recommendations. Trace elements between couples play the key role in the identification of the relations and are extensively used by our inter-model transformations. All inter-model transformation rules include at least one Trace object representing a connection between an SA and an FT model. In sum, we have developed 16 inter-model transformation rules describing possible co-evolution steps of SAs and FTs as presented in Table 2. We can classify these rules in four categories to which we refer to as *consistency*, *coupling/connecting*, *decoupling* and *propagation*, respectively.

Firstly, *consistency* transformation rules aim at ensuring that if an SA ComponentInstance has a connection to an ErrorInstance or a FailureInstance, then an associated BasicEvent or IntermediateEvent is being created. The other way round is also possible, i.e., a BasicEvent or IntermediateEvent associated to an ErrorInstance or FailureInstance having no connection to a ComponentInstance may be deleted. Such rules can support developers in achieving consistent co-evolution by adding the corresponding element (1-2 in Table 2) or removing it as reverse transformations (3-4 in Table 2). For instance, the rule *EnsureBasicEvent* shown in Figure 10(a) gets a component name and event id as input. The rule searches for a ComponentInstance with the given name and checks if there is an ErrorInstance being connected via a Trace element. If so, the rule is applicable and creates a BasicEvent with given event id. Hence, the rule *EnsureBasicEvent* enables the developer to ensure the existence of a basic event for each component in the architectural model being connected to an ErrorInstance but yet lacking a basic event. However, this cannot be done fully automated since the developer needs to decide in which fault tree the basic event is relevant and thus shall be added by executing the transformation rule.

Secondly, *coupling/connecting* transformation rules aim at creating new trace objects to relate the corresponding elements between M_{SA} and M_{FT}. In other words, they specifically add new inter-model relationships to the models. An example is demonstrated with the rule *CreateAssociatedErrorInstance* in Figure 10(b). The rule *CreateAssociatedErrorInstance* basically builds the relationship for corresponding elements of type ComponentInstance and ErrorInstance by ensuring that there there is an error type defined for the ErrorInstance. It creates one ErrorInstance and the Trace between a ComponentInstance and an ErrorInstance at the same time. Coupling/connecting transformations can i) add new trace objects together with new elements (5-6 in Table 2) as demonstrated in Figure 10(b), which creates for an existing ComponentInstance an associated ErrorInstance in the FT model and connects

these two elements with a Trace object, and ii) connect existing elements with a new Trace object (7-11 in Table 2).

Third, *decoupling* transformation rules reverse the coupling transformation functionality. In Figure 10(c), we provide a generic rule to clear a trace object that does not require a specific name.

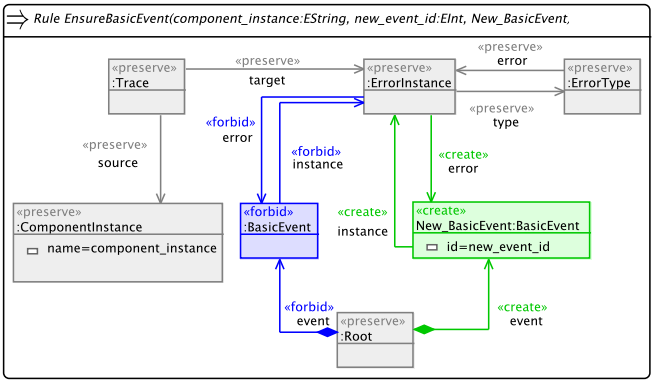
Finally, *propagation* transformation rules aim at implementing various combinations of inter-model relationships on a SA model. For example, the transformation *PropagateFailureToParentComponent* propagates the FailureInstance of a ComponentInstance to its parent component ComponentInstance by applying *CreateAssociatedFailureInstance* (Rule 6). The transformation is applicable if there exists a parent ComponentInstance not being connected to the FailureInstance yet.

Category		**#**	**Name**	**+Node**	**+Edge**	**-Node**	**-Edge**
consistency	addition	1	EnsureBasicEvent	1	2	0	0
	addition	2	EnsureIntermediateEvent	1	2	0	0
	deletion	3	RemoveConnectedBasicEvent	0	0	1	3
	deletion	4	RemoveConnectedIntermediateEvent	0	0	1	3
coupling/connecting		5	CreateAssociatedErrorInstance	2	5	0	0
		6	CreateAssociatedFailureInstance	2	5	0	0
		7	ConnectPortInstanceWithFailureInstance	1	2	0	0
		8	ConnectComponentInstanceWithFailureInstance	1	2	0	0
		9	ConnectComponentInstanceWithErrorInstance	1	2	0	0
		10	ConnectPortInstanceWithErrorInstance	1	2	0	0
		11	ConnectConnectorWithFailureInstance	0	0	1	2
decoupling		12	ClearTraceElement	0	0	1	2
propagation		13	PropagateFailureToParentComponent	1	2	0	0
		14	PropagateFailureToConnectedPortFromPort	1	2	0	0
		15	PropagateFailureToComponentFromPort	1	2	0	0
		16	PropagateErrorToComponentFromPort	1	2	0	0

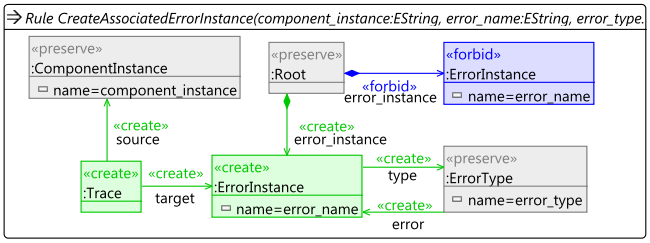
Table 2: Summary of inter-model Transformations

5.3. Running Example showing Evolution Actions

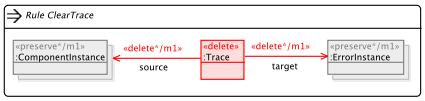
In Figure 11(a), we demonstrate one evolution step of the PPU system from scenario SC2 to scenario SC3, where the PPU is evolved by the addition of a *Stamp* component. We illustrate some changes in the system architecture (left) and their impact on the corresponding fault tree (right) for parts of the models. Here, we focus on one change in the *Crane* component and the corresponding changes in the fault tree when the stamp is added to the system. A new sensor *atPositionStampS* is added as a subcomponent of component



(a) Creation of a BasicEvent for the corresponding ComponentInstance



(b) Creation of an ErrorInstance and a corresponding Trace element



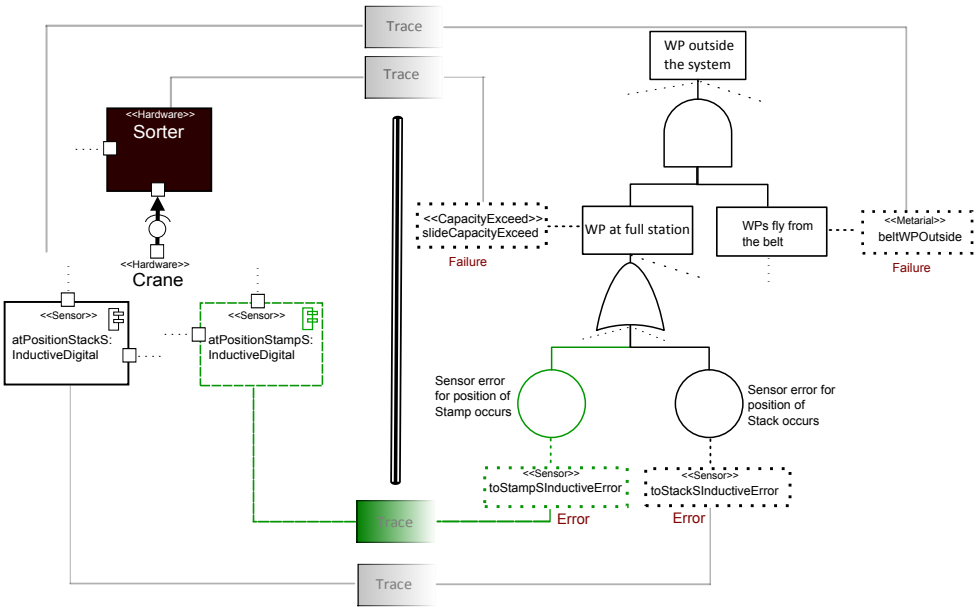
(c) Removing the Trace element

Figure 10: Inter-model Transformations between System Architecture and Fault Trees

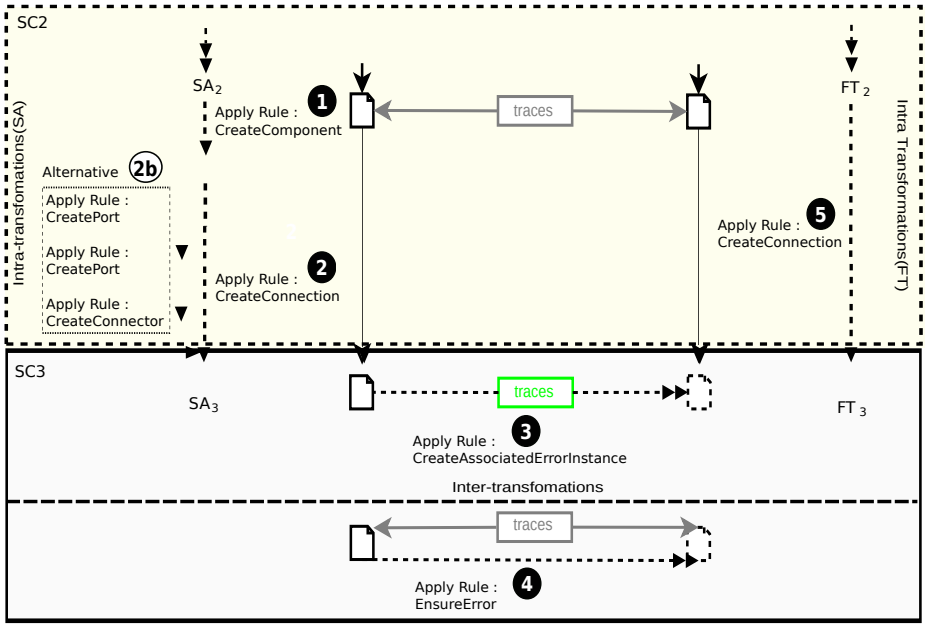
Crane. In this specific example, the new component *atPositionStampS* and its ports are associated with existing types, i.e. an existing component type and an existing port type, respectively. Therefore, no new component type or port type is being created. Concerning the related fault tree, the addition of the component *atPositionStampS* leads to the creation of a new error instance *toStampSInductiveError* which is connected to component *atPositionStampS* by a new trace element. Furthermore, the new error instance is connected to a new basic event called "Sensor error for position of Stamp occurs" such that *atPositionStampS* and the basic event are properly connected to each other. In turn, the new basic event is connected to an existing OR gate instead of appearing standalone. This gate takes the input events associated to the sibling components of *atPositionStampS* in the component hierarchy, i.e. the events associated to other subcomponents of component *Crane*.

In Figure 11(b), we illustrate how the changes of 11(a) can be achieved by applications of our intra- and inter-model transformations. First, two intra-model transformations are performed on model SA_2 by applying rules *CreateComponent* and *CreateConnection*, referred to as rule applications 1 and 2 Figure 11(b). by applying these rules, we create the new sensor component and embed it into the SA model. Thereafter, as a response to these changes, the FT model is being adapted by applying inter-model transformations *CreateAssociatedErrorInstance* and *EnsureError*. Thereby, we create new error instance being traced to the new component (rule application 3) together with a new basic event associated to that error instance (rule application 4). Finally, applying the intra-model transformation rule *CreateConnection* connecting the new basic event to an existing OR gate in the FT model (rule application 5) completes the synchronization, thus leading to a consistent co-evolution step.

As already mentioned in Section 5.1, our rule sets include basic editing operations as well as more complex transformation rules which cover several basic editing operations to improve efficiency. This is also illustrated in our example of Figure 11(b). Here, the application of the intra-model transformation rule *CreateConnection* (rule application 2) could be replaced by a sequence of three rule applications yielding the same result, namely *CreatePort, CreatePort, CreateConnector*, illustrated as Alternative 2b in Figure 11(b).



(a) Partial PPU models showing the trace elements and one evolution step from scenario 2 to 3



(b) Execution of the transformations for the change in Figure 11(a)

Figure 11: Example of an evolution step

6. Evaluation

The quality of a co-evolution framework such as CoWolf strongly depends on the quality of the transformation rules being offered to developers as interactive editing commands. Thus, we investigate two major quality aspects of our set of manually defined transformation rules which must be *i.) complete* in the sense that every couple (SA_i, FT_i) can be evolved to (SA_{i+1}, FT_{i+1}) in a consistent way, and *ii.)* helpful in the sense that this evolution can be achieved by a developer with minimal effort, which is also referred to as *task efficiency* in the literature [1].

Research methodology. We choose a quantitative approach in order to assess these quality aspects. For each evolution step of the PPU case study, i.e., for each pair of successive evolution scenarios SCi and SCi+1, we calculate model differences $\Delta(SCi, SCi+1)$ which are based on our transformation rules presented in Section 5. A difference $\Delta(SCi, SCi+1)$ provides a specification of how scenario SCi can be evolved to scenario SCi+1 using transformation rules available in a dedicated rule set. Thus, we can utilize difference metrics (see, e.g., [86, 89, 90]) in order to reason about quality aspects *i)* and *ii)* of a transformation rule set.

In our study, we use the SiLift model differencing framework [51], which has been specifically designed for the comparison of graph-structured models. In addition to the two input models M_1 and M_2, which are to be compared with each other, the differencing engine takes a set R of transformation rules as additional configuration parameter as input (cf. Figure 12). The output of the differencing engine, i.e., the difference between M_1 and M_2, is given as a sequence of rule applications, each rule is part of the pre-defined set R [54]. We write $\Delta_R(M_1, M_2)$ to refer to a difference which is based on a set R of transformations rules. Transformation rules must be specified in Henshin.

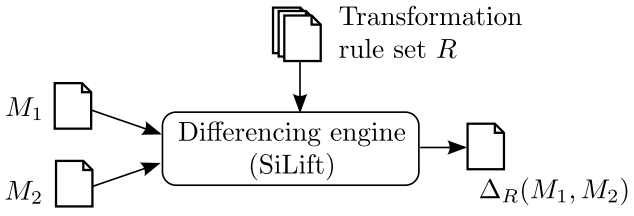


Figure 12: SiLift model differencing engine: In- and output parameters

Our study design is described in Section 6.1, quality metrics are introduced in Section 6.2. Results are summarized by Section 6.3, and Section 6.4 finally discusses threats to validity.

6.1. Study Design

Pairs of models which are subject to a difference calculation are given by the evolution steps of the PPU case study. We consider a triple $(SA_i, FT1_i, FT2_i)$, including the trace elements between SA_i and $FT1_i$ as well as the trace elements between SA_i and $FT2_i$, as a single input model. For each evolution step, we compute three distinguished kinds of differences which are based on different sets of transformation rules. Obviously, one set of transformation rules is given by our intra- and inter-model transformation rules presented in Section 5. For brevity, we refer to this set as *Inter/Intra*. The following two sets of transformation rules serve as reference rule sets for our evaluation:

- *Atomic*: Transformation rules in this set provide all atomic change operations on graph-structured models which cannot be split into smaller operations, i.e. rules to create and delete single nodes and edges of a graph-structured model.
- *Generated*: Transformation rules in this set are generated from our SA/FT meta-models (cf. Figures 1 and 2) using the approach presented in [55], which is implemented in the SiDiff Edit Rule Generator (SERGe) [71]. As argued in [55], transformation rules generated with SERGe are similar to the edit operations provided by typical editors for visual modeling languages.

We refer to the calculated kinds of differences as Δ_{Atomic}, $\Delta_{Generated}$ and $\Delta_{Inter/Intra}$, respectively. We consider these rule sets as the baseline for evaluating the quality of our intra-/inter-model transformation rules.

6.2. Measures

Completeness. Our set of intra-/inter transformation rules is complete if we can describe any evolution step based on rules available in this set. To that end, we check whether all differences $\Delta_{Inter/Intra}(SCi, SCi+1)$ can be calculated by SiLift. If we can compute a difference for each evolution scenario of the PPU case study, this serves as strong indicator for the completeness of our set of intra-/inter transformation rules.

Task efficiency. In a co-evolution framework which primarily serves as a recommender system, each transformation rule would be offered to the developer as an interactive editing command which is executable in a well-defined context. Consequently, the amount of manual work for an evolution step can be measured by counting the number of editing commands which have to be executed in order to (co-)evolve SCi to $SCi+1$. The less editing commands have to be executed, the more efficient the semi-automated (co-)evolution.

In our study design, the number of editing commands is represented by the number of rule applications contained by a difference. We write $|\Delta(M_1, M_2)|$ to refer to the number of rule applications contained by $\Delta(M_1, M_2)$. Let $R1$ and $R2$ be two sets of transformation rules serving as configuration parameter of the SiLift differencing engine, then the reduction of the number of rule applications for a difference $\Delta_{R1}(M_1, M_2)$ compared to a difference $\Delta_{R2}(M_1, M_2)$ (with $|\Delta_{R1}(M_1, M_2)| \leq |\Delta_{R2}(M_1, M_2)|$) can be evaluated by a function f_{red} as

$$f_{red}(R1, R2, M_1, M_2) = 1 - \frac{|\Delta_{R1}(M_1, M_2)|}{|\Delta_{R2}(M_1, M_2)|} \tag{6.1}$$

Thus, for each evolution step, the reduction of the amount of changes using intra-/inter-model transformation rules compared to using generic graph operations can be assessed by $f_{red}(Inter/Intra, Atomic, SCi, SCi{+}1)$. Compared to the generated transformation rule set, we can quantify the improvement by $f_{red}(Inter/Intra, Generated, SCi, SCi + 1)$.

6.3. Results

The results of the calculation of the model differences Δ_{Atomic}, $\Delta_{Generated}$ and $\Delta_{Inter/Intra}$ for each evolution step SCi→SCi+1 are summarized by Table 3.

The evolution step is shown in column 1. Columns 2 and 3 show the number of atomic graph operations, i.e., the addition and deletion of nodes (+/-Node) and edges (+/-Edge) in terms of a graph representation of our SA/FT models. The total number of atomic graph operations is summarized by column 4. It quantifies the number of editing actions being required to evolve a scenario SCi to SCi+1 if no specific tool support is provided. Columns 5 and 6 report about the number of rule applications which are required to evolve a scenario SCi to SCi+1 using transformation rules generated by SERGe. Column 5 refers to the rule applications on the SA and FT

models, column 6 quantifies the required change operations in order to evolve the respective trace elements. Both types of changes are summarized by column 7. Finally, basic properties of each difference $\Delta_{Inter/Intra}(SCi, SCi+1)$, namely the number of intra- and inter-model rule applications, are shown by columns 8 and 9, summarized by column 10.

Completeness: An important result of our evaluation is that every difference $\Delta_{Inter/Intra}(SCi, SCi+1)$ can be calculated based on our intra- and inter-model transformation rules. This means in turn that any historically observable evolution step can be expressed by exclusively using transformation rules available in our defined set of intra- and inter-model transformation rules. Thus, we can conclude that this set of rules is complete w.r.t. all evolution steps provided by the PPU case study.

Task efficiency: As already mentioned, we assume here that the number of changes comprised by a model difference SCi→SCi+1 serves as an indicator for the manual editing effort which is required to achieve the respective co-evolution step. In other words, the manual effort depends on the edit operations which are available for modifying a model. Improved task efficiency by using our co-evolution rules as edit operations compared to conventional edit operations is thus demonstrated by columns 11 and 12. The reduction of the number of edit steps compared to evolving SA and FT models using atomic graph operations is shown by column 11. On average, our intra- and inter-model rules reduce the number of changes by 84.5% (independently of the overall size of a difference), which is as a significant reduction of the amount of work for a developer. Even compared to the generated visual editor operations, as shown by column 12, our manually defined transformation rules reduce the amount of required user interactions to realize the PPU co-evolution by on average 52%. In particular, a detailed inspection of the columns 9 (*Inter*) and 6 (*Traces*) reveals that a considerable portion of the observed reduction compared to typical editor operations is achieved by our inter-model transformation rules, which are specifically designed to support the co-evolution.

To get a better impression of the usefulness of each individual inter-model transformation rule, we present a detailed distribution of the observable inter-model transformations over the 11 evolution steps of the PPU in Table 4. Not surprisingly, none of the inter-model transformations can be observed for the initial evolution step SC0→SC2 since there is no change on the fault trees and thus no co-evolution at all. For almost any other evolution step, we observe a high number of occurrences of the *consistency* rules *EnsureBa-*

	Δ_{Atomic}			$\Delta_{Generated}$			$\Delta_{Inter/Intra}$				
Ev. step	+/-Node	+/-Edge	Σ	SA/FT	Traces	Σ	Intra	Inter	Σ	Red. to $\Delta_{Bsc.}$	Red. to $\Delta_{Gen.}$
SC0→SC2	9	33	42	9	0	9	5	0	5	88,1%	44,4%
SC2→SC3	89	329	418	84	48	132	36	21	57	86,4%	56,8%
SC3→SC4	64	322	386	88	60	148	54	26	80	79,3%	46,0%
SC4→SC4b	122	530	652	144	30	174	65	16	81	87,6%	53,5%
SC4b→SC7	136	581	717	158	36	194	71	13	84	88,3%	56,7%
SC7→SC8	46	167	213	50	30	80	25	15	40	81,2%	50,0%
SC8→SC9	54	188	242	58	30	88	34	15	49	79,8%	44,3%
SC9→SC10	110	421	531	114	72	186	46	36	82	84,6%	55,9%
SC10→SC11	60	218	278	58	54	112	19	26	45	83,8%	59,8%
SC11→SC13	80	325	405	85	36	121	43	13	56	86,2%	53,7%
SC13→SC14	28	103	131	31	12	43	16	5	21	84,0%	51,2%
Avg.	73	292	365	80	37	117	38	17	55	84,5%	52,0%

Table 3: Difference metrics by calculated difference (Δ_{Atomic}, $\Delta_{Generated}$, $\Delta_{Inter/Intra}$) and evolution step SCi→SCi+1

sicEvent and *EnsureIntermediateEvent* (see Section 5.2). This means that these rules are indeed very helpful to support a consistent co-evolution of the models involved in our case study. Likewise, transformations of type *(de-)coupling/connecting* can be observed frequently over the evolution steps. In most of the evolution scenarios, the PPU is extended by additional features and thus evolving into a larger system. Therefore, coupling/connecting usually manifests in the addition of new elements and their connections, see transformations *CreateAssociatedErrorInstance* and *ConnectComponentInstanceWithFI*. Decoupling (*ClearTrace*) arises sporadically because of changing some components. Finally, *propagation* transformations are observable very rarely, particularly for larger evolution steps whose respective differences comprise many changes (e.g., SC9→SC10). The reason is that for the PPU case study the majority of errors/failures of sub-components are not propagated to the parent component in the component hierarchy. Most of the time, the sub-components of the PPU can handle the error instances (e.g. sensors) before the errors/failures are propagated to the parent components.

6.4. Threats to Validity

Our conclusions are subject to several threats to validity [88], the major ones will be discussed in the remainder of this section.

Inter-model Trans./Ev. step	SC0→SC2	SC2→SC3	SC3→SC4	SC4→SC4b	SC4b→SC7	SC7→SC8	SC8→SC9	SC9→SC10	SC10→SC11	SC11→SC13	SC13→SC14	Σ
EnsureBasicEvent (1)	0	4	6	6	1	4	3	10	7	1	1	43
EnsureIntermediateEvent (2)	0	1	0	0	0	1	2	2	1	0	0	7
CreateAssociatedErrorInstance (5)	0	14	10	10	2	7	6	20	15	2	2	88
ConnectComponentInstanceWithFI (8)	0	1	0	0	0	1	4	2	2	0	0	10
ClearTrace (12)	0	0	10	0	10	1	0	0	0	10	2	33
PropagateErrorToComponent (16)	0	0	0	0	0	0	0	0	1	0	0	1
PropagateFailureToComponent (15)	0	1	0	0	0	1	0	2	0	0	0	4
Σ	0	21	26	16	13	15	15	36	26	13	5	

Table 4: Distribution of inter-model transformations over evolution steps SCi→SCi+1

Construct validity: The reduction of the user's effort in achieving consistent co-evolution is only indirectly addressed by our evaluation, namely by showing that our transformation rule set indeed can significantly reduce the number of required editing steps. These editing steps, however, would be presented to users as sets of potential (partial) recommendations. Users would have to select the most suitable recommendations from such a set, some of which would have to be completed by passing concrete arguments to the underlying rule applications. A different approach to evaluate the efficiency of the recommendations would be to follow established guidelines for evaluation recommender systems in both online and offline experiments [78]. However, the results would not only depend on our catalog of operators but would largely be influenced by the quality of the recommendation system itself. The latter, i.e., the CoWolf framework which is intended to be configured by our transformation rules, is not a contribution of this paper and thus needs to be evaluated in a separate context.

Moreover, it is debatable whether the pure number of changes, which are contained by a difference, reflects the quality of a difference in an adequate way. However, the domain of comparison and versioning of software models still lacks a standardized set of quality metrics for model differences, and the number of changes is a commonly accepted indicator for the understandability of a difference [53, 57].

Another threat to construct validity is that we use generated transformation rules as reference value to quantify the amount of manual work being reduced by using our manually defined transformation rules. Thus, the reduction of manual work compared to sophisticated editors might actually be

smaller. However, inter-model transformations are typically not supported.

External validity: The PPU evolution scenarios and the respective SA/FT models have been created in a laboratory environment and we have only studied a single case. Thus, it is questionable whether our rules meet the requirements of real projects in the same way as they do for the PPU case study, and we might have missed several transformation rules which are helpful in other contexts. However, Legat et al. show in [58] that the evolution scenarios of the case reflect typical evolutions in industrial practice. A second threat to external validity is that the completeness of inter-/intra-model transformation rules have only been demonstrated w.r.t. the evolution steps of the PPU case study. However, due to the infinite number of valid SA and FT models, a general proof for completeness is hard to achieve. To more exhaustively evaluate the suitability of our transformation rules for achieving co-evolution of architectural models and fault trees, we would need to study another case for which a consistent co-evolution history of these types of models is readily available. However, to the best of our knowledge, there are no other data sets where we could evaluate the transformation rules on.

Moreover, it is a largely open question how the general approach would perform on different kinds of models from another domain, particularly w.r.t. the necessary effort in studying and encoding similar inter- and intra-model transformation rules for this domain. However, domain-independence is not a claim of this paper, and thus we leave such an evaluation for future work. According to our experience, the manual effort for creating an extensive catalog of transformations is hard to measure and estimate since it depends on a multitude of different factors, such as the expertise level of the developers, the sizes of meta-models, the degree of logical coupling between inter-related models, etc. However, we argue that the creation of a catalog of transformation rules is a one time setup effort while the catalog itself is a highly re-usable asset.

In conclusion, concerning our objective to provide a general framework supporting the co-evolution of SA/FT models, we are convinced that our transformation rule set serves as a valuable foundation which, if needed, can be adapted and/or extended to project-specific needs.

7. Conclusion

In this paper, we have thoroughly analyzed the co-evolution of architecture models and fault trees for a factory automation system called Pick and

Place Unit as an extension of our previous work [30].

As a major contribution, we provided a set of model transformation rules for achieving co-evolution of software architecture and fault tree models ensuring a correct evolution of both models, and demonstrated how to use these rules for a particular (co-)evolution step of the PPU. Our evaluation of these rules shows that they support all co-evolutions of the pick&place unit evolution scenarios. Furthermore, the rules significantly reduce the amount of required model transformation applications to realize the co-evolution compared to the usual visual editing operations and to atomic model changes. Obviously, although developed in terms of the PPU case study, the intra- and inter-model transformation rules provided by this paper are case study-specific but may be used as a basis for consistently co-evolving architectural models and fault of any other system. We make the all transformation rules, (meta-)models and analyzed data online available such that they can be implemented as a case study [7].

In our future research, we plan to extend the analysis to different models and type of models as well as also to investigate further co-evolution scenarios for similar systems. We believe that the results presented in this paper can be generalized, however a careful investigation is needed. Based on the results of the co-evolution of architecture models and quality evaluation models, the next step is to provide methods and tools support for efficient evaluation of co-evolving quality evaluation models. The goal is to provide continual verification of the system for each evolution step at design time and at run time.

The current approach only supports the developer by providing the transformations to evolve and co-evolve software architecture models and fault trees. However, it does not support the developer which transformation to use, particularly, which evolution of one model should be applied after a change in the other model. We plan to use the presented transformations and the developed tooling to analyze historical co-evolution behavior, i.e., the types, order, and applications of transformations, to predict or prioritize the application of transformations in one model after a change in the other model. Early positive results evaluating techniques from artificial intelligence for that purpose encourage us in further working in that direction.

References

[1] ISO/IEC 9126, Software engineering - Product quality - Part 4, Quality

in Use, 2001.

[2] Henshin Trace Metamodel, http://wiki.eclipse.org/Henshin_Trace_Model, last access: Dec 2016.

[3] Rasmus Adler, Marc Förster, and Mario Trapp. Determining Configuration Probabilities of Safety-Critical Adaptive Systems. In *21st International Conference on Advanced Information Networking and Applications (AINA 2007)*, pages 548–555. IEEE Computer Society, 2007.

[4] S. Amari, G. Dill, and E. Howald. A new approach to solve dynamic fault trees. In *Reliability and Maintainability Symposium, 2003. Annual*, pages 374–379, 2003.

[5] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In *13th International Conference-Model Driven Engineering on Languages and Systems, MODELS 2010, Proceedings, Part I*, pages 121–135, 2010.

[6] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[7] Tolga Ayav and Hasan Sözer. Identifying critical architectural components with spectral analysis of fault trees. *Appl. Soft Comput.*, 49:1270–1282, 2016.

[8] Joanne Bechta-Dugan, Salvatore Bavuso, and Mark Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–77, September 1992.

[9] Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. *Software & Systems Modeling*, 11(3):431–461, 2012.

[10] Andrea Bondavalli, Istvan Majzik, and Ivan Mura. Automated Dependability Analysis of UML Designs. *IEEE International Symposium on Object-oriented Real-time distributed Computing*, 2, 1999.

[11] J.-L. Boulanger and Van Quang Dao. Experiences from a model-based methodology for embedded electronic software in automobile. pages 1–6, April 2008.

[12] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The compass approach: Correctness, modelling and performability of aerospace systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security, 28th International Conference, SAFECOMP 2009 Proceedings*, volume 5775 of *LNCS*, pages 173–186. Springer, 2009.

[13] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 54(5):754–775, 2011.

[14] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2012.

[15] M. Bretschneider, H. J. Holberg, E. Bode, and I. Bruckner. Model-based safety analysis of a flap control system. *14th Annual INCOSE Symposium*, 2004.

[16] CENELEC EN 50126,128,129. CENELEC (European Committee for Electro-technical Standardisation) : Railway Applications – the specification and demonstration of Reliability, Availability, Maintainability and Safety, Railway Applications – Software for Railway Control and Protection Systems, Brussels, 2000.

[17] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Jtl: A bidirectional and change propagating transformation language. In *Software Language Engineering - Third International Conference, SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 2010.

[18] Tadeusz Cichocki and Janusz Górski. Failure mode and effect analysis for safety-critical systems with software components. In Floor Koornneef and Meine van der Meulen, editors, *Computer Safety, Reliability and Security, 19th International Conference, SAFECOMP 2000*, volume 1943 of *Lecture Notes in Computer Science*, pages 382–394. Springer, 2000.

[19] Tadeusz Cichocki and Janusz Górski. Formal support for fault modelling and analysis. In Udo Voges, editor, *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001*, volume 2187 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 2001.

[20] Pierre David, Vincent Idasiak, and Frederic Kratz. Towards a Better Interaction Between Design and Dependability Analysis: FMEA Derived From UML/SysML Models. In *Safety, Reliability and Risk Analysis: Theory, Methods and Applications*, pages 2259–2266, Jan. 2008.

[21] M. A. de Miguel, J. F. Briones, J. P. Silva, and A. Alonso. Integration of safety analysis in model-driven software development. *Software, IET*, 2(3):260–280, June 2008.

[22] Josh Dehlinger and Joanne Bechta Dugan. Analyzing dynamic fault trees derived from model-based system architectures. *Nuclear Engineering and Technology: An International Journal of the Korean Nuclear Society*, 40(5):365–374, 2008.

[23] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *International Conference on Model Driven Engineering Languages and Systems*, pages 287–303. Springer, 2013.

[24] Dominik Domis and Mario Trapp. Integrating Safety Analyses and Component-Based Design. In *"Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2008*, pages 58–71, 2008.

[25] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and evaluating choices for fixing inconsistencies in uml design models. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 99–108. IEEE, 2008.

[26] Jonas Elmqvist and Simin Nadjm-Tehrani. Safety-Oriented Design of Component Assemblies using Safety Interfaces. *Formal Aspects of Component Software*, pages 57–72, 2007.

[27] Anthony Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on*, 20(8):569 –578, aug 1994.

[28] P. Ganesh and J.B. Dugan. Automatic Synthesis of Dynamic Fault Trees from UML SystemModels. *13th International Symposium on Software Reliability Engineering (ISSRE)*, 2002.

[29] Sinem Getir, Lars Grunske, Christian Karl Bernasko, Verena Käfer, Tim Sanwald, and Matthias Tichy. Cowolf - A generic framework for multi-view co-evolution and evaluation of models. In *Theory and Practice of Model Transformations, 8th International Conference, ICMT 2015, Held as Part of STAF 2015*, pages 34–40, 2015.

[30] Sinem Getir, André van Hoorn, Lars Grunske, and Matthias Tichy. Co-evolution of software architecture and fault tree models: An explorative case study on a pick and place factory automation system. In *5th International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013 (NIM-ALP 2013)*, volume 1074 of *CEUR Workshop Proceedings*, pages 32–40, 2013.

[31] Holger Giese and Matthias Tichy. Component-based hazard analysis: Optimal designs, product lines, and online-reconfiguration. In *SAFECOMP 2006*, volume 4166 of *LNCS*, pages 156–169. Springer, 2006.

[32] Holger Giese, Matthias Tichy, and Daniela Schilling. Compositional hazard analysis of uml component and deployment models. In Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, editors, *Computer Safety, Reliability, and Security, 23rd International Conference, SAFECOMP 2004*, volume 3219 of *LNCS*, pages 166–179. Springer, 2004.

[33] Ursula Goltz, Ralf H Reussner, Michael Goedicke, Wilhelm Hasselbring, Lukas Märtin, and Birgit Vogel-Heuser. Design for future: managed software evolution. *Computer Science-Research and Development*, 30(3-4):321–331, 2015.

[34] Joel Greenyer and Ekkart Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[35] Lars Grunske. Towards an Integration of Standard Component-Based Safety Evaluation Techniques with SaveCCM. In Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner, editors, *Second Int. Conf. on Quality of Software Architectures, QoSA 2006,*, volume 4214 of *LNCS*, pages 199–213. Springer, 2006.

[36] Lars Grunske, Robert Colvin, and Kirsten Winter. Probabilistic model-checking support for FMEA. In *Fourth International Conference on the Quantitative Evaluaiton of Systems (QEST 2007)*, pages 119–128. IEEE Computer Society, 2007.

[37] Lars Grunske and Jun Han. A comparative study into architecture-based safety evaluation methodologies using AADL's error annex and failure propagation models. In *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008*, pages 283–292. IEEE Computer Society, 2008.

[38] Lars Grunske and Bernhard Kaiser. Automatic generation of analyzable failure propagation models from component-level failure annotations. In *Fifth International Conference on Quality Software (QSIC 2005*, pages 117–123. IEEE, 2005.

[39] Lars Grunske, Bernhard Kaiser, and Yiannis Papadopoulos. Model-driven safety evaluation with state-event-based component failure annotations. In *8th Int. Symp. on Component-Based Software Engineering, CBSE 2005, Proc.*, pages 33–48, 2005.

[40] Matthias Güdemann and Frank Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *12th IEEE High Assurance Systems Engineering Symposium, HASE 2010, San Jose, CA, USA, November 3-4, 2010*, pages 132–141. IEEE Computer Society, 2010.

[41] M. Gündemann, F. Ortmeier, and W. Reif. Using Deductive Cause-Sequence Analysis (DCCA) with SCADE. *SAFECOMP 2007, LNCS 4680*, pages 465–478, 2007.

[42] Mats Per Erik Heimdahl, Yunja Choi, and Michael W. Whalen. Deviation analysis: A new use of model checking. *Automated Software Engineering*, 12(3):321–347, 2005.

[43] Markus Herrmannsdörfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In *Software Language Engineering*, pages 3–22. Springer, 2010.

[44] IEC61508. International Standard IEC 61508, 1998. International Electrotechnical Commission (IEC).

[45] ISO 26262. ISO 26262 Road vehicles and Functional Safety, 2009.

[46] Anjali Joshi, Steve Vestal, and Pam Binns. Automatic Generation of Static Fault Trees from AADL Models. In *DSN Workshop on Architecting Dependable Systems*, Lecture Notes in Computer Science. Springer, 2007.

[47] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, Revised Selected Papers*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.

[48] Bernhard Kaiser. *State/Event Fault Trees: A Safety and Reliability Analysis Technique for Software-Controlled Systems.* PhD thesis, Technische Universität Kaiserslautern, Fachbereich Informatik, 2005.

[49] Bernhard Kaiser, Catharina Gramlich, and Marc Förster. State/event fault trees–A safety analysis model for software-controlled systems. *Reliability Engineering & System Safety*, 92(11):1521 – 1537, 2007. SAFECOMP 2004, the 23rd International Conference on Computer Safety, Reliability and Security.

[50] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. A new component concept for fault trees. In *SCS '03: Proceedings of the 8th Australian workshop on Safety critical systems and software*, pages 37–46, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[51] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. Understanding model evolution through semantically lifting model differences

with silift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641. IEEE, 2012.

[52] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. Adaptability of model comparison tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 306–309. ACM, 2012.

[53] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172. IEEE Computer Society, 2011.

[54] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-preserving edit scripts in model versioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 191–201. IEEE, 2013.

[55] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. Automatically deriving the specification of model editing operations from meta-models. In *Theory and Practice of Model Transformations - 9th International Conference, ICMT 2016*, volume 9765 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2016.

[56] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE, 2009.

[57] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. A posteriori operation detection in evolving software models. *Journal of Systems and Software*, 86(2):551–566, 2013.

[58] Christoph Legat, Jens Folmer, and Birgit Vogel-Heuser. Evolution in industrial plant automation: A case study. In *Proc. IECON 2013*, pages 4386–4391. IEEE, 2013.

[59] Michael Lipaczewski, Simon Struck, and Frank Ortmeier. Using tool-supported model based safety analysis - progress and experiences in saml development. In *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012*, pages 159–166. IEEE Computer Society, 2012.

[60] István Madari, László Angyal, and László Lengyel. Traceability-based incremental model synchronization. *WSEAS Transactions on Computers*, 8(10):1691–1700, 2009.

[61] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proc. IWPSE 2005*, pages 13–22. IEEE Computer Society, 2005.

[62] Vukasin Milovanovic and Dragan Milicev. An interactive tool for uml class model evolution in database applications. *Software & Systems Modeling*, 14(3):1273–1295, 2015.

[63] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering 2003*, pages 455–464. IEEE Computer Society, 2003.

[64] Object Management Group (OMG). MOF 2.0 QVT Final Adopted Specification v1.1, 2011. OMG Adopted Specification formal/2011-01-01.

[65] Y. Papadopoulos and M. Maruhn. Model-Based Automated Synthesis of Fault Trees from Matlab.Simulink Models. *International Conference on Dependable Systems and Networks*, 2001.

[66] Y. Papadopoulos, J. A. McDermid, R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Int. Journal of Reliability Engineering and System Safety*, 71(3):229–247, 2001.

[67] Y. Papadopoulos, D. Parker, and C. Grante. Automating the failure modes and effects analysis of safety critical systems. In *Int. Symp. on High-Assurance Systems Engineering (HASE 2004)*, pages 310–311. IEEE Comp. Society, 2004.

[68] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. Timed hazard analysis of self-healing systems. In *ASAS*, volume 7740 of *LNCS*, pages 112–151. Springer, 2013.

[69] Andrew Rae and Peter Lindsay. A behaviour-based method for fault tree generation. *Proceedings of the 22nd International System Safety Conference*, pages 289 – 298, 2004.

[70] Arend Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.

[71] Michaela Rindt, Timo Kehrer, and Udo Kelter. Automatic generation of consistency-preserving edit operations for mde tools. In *Proceedings of the Demonstrations Track of MoDELS 2014*, volume 1255 of *CEUR Workshop Proceedings*, 2014.

[72] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Grah Transformation, Volume 1: Foundations*. World Scientific, 1997.

[73] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. A System Dependability Modeling Framework Using AADL and GSPNs. In *Architecting Dependable Systems IV*, volume 4615 of *LNCS*, pages 14–38. Springer, 2007.

[74] Thomas Ruhroth and Heike Wehrheim. Model evolution and refinement. *Science of Computer Programming*, 77(3):270–289, 2012.

[75] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. What is needed for managing co-evolution in MDE? In *Proc. IWMCP 2011*, pages 30–38. ACM, 2011.

[76] Andy Schürr. Specification of graph translators with triple graph grammars. In 20^{th} *Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer Verlag, 1994.

[77] Andy Schürr and Arend Rensink. Software and systems modeling with graph transformations theme issue of the journal on software and systems modeling. *Software and System Modeling*, 13(1):171–172, 2014.

[78] Guy Shani and Asela Gunawardana. Evaluating recommendation systems. *Recommender systems handbook*, pages 257–297, 2011.

[79] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2. edition, 2009.

[80] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for emf model transformation development. In *International Conference on Graph Transformation*, pages 196–208. Springer, 2017.

[81] G. Szabo and G. Ternai. Automatic Fault Tree Generation as a Support for Safety Studies of Railway Interlocking Systems. *IFAC Symposium on Control in Transportation Systems*, 2009.

[82] Gabriele Taentzer, Florian Mantz, and Yngve Lamo. Co-transformation of graphs and type graphs with application to model co-evolution. In *Proc. ICGT 2012*, pages 326–340, 2012.

[83] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, Inc., 2009.

[84] Laurence Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–124, 2008.

[85] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, U.S. Nuclear Regulatory Commission, NUREG–0492, 1981.

[86] Sven Wenzel. Scalable visualization of model differences. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 41–46. ACM, 2008.

[87] Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 336–352. Springer, 2012.

[88] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, USA, 2000.

[89] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. Statistical analysis of changes for synthesizing realistic test models. In *Software Engineering*, pages 225–238, 2013.

[90] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. Synthesizing realistic test models. *Computer Science-Research and Development*, pages 1–23, 2014.

Appendix A. Detailed Description of PPU Evolution Scenarios

Each description of the scenarios SC0–SC14 starts with a general description followed by a description of the related changes in the SA and FT models. Note that our goal is not to perform a complete hazard analysis in each scenario to assess the safety of the system. We are only interested in the identification of the relations between the evolution of the different models. Figure 5 and Figures A.13–A.15 depict the SA and FT instances summarizing changes from different scenarios. For our SA language, a graphical concrete syntax is used, which is similar to SysML Composite Structure Diagrams. The component instances are labeled with a combination of identifier and component type name (e.g., *stackS:TactileDigital*), as well as a stereotype indicating the component type meta-class (≪*Sensor*≫).

SC0 — Initial Situation. In the initial scenario, the PPU consists of a stack, a crane, and a slide. The stack includes a separator that pushes a WP to a position from where it is picked up by the crane (using a vacuum). The crane places the WP at a slide, which serves as the output storage. The PPU includes nine sensors (all tactile digital): in the stack, one sensor detects the presence of a WP at the pick up position and two sensors detect whether the separator is extracted or retracted; in the crane, four sensors detect the crane position and two sensors detect whether the crane's cylinder is up or down. In this scenario, the PPU processes only one kind of WPs (metallic).

Figure 5(a) includes the decomposition of the PPU into three top-level component instances for stack, crane, and slide (depicted as part of the sorter introduced in SC10)—with a dedicated component type for each. The stack and the crane are further decomposed according to the afore-mentioned information about this scenario, including the software components responsible

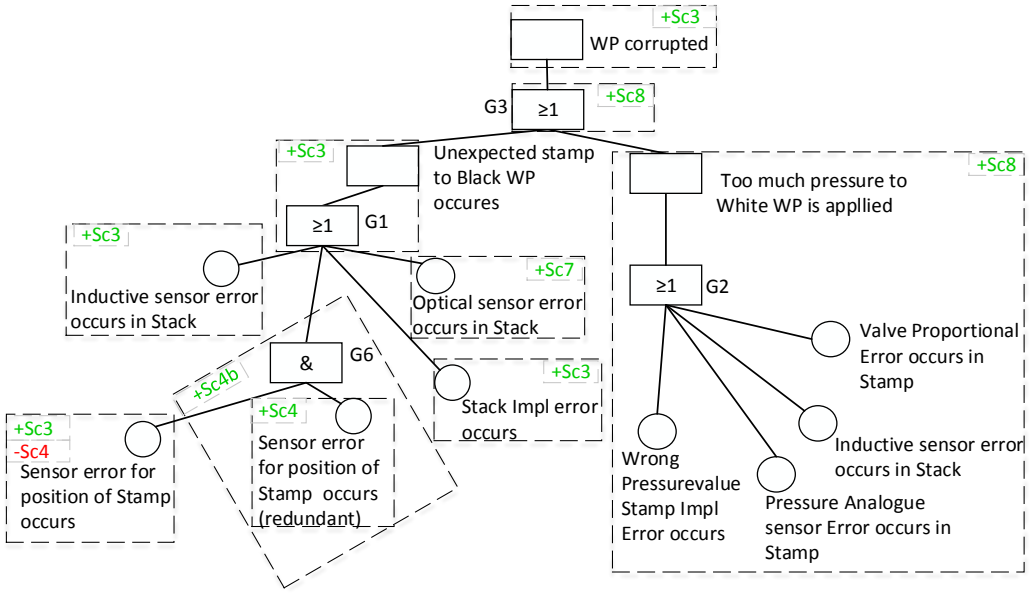


Figure A.13: FT for the hazard that a WP is corrupted (FT2)

for their control. Note that both sensors and the software components share the same respective type: a type for tactile digital sensors and one for software building blocks. Without the loss of generality, we model the software components to have a common type.

With respect to safety, the FT model for this scenario includes five error types (software implementation error, sensor error, timing and general vacuum errors, and external error), three failure types (position failure, timing failure, exceeded capacity), as well as respective failure (four) and error (eight) instances for the respective component instances. Figure 5(b) shows an FT, referred to as FT1, for the hazard that a WP is outside the system. Failure and error instances are related to component instances by generic trace elements.

SC2 — Black Plastic WPs. A sensor (inductive digital) is added to the stack, which—together with the existing tactile digital sensor—allows to distinguish metallic WPs from black plastic WPs introduced in this scenario. In the SA model, this leads to an addition of a new component type (for inductive digital sensors) and a component instance of this type as subcomponent of the stack. With respect to safety, no changes to the failure model

and the FT appear, as the two types of WPs are not handled differently, so far.

SC3 — Stamp Module Added. A stamp is added, including a magazine, a cylinder, and four sensors (tactile digital). The magazine moves a WP to/from the stamp position; the cylinder does the actual stamping by moving down, pressing, and retracting. Two of the sensors are used for the magazine; the remaining two for the cylinder. An additional tactile digital sensor is added to the crane in order to detect when it is at the position of the stamp. Only metallic WPs are stamped. The SA model is changed at two places. First, a new sensor component instance (existing type) is added to the crane. Second, a new top-level component instance for the stamp (along with the addition of a new component type), including component instances for the software (existing type), magazine (including a new type), cylinder (existing type), and the four sensors (existing type) are added. With respect to safety, six error instances (existing error types) for sensors are added: five for the sensors introduced in this scenario and another for the sensor added in SC2, which is used now. A failure instance and a corresponding failure type are added for the event that a wrong WP is stamped. This scenario also introduces a new hazard: WPs may get corrupted. Therefore, we created a second FT, referred to as FT2, which includes three basic events—a sensor error in the stack as well as a sensor and an implementation error in the stack—and an OR gate leading to an intermediate event for pressing wrong WPs. FT2 is shown in Figure A.13.

SC4 — Inductive Sensors for Crane Positioning. Each of the five tactile digital crane positioning sensors are replaced by inductive digital sensors, which are more robust against pollution. In the SA model, this changes the component type of the component instances for the crane sensors. With respect to safety, new five basic events are replaced by previous ones which correspond to added and deleted sensors in the crane respectively in FT1 (same for error instances). FT2 remains unchanged.

SC4b — Increase Reliability of Crane Positioning. As a variant of SC4 with redundancy being introduced, the new inductive sensors are added but the existing tactile sensors remain (being spatially shifted). In the SA model, this scenario leads to the addition of five sensors as subcomponents of the crane (component instances with existing component type). With respect to safety, five error instances (existing type) are added to the failure model for the new sensors. In FT1, new basic events are added for the sensor errors. Five AND gates ($G3$–7 in Figure 5(b)) are added, each having two

basic events as input and leading to the already-existing OR gate ($G8$). Note that the following scenarios are not based on this one but on SC4.

SC7—Additional White WPs. In order to support newly introduced white WPs, a new optical digital sensor is added to the stack. White WPs are stamped. In the SA model, the new sensor is added as a new component instance of the stack, including a new type for the optical digital sensor. The controller logics of the stack is changed to incorporate the kind of WPs. With respect to safety, a new error instance (existing error type) is introduced for the new sensor. A basic event for the sensor error is added to FT2 as input to an existing intermediate event as output of an existing OR gate.

SC8—Different Pressure Profiles. This scenario introduces two additional components to the stamp, in order to support stamping with different pressure profiles: a proportional valve and an analogue pressure sensor. White WPs are stamped with less pressure than metallic WPs. Changes to the SA model are the addition of subcomponent instances (proportional valve and analogue pressure sensor) to the stamp (including types) and changes to the stamp's controller logic (software). In the failure model, new error instances are added for the stamp's controller (existing error type), as well as for errors of the valve (new error type for actuator errors) and the sensor (existing error type). A new failure instance (existing type) is added for the event that too much pressure is put to white WPs. In FT2, four new basic events are added: two for sensor errors (the stack's WP sensor and the stamp's pressure sensor), and others for errors in the valve and the stamp's controller logic. These new basic events lead to a new intermediate event (referring to the created failure instance) via a new OR gate.

SC9—Installation of Sorter. A conveyor is added to the PPU, which uses a belt to transport WPs to the slide—now located at the end of the belt. Conveyor and slide are now referred to as the sorter. Changes to the SA model are the creation of a new top-level component for the sorter, including the conveyor and the slide—which previously was a top-level component—as subcomponents. With respect to safety, an error type for the belt material corruption and two corresponding error instances for the belt to become slack or time-worn, respectively, are added. One failure instance along with a new failure type for speed failures of the belt is added: belt too fast. Basic events for each new error instance, an intermediate event for the new failure instance, and two OR gates ($G1$, $G12$) are added to FT1.

SC10—Additional Slides and Pushers. Two additional slides are added to the sorter at both sides of the conveyor's belt to increase the PPU's

output storage capacity. Pushers are pushing the WPs into the slides. Two optical digital sensors are used to detect WPs. The SA model is changed by adding two additional slides, the two pushers, and the two sensors as subcomponent instances (new type for the pushers) of the sorter component. With respect to safety, two error instances of existing type (external cause for exceeded slide capacity, sensor error for WP detection), and a failure instance of existing type (timing failure for the pushers) are added for both slides. Also for both slides, FT1 is extended by two intermediate events referring to the new failure instances, as a result of two OR-connected (new Gates $G9$, $G10$) occurrences of the basic events.

SC11—Specific Order of Work Pieces. To sort the WPs in a specific order, two inductive sensors are installed at the slides on both sides of the belt to detect the kind of work pieces. In this case, the software orders white, metal, and black WPs respectively. The SA model is changed by adding two inductive sensors to the sorter component. With respect to safety, the addition of the new sensors leads to the creation of four basic events with new error instances. These basic events lead to a new intermediate event together with dependent three basic events regarding the belt. This eventually causes a creation of a failure instance associated to a new intermediate event. Finally an OR gate (G14) is added to operate the given basic events as an output of the above intermediate event. The SA and FT changes for this and the following scenarios are included in Figures A.14 and A.15.

SC13—Potentiometer at the Crane. The crane's five inductive digital positioning sensors are replaced by a single potentiometer to increase the accuracy and to avoid spending cables and terminal blocks. In the SA model, the five component instances for the positioning sensors are removed and the potentiometer is added as a new component instance (along with an introduction of the type). With respect to safety, the error instances, basic events, and the gate for the removed sensors (FT1) are removed. For the potentiometer, a new error instance (existing type) is added to the failure model. To FT1, a corresponding basic event is added as replacement for the OR gate $G8$ and the connected basic events.

SC14—Incremental Encoder at the Crane. The crane's potentiometer is replaced by an incremental encoder to increase the resistance to electromagnetic influences. Changes to the SA model are the addition of the new component type for the incremental encoder and its use for the positioning sensor (potentiometer introduced in SC13). With respect to safety, the basic event and error instance corresponding to the incremental encoder

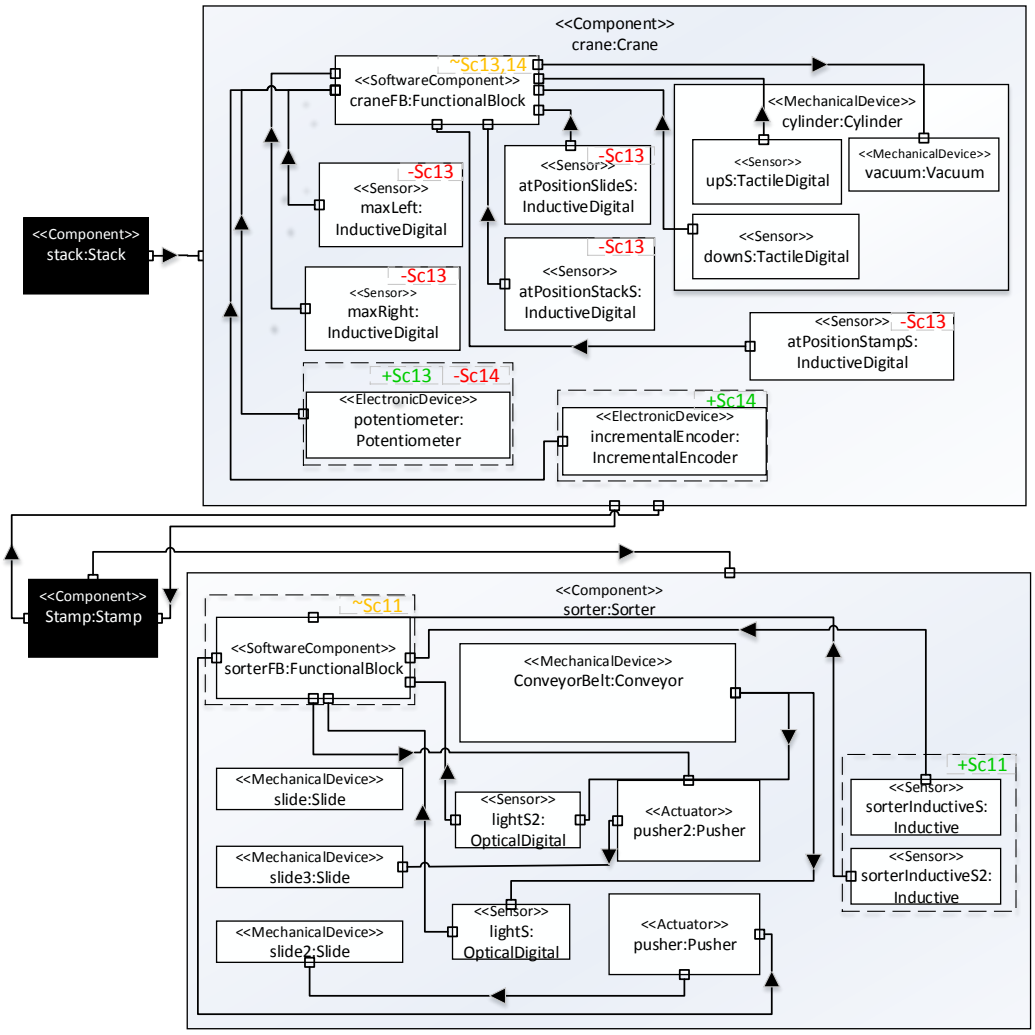


Figure A.14: Partial SA component model for the PPU scenarios 11,13,14

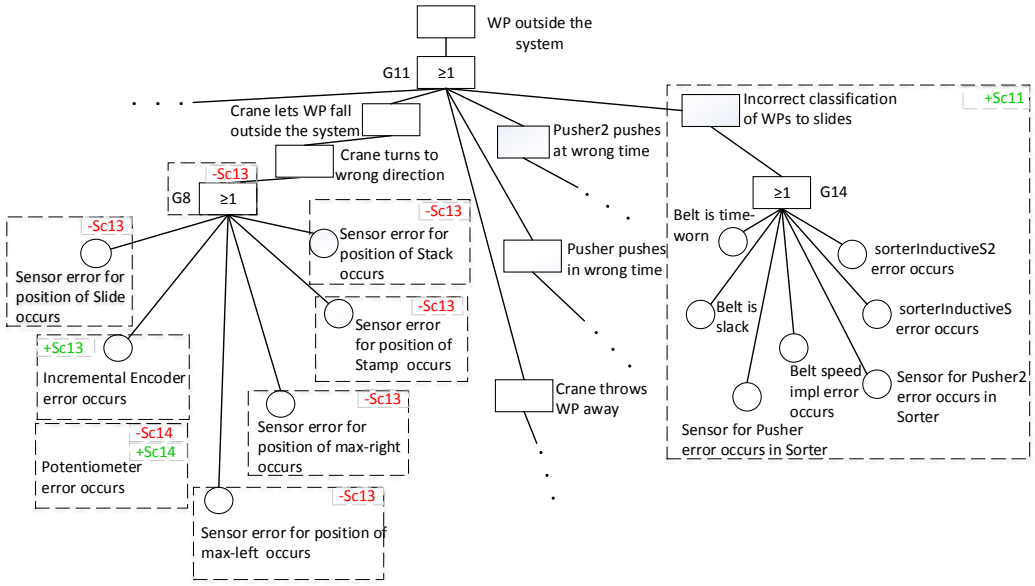


Figure A.15: Partial FT model (FT1) with changes only for the scenarios 11,13,14

is replaced by the new basic event and error instance corresponding to the potentiometer in FT1.

Related Papers Written by the Authors

Co-Evolution of Software Architecture and Fault Tree Models: An Explorative Case Study on a Pick and Place Factory Automation System

Sinem Getir[1], André van Hoorn[1], Lars Grunske[1], and Matthias Tichy[2]

[1] Reliable Software Systems, University of Stuttgart, Germany,
[2] Software Engineering Division, Chalmers | University of Gothenburg, Sweden

Abstract. Safety-critical systems are subject to rigorous safety analyses, e.g., hazard analyses. Fault trees are a deductive technique to derive the combination of faults which cause a hazard. There is a tight relationship between fault trees and system architecture as the components contain the faults and the component structure influences the fault combinations. In this paper, we describe an explorative case study on multiple evolution scenarios of a factory automation system. We report on the evolution steps on the system architecture models and fault trees and how the evolution steps in the different models relate to each other.

1 Introduction

Safety-critical systems require a rigorous assessment of the system's safety. Different techniques like Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA) are used to analyze the relations between failures of system parts and hazards, which are situations that might lead to accidents which harm life, health, property or the environment.

The outcome of hazard analysis techniques like FTA and FMEA are the corresponding safety evaluation models, e.g., fault trees, as well as improved and revised architectural and behavioral models. However, all these models are not totally independent but rather have a tight relation, e.g., the failures of an architectural component must be considered in the fault tree. Hence, the consistency of these models is of utmost importance since inconsistencies would lead to an incorrect safety evaluation which can lead to severe consequences. System evolution makes the consistency problem worse as not only at one point in time consistency between the models must be ensured but also after each evolution step as also noted as challenge for evolution in [7].

The overall goal of our work is to support the co-evolution of system architecture and fault tree models to ensure the consistency between those two models. We envision a model transformation based approach where incremental model transformations are used to evolve one model and co-evolve another model. Existing approaches (e.g., [3,5,6]), which consider both the system architecture and fault tree models, typically use manually or quasi-automatic generation of fault trees from architectural models with fault tree specific annotations. This only shifts the consistency problem inside a single model but does not solve it.

As a first step in this research, we analyzed a case study for the evolution of a factory automation system to identify the possible model changes, the relations between elements of the two models and the changes in the two models, as

well as where input from the user is required. The example is commonly used in the German priority program "Design for Future — Managed Software Evolution" and addresses a pick and place unit (PPU). The evolution scenarios on the architecture have been described in [4]. Factory automation systems are an interesting case for evolution since they contain mechanical parts, electrical parts, and software parts. All these parts can be evolved individually or in combination. Additionally, these systems are also typically safety-critical.

We developed architecture and fault tree models for a safety-relevant subset of the PPU evolution scenarios. This enabled us to study the evolution of the individual models as well as to study the relation between the individual evolution of the two models in order to understand which changes in one model affect changes in another model.

The models and detailed model changes for the selected evolution changes are the first contribution of this paper which enables other researchers to study co-evolution as well. The raw data is made available to the general public at [2]. The second contribution is the identification and generalization of the relations between the model changes as initial requirements for an approach to support the developer in the co-evolution of architecture and fault tree models.

The next section introduces the two modeling languages for software architecture and fault trees as well as the evolution scenarios of the PPU case system—including the individual evolution of the two models. Based on that, Section 3 describes the identified general evolution changes and the identified relations between the evolutions of the different models. Section 4 draws the conclusions and outlines future work.

2 Modeling Languages and PPU Case Study System

Section 2.1 introduces the two modeling languages used to express the two types of co-evolving models: system architecture and fault trees. Section 2.2 describes the pick and place unit (PPU) case study system, including the manually created—and individually evolving—models.

2.1 Modeling Languages

Due to space limitations, we provide only textual descriptions of the core concepts of both languages, referred to as *SA* and *FT*. In both cases, well-known concepts from architecture description languages (ADLs) [8] and fault tree modeling [9], respectively, are used.

The core entities provided by our software architecture (SA) language for describing system architectures are components, ports, and connectors. SA distinguishes between type and instance level for these elements. Component types can be further distinguished between hardware (electronic and mechanical) and software. Components may be composite structures of other interconnected components. SA also includes concepts for ports and connectors, which are omitted in this paper due to space limitations.

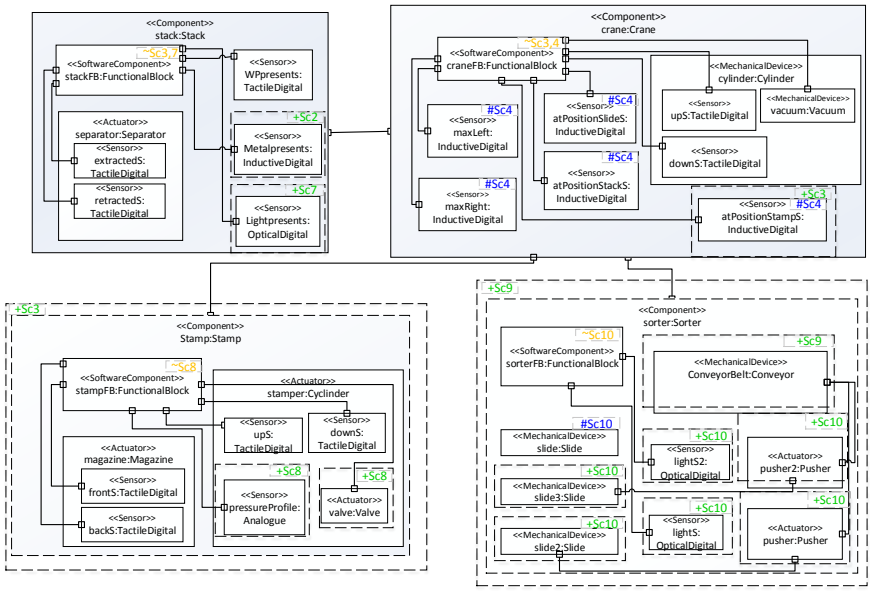
Our second modeling language FT allows the definition of a failure model and a set of corresponding fault trees. A failure model includes the definition of error types and failure types and their instances based on [1]. To exemplify the difference between instance- and type-level, a sensor error is an error type, while the error of a specific sensor is an error instance. The core (abstract) entities of a fault tree are events (hazard as top event, basic event relating to an error instance, and intermediate events) as well as boolean gates.

2.2 Case Study: Evolution Scenarios

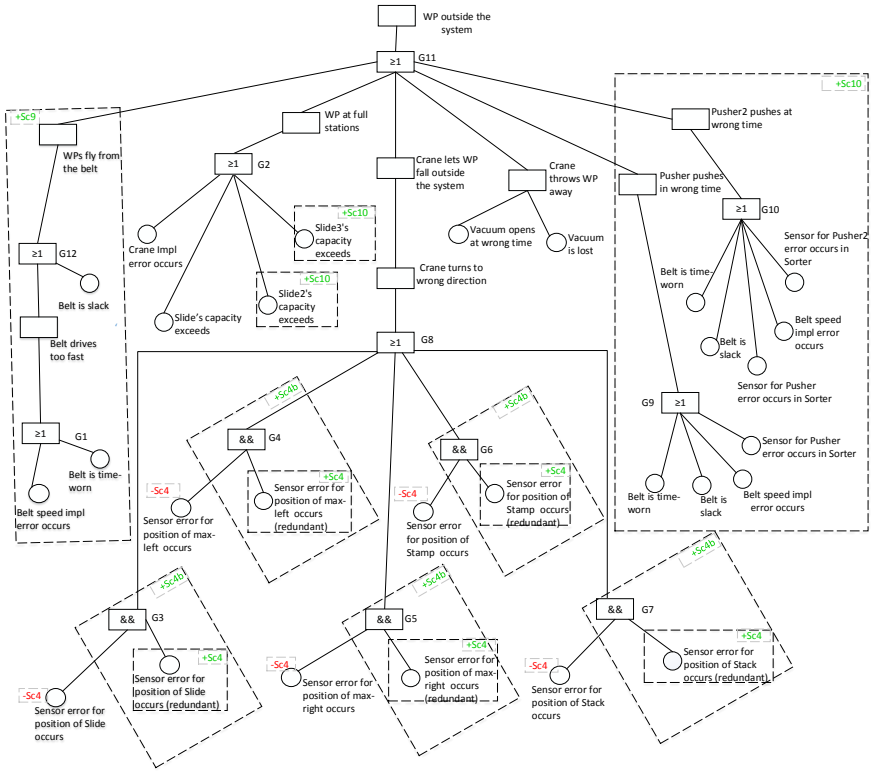
The case study system is a laboratory plant, called pick and place unit (PPU). The PPU mimics an industrial plant by moving so-called work pieces (WPs) between different working positions where they are stored or processed. Out of the 14 evolution scenarios that have been defined for the PPU [4], we selected a subset of eleven scenarios (0, 2, 3, 4, 4b, 7, 8, 9, 10, 13, 14) for our study that include system changes affecting the system's safety properties. For each scenario, we manually created SA and FT models. In this section, we will describe the different scenarios—limited to the safety-relevant aspects—and the changes they implied to the SA and FT models. Each scenario description starts with a general description followed by a description of the related changes in the SA and FT models. Note that our goal is not to perform a complete hazard analysis in each scenario to assess the safety of the system. We are only interested in the identification of the relations between the evolution of the different models. Figure 1 depicts the SA and FT instances as a combination of the scenarios which we will present in detail in the following. Due to space restrictions, we do not present the scenarios 13 and 14. For our SA language, a graphical concrete syntax is used, which is similar to UML2 composite structures. The component instances are labeled with a combination of identifier and component type name (e.g., *stackS:TactileDigital*), as well as a stereotype indicating the component type meta-class (≪*Sensor*≫). For the FT model, we use the usual notation [9].

SC0 — Initial Situation In the initial scenario, the PPU consists of a stack, a crane, and a slide. The stack includes a separator that pushes a WP to a position from where it is picked up by the crane (using a vacuum). The crane places the WP at a slide, which serves as the output storage. The PPU includes nine sensors (all tactile digital): in the stack, one sensor detects the presence of a WP at the pick up position and two sensors detect whether the separator is extracted or retracted; in the crane, four sensors detect the crane position and two sensors detect whether the crane's cylinder is up or down. In this scenario, the PPU processes only one kind of WPs (metallic).

Figure 1(a) includes the decomposition of the PPU into three top-level component instances for stack, crane, and slide (depicted as part of the sorter introduced in SC10)—with a dedicated component type for each. The stack and the crane are further decomposed according to the afore-mentioned information about this scenario, including the software components responsible for their control. Note that both the sensors and the software components share the same



(a) SA model of the PPU



(b) FT model for the hazard that a WP is outside the system (FT1)

Fig. 1. SA and FT models for the PPU scenarios SC0–10.
(Legend for change operations: + addition, - deletion, # replacement by other implementation, ∼ new version of implementation, && AND, ≥1 OR.)

respective type for simpler presentation: a type for tactile digital sensors and one for software building blocks.

With respect to safety, the FT model for this scenario includes five error types (software error, sensor error, timing and general vacuum errors, and external error), three failure types (position failure, timing failure, exceeded capacity), as well as respective failure (four) and error (eight) instances for the respective component instances. Figure 1(b) shows an FT, referred to as FT1, for the hazard that a WP gets outside the system.

SC2 — Black Plastic WPs A sensor (inductive digital) is added to the stack, which—together with the existing tactile digital sensor—allows to distinguish metallic WPs from black plastic WPs introduced in this scenario. In the SA model, this leads to an addition of a new component type (for inductive digital sensors) and a component instance of this type as subcomponent of the stack. With respect to safety, no changes to the failure model and the FT appear, as the two types of WPs are not handled differently, so far.

SC3 — Stamp Module Added A stamp is added, including a magazine, a cylinder, and four sensors (tactile digital). The magazine moves a WP to/from the stamp position; the cylinder does the actual stamping by moving down, pressing, and retracting. Two of the sensors are used for the magazine; the remaining two for the cylinder. An additional tactile digital sensor is added to the crane in order to detect when it is at the position of the stamp. Only metallic WPs are stamped. The SA model is changed at two places. First, a new sensor component instance (existing type) is added to the crane. Second, a new top-level component instance for the stamp (along with the addition of a new component type), including component instances for the software (existing type), magazine (including a new type), cylinder (existing type), and the four sensors (existing type) are added. With respect to safety, six error instances (existing error types) for sensors are added: five for the sensors introduced in this scenario and another for the sensor added in SC2, which is used now. A failure instance and a corresponding failure type are added for the event that a wrong WP is stamped. This scenario also introduces a new hazard: WPs may get corrupted. Therefore, we created a second FT, referred to as FT2, which includes three basic events—a sensor error in the stack as well as a sensor and an implementation error in the stack—and an OR gate leading to an intermediate event for pressing wrong WPs. A diagram for FT2 is not included due to space limitations.

SC4 — Inductive Sensors for Crane Positioning Each of the five tactile digital crane positioning sensors are replaced by inductive digital sensors, which are more robust against pollution. In the SA model, this changes the component type of the component instances for the crane sensors. With respect to safety, the probability of the five basic events in FT1 that one of the crane sensors fails is decreased. FT2 remains unchanged.

SC4b — Increase Reliability of Crane Positioning As a variant of SC4 with redundancy being introduced, the new inductive sensors are added but the existing sensors remain (being spatially shifted). In the SA model, this scenario

leads to the addition of five sensors as subcomponents of the crane (component instances with existing component type). With respect to safety, five error instances (existing type) are added to the failure model for the new sensors. In FT1, new basic events are added for the sensor errors. Five AND gates ($G3$–7 in Figure 1(b)) are added, each having two basic events as input and leading to the already-existing OR gate ($G8$). Note that the following scenarios are not based on this one but on SC4.

SC7—Additional White WPs In order to support newly introduced white WPs, a new optical digital sensor is added to the stack. White WPs are stamped. In the SA model, the new sensor is added as a new component instance of the stack, including a new type for the optical digital sensor. The controller logics of the stack is changed to incorporate the kind of WPs. With respect to safety, a new error instance (existing error type) is introduced for the new sensor. A basic event for the sensor error is added to FT2 as input to an existing intermediate event as output of an existing OR gate.

SC8—Different Pressure Profiles This scenario introduces two additional components to the stamp, in order to support stamping with different pressure profiles: a proportional valve and an analogue pressure sensor. White WPs are stamped with less pressure than metallic WPs. Changes to the SA model are the addition of subcomponent instances (proportional valve and analogue pressure sensor) to the stamp (including types) and changes to the stamp's controller logic (software). In the failure model, new error instances are added for the stamp's controller (existing error type), as well as for errors of the valve (new error type for actuator errors) and the sensor (existing error type). A new failure instance (existing type) is added for the event that too much pressure is put to white WPs. In FT2, four new basic events are added: two for sensor errors (the stack's WP sensor and the stamp's pressure sensor), and others for errors in the valve and the stamp's controller logic. These new basic events lead to a new intermediate event (referring to the created failure instance) via a new OR gate.

SC9—Installation of Sorter A conveyor is added to the PPU, which uses a belt to transport WPs to the slide—now located at the end of the belt. Conveyor and slide are now referred to as the sorter. Changes to the SA model are the creation of a new top-level component for the sorter, including the conveyor and the slide—which previously was a top-level component—as subcomponents. With respect to safety, an error type for the belt material corruption and two corresponding error instances for the belt to become slack or time-worn, respectively, are added. One failure instance along with a new failure type for speed failures of the belt is added: belt too fast. Basic events for each new error instance, an intermediate event for the new failure instance, and two OR gates ($G1$, $G12$) are added to FT1.

SC10—Additional Slides and Pushers Two additional slides are added to the sorter at both sides of the conveyor's belt to increase the PPU's output storage capacity. Pushers are pushing the WPs into the slides. Two optical digital sensors are used to detect WPs. The SA model is changed by adding two additional slides, the two pushers, and the two sensors as subcomponent instances

(new type for the pushers) of the sorter component. With respect to safety, two error instances of existing type (external cause for exceeded slide capacity, sensor error for WP detection), and a failure instance of existing type (timing failure for the pushers) are added for both sides. Also for both sides, FT1 is extended by two intermediate events referring to the new failure instances, as a result of two OR-connected (new Gates $G9$, $G10$) occurrences of the basic events.

3 Identified Relations Between Model Changes

In order to understand the relations between the changes in one model and changes in the other models presented in the previous section, we summarized the changes in Table 1. The table shows the individual changes of the architecture model in the rows and the changes on the failure model and the fault trees in the columns. The cells contain the scenario IDs. This means that in the given scenario a certain change in the architecture coincides with a certain change in the failure model and fault tree. We do not include ports and connections for simplicity and exclude the initial scenario.

	Failure model						Fault tree						
	+Error Type	+Error Instance	-Error Instance	+Failure Type	+Failure Instance	+Hazard	+Basic Event	-Basic Event	#Probability	+Gate	-Gate	+Intermediate Event	No change
+Component Type	8, 9	7, 8, 9, 13		3, 9	3, 9, 10	3	7, 8, 9, 13			3, 9, 10	13	3, 9, 10, 13	2, 3, 9, 14
+Component Instance	8	3, 4b, 7, 8, 9, 10, 13		3, 9	3, 9, 10	3	3, 4b, 7, 8, 9, 10, 13			3, 9, 10	13	3, 9, 10, 13	2, 3, 9
-Component Instance			13					13					
~Component Instance (SW)		3			7, 8		3, 7, 8			4b			3, 7, 10, 13
#Component Instance (type)									4,14				
No change		3			8		3, 8, 10			8		8	

Legend: + addition, - deletion, # replaced by other implementation, ~ new version of implementation (incl. new features)

Table 1. Mapping of scenarios and model changes

We made a couple of general observations from the results of the case study and building the aforementioned table (and its detailed version [2]). The creation of error instances in the components eventually leads to a basic event in the fault tree. However, this can be in the same scenario (SC8) or in different evolution scenarios (SC2 and SC3). Sometimes, changes in one model do not coincidence with changes in another model. The addition of components often triggers changes of failure model and fault trees only when the component is actually used in the system. In some scenarios, individual changes in the architecture results in individual changes in the fault trees. However, in other scenarios, only a set of changes in the architecture is related to a set of changes in the fault tree. There are changes in one model where the user needs to decide on the correct changes

in another model. For example, the addition of the pusher in SC10 triggers the addition of basic events related to errors of the belt which has not been changed in that scenario.

Hence, the main result of the case study is that there is no simple, straightforward co-evolution of system architecture and fault tree models that could be fully automated for all possible different co-evolution steps contained in the case study. Instead, user interaction is required for some of them, e.g., when to add a basic event to the fault tree for a new component as described above.

4 Conclusion and Future Work

We presented the results of a case study in the co-evolution of system architecture and fault trees based on the evolution scenarios presented in [4]. For a subset of the evolution scenarios, we, first, built fault trees for two exemplary hazards and, second, identified evolution changes in both architecture and fault tree models including in which way the evolution changes depend on each other as co-evolutions.

Threats to validity of our results are (1) the limits of our metamodel and instance models, (2) the models were built by ourselves, (3) the selected subset of scenarios and hazards, and (4) the result is based on only one case study.

Based on the identified evolution changes, we currently work on a tool-supported co-evolution approach that supports the developer if one model evolves to choose a consistent co-evolution of the other model.

Acknowledgements: This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

References

1. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
2. Getir, S., van Hoorn, A., Grunske, L., Tichy, M.: Supplementary material. `http://www.iste.uni-stuttgart.de/en/rss/projects/ensure/consistencysaft.html`
3. Grunske, L., Kaiser, B., Papadopoulos, Y.: Model-driven safety evaluation with state-event-based component failure annotations. In: CBSE 2005. pp. 33–48 (2005)
4. Legat, C., Folmer, J., Vogel-Heuser, B.: Evolution in industrial plant automation: A case study. In: Proc. of IECON 2013. IEEE (2013), to appear
5. Papadopoulos, Y., McDermid, J.A., Sasse, R., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Int. Journal of Reliability Engineering and System Safety 71(3), 229–247 (2001)
6. Priesterjahn, C., Steenken, D., Tichy, M.: Timed hazard analysis of self-healing systems. In: ASAS, LNCS, vol. 7740, pp. 112–151. Springer (2013)
7. Ruscio, D.D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proc. of IWMCP 2011. pp. 30–38. ACM (2011)
8. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory and Practice. John Wiley & Sons, Inc. (2009)
9. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault tree handbook. Tech. rep., U.S. Nuclear Regulatory Commission, NUREG–0492 (1981)

A Generic Framework for Analyzing Model Co-Evolution

Sinem Getir[1], Michaela Rindt[2] and Timo Kehrer[2]

[1]Reliable Software Systems, University of Stuttgart, Germany
sinem.getir@informatik.uni-stuttart.de
[2]Software Engineering Group, University of Siegen, Germany
{mrindt,kehrer}@informatik.uni-siegen.de

Abstract. Iterative development and changing requirements lead to continuously changing models. In particular, this leads to the problem of consistently co-evolving different views of a model-based system. Whenever one model undergoes changes, related models should evolve with respect to this change. Domain engineers are faced with the huge challenge to find proper co-evolution rules which can be finally used to assist developers in the co-evolution process. In this paper, we propose an approach to learn about co-evolution steps from a given co-evolution history using an extensive analysis framework. We describe our methodology and provide the results of a case study on the developed tool support.

Keywords: Model-driven engineering, model evolution, multi-view modeling, model co-evolution, model synchronization, model differencing

1 Introduction

The multi-view paradigm is a well-established methodology to manage complexity in the construction of large-scale software systems. In Model-driven Engineering (MDE), this paradigm leads to the concept of multi-view modeling; different modeling notations are used to describe different aspects such as structure, behavior, performance, reliability etc. of a system.

Iterative development and changing requirements lead to continuously changing models. Consequently, this entails the special challenge to consistently co-evolve different views of a system [12]. In practice, this challenge usually appears as a synchronization problem; different (sub-)models, each of them representing a dedicated view on the system, are usually edited independently of each other. This occurs if they are assigned to different developers or due to the fact that a developer concentrates on a single aspect at a specific point of time [13]. Thus, changes to one model must be propagated to all related models in order to keep the views synchronized and to avoid inconsistencies.

We assume a setting as shown by the bottom-left part of Figure 1, the terminology is partly adopted from related work on model synchronization and model co-evolution [5, 6]: A source model $M_{src,n}$ is related to a target model $M_{tgt,n}$ via traces. A source model is the model that undergoes changes and a target

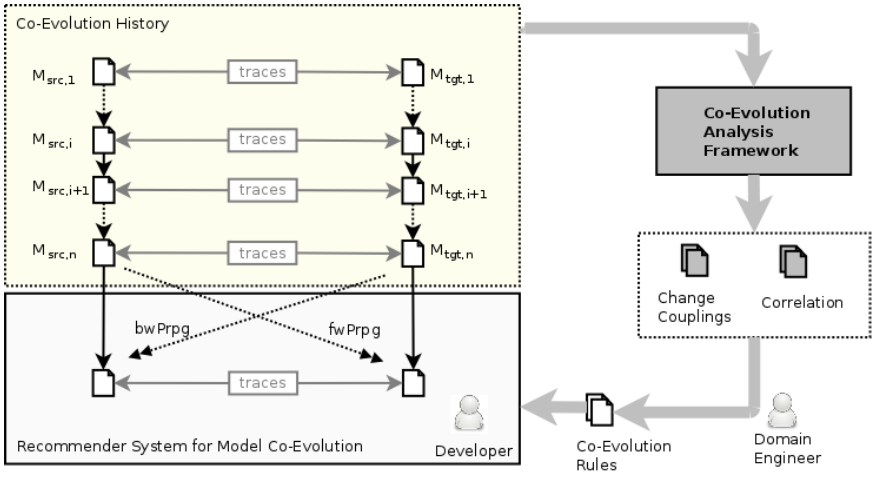


Fig. 1. Overview of the overall co-evolution process

model is the model to which these changes have to be propagated. Finally, a trace is a relationship between elements in these two different models. Forward propagation (*fwPrpg*) denotes the migration of the target model in response to changes occurring in the source model. Backward propagation (*bwPrpg*) denotes the migration of the source model in response to changes occurring in the target model. We refer to both kinds of propagations as *co-evolution steps*. From a technical point of view, co-evolution steps can be (semi-)automated via bidirectional model transformations. We call the transformation rules from which propagation rules can be derived as *co-evolution rules*.

However, due to the multitude of different modeling notations, the manual specification of co-evolution rules is a tedious and challenging task. Domain engineers, who have to find proper co-evolution rules, are faced with two essential questions: (1) Do certain changes on a source model correlate with changes on the target model? (2) If so, how are the changes coupled with each other? There are several domains for which no simple and straightforward co-evolution exists. The only viable solution is to pre-define possible co-evolution rules which can be offered to developers as possible options. For instance, this is the case for software architecture and quality of service models [4]. In Section 2, we introduce software architecture models and state charts as another example of co-evolving models which demonstrates the aforementioned research questions. We use the same example to serve as a running example throughout the paper.

This paper reports on our ongoing work on the semi-automated co-evolution of models of arbitrary source and target domains. The general process is illustrated by Figure 1. We propose to observe the co-evolution history in order to learn about developer decisions and to finally predict the co-evolution steps with a certain degree of probability. The more evolution steps are analyzed, the more accurate prediction results are expected. The contribution of this paper is the co-

evolution analysis framework which serves as a foundation for this co-evolution process. The analysis results can be used to generate co-evolution rules for a recommender system to interactively support model co-evolution. We describe our approach in Section 3. Tool support and early evaluation results which demonstrate the feasibility of our approach are briefly discussed in Section 4. Related work is analyzed in Section 5. We draw some conclusions and give an outlook on future work in Section 6.

2 Co-Evolution of Multi-View Models

Component diagrams and state charts are widely used notations to model structure and behavior in component-based software engineering. Intuitively, there are several relations between model elements of both views. For example, every state usually has a relation to a component, not necessarily the other way round. Transitions between states somehow reflect the interfaces and connections of the corresponding components in the component diagram. The hierarchy of composite states is expected to correspond to the hierarchical structure of components and their respective sub-components. Despite those rather intuitive relationships, consistently co-evolving both views is not a straight forward process, which is illustrated by the following example.

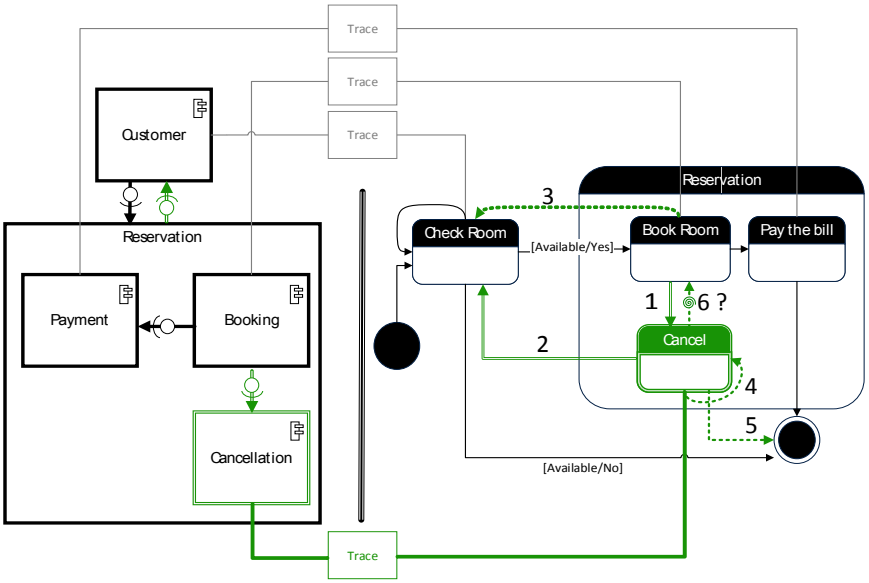


Fig. 2. Sample hotel reservation system modeled from two different viewpoints

Figure 2 shows a simple hotel reservation system modeled from two different viewpoints. The initial version of the system architecture consists of three

components, namely *Customer*, *Booking* and *Payment*. Relations between corresponding states and components are explicitly given by trace links. The system evolves at some point of time because it requires a new function to cancel a reservation process. In general, we assume that models are edited by means of a set of language-specific *edit operations*. An *edit step* invokes an edit operation and supplies appropriate actual parameters, which are also referred to as arguments. In our example, the revised version of the component model is obtained in three edit steps, namely the creation of the component *Cancellation* and two connectors. The new component and its connections to other components are highlighted in Figure 2 by doubled lines.

State chart elements printed in doubled lines indicate the developer's intention of co-evolution steps in response to the changes in the component diagram (1,2). We discuss several additional co-evolution steps which are possible on the state chart (M_{tgt}) in response to the changes in the component diagram (M_{src}). Note that these co-evolution steps are only assumptions which are based on domain knowledge, they are not meant to be a result of an empirical analysis.

Elements printed in dotted lines represent expected co-evolution steps which are, however, not intended by the user (3,4,5). Finally, a dotted line with spiral indicates an unexpected co-evolution step which is nonetheless intended by the user (6). We do not claim the set of possible options (1)-(6) to be complete. Nevertheless, it demonstrates the huge challenge of predicting the proper co-evolution steps:

- As the component *Cancellation* is added as a sub-component of reservation, a new state called *Cancel* is expected to be created as a sub-state of the corresponding composite state *Reservation*.
- The creation of transition (1) is expected due to the creation of port and interface relations of the corresponding components in the component model.
- Although there is no explicit relation between the components *Cancellation* and *Customer*, the creation of transition (2) is expected. Because the newly created relation between the composite component *Reservation* and the top-level component *Customer*, as a result of the creation of *Cancellation*. However, the new component may lead to an interaction between the components *Booking* and *Customer* indirectly via interfaces as well, therefore we should also consider the transition (3) with a small expectation.
- The required information for the proposed transitions (4) and (5) cannot be gathered from the component diagram. However, taking general state chart semantics into account, they can be presented to the developer as a possible option.
- Finally, we point out transition (6). The developer wants to create a loop between the states *Book Room* and *Cancel* which cannot be clearly anticipated from the component diagram since we observe only one direction for communication. Nonetheless, this option can be offered to the developer with a low probability.

We can conclude that each edit step on the component model may lead to many arbitrary co-evolution steps on the state chart. Some forward propagations

can be expected with a high probability based on the changes in the component diagram, others can only be offered as a set of possible choices.

3 Co-Evolution Analysis Framework

In Section 2, we have demonstrated a running example as a motivation of our analysis framework. We have presented possible co-evolution steps and observed that there are highly expected, less expected and unexpected changes for state charts when the component diagram evolves.

To study such changes and their relations, our co-evolution analysis framework takes a co-evolution history as illustrated by Figure 1 as input. Each pair of successive versions $i \rightarrow i+1$ from the given history is referred to as *evolution scenario* $ev_{i \rightarrow i+1}$. We assume that the co-evolution history includes consistent views for every evolution scenario. We further assume a model differencing engine to be available, which, given a set of possible edit operations for instances of a meta-model MM and successive model versions M_i and M_{i+1}, calculates a difference diff(M_i, M_{i+1}). A difference diff(M_i, M_{i+1}) is defined to be a partially ordered set of edit steps $s_1 \ldots s_k$. We finally offer two kinds of analysis functions; the *correlation analysis* is described in Section 3.1, the additional *coupling analysis* is presented in Section 3.2.

3.1 Correlation Analysis

We use the well-known Pearson correlation coefficient to assess the dependency between edit operations which are applicable to the source and target models. The basic processing steps of our correlation analysis are shown by Figure 3. For each evolution scenario $ev_{i \rightarrow i+1}$ of the co-evolution history, we first compute the differences diff($M_{src,i}$,$M_{src,i+1}$) and diff($M_{tgt,i}$,$M_{tgt,i+1}$). Subsequently, we count the edit steps contained by each of the obtained differences and group them by evolution scenarios and edit operations invoked by the respective edit steps. The sets of edit operations, which are available for instances of MM_{src} and MM_{tgt}, are given as additional input parameters of the correlation analysis.

Based on the calculated differences, we basically construct two matrices. For source model changes, we construct an e-×-s matrix where e denotes the number of evolution scenarios in the history (i.e., $e = n-1$), s denotes the number of edit operations available for instances of MM_{src}. A variable $a_{i,j}$ ($i \in \{1, ..., e\}$, $j \in \{1, ..., s\}$) represents the number of edit steps of type j (i.e. edit steps invoking edit operations represented by j, e.g. *createComponent* in our running example) in evolution scenario i. Analogously, an e-×-t matrix is being constructed for target model changes, where t denotes the number of edit operations available for instances of MM_{tgt}.

Let $X = \langle x_1, x_2, ..., x_e \rangle$ be a column vector of the e-×-s matrix, and $Y = \langle y_1, y_2, ..., y_e \rangle$ be a column vector of the e-×-t matrix. Then we can compute the Pearson correlation coefficient $r_{X,Y}$ for each combination of column vectors X and Y in order to quantify the linear relationship between edit operations that have been applied to the source and target models.

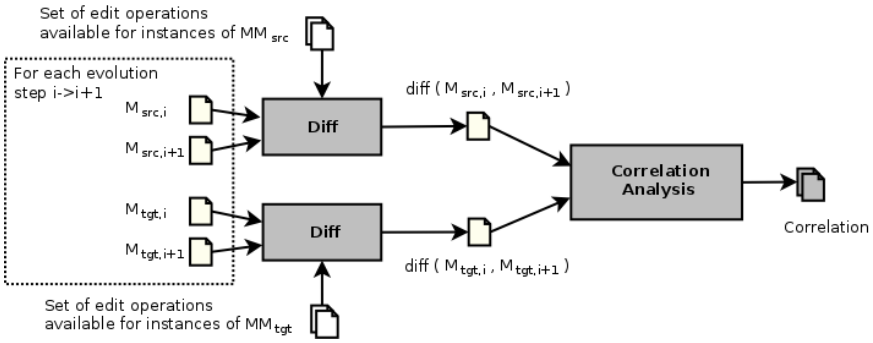


Fig. 3. Correlation analysis: basic proceeding, input and configuration parameter

3.2 Coupling Analysis

The correlation analysis has the advantage that it only requires the source and target models of each evolution scenario $ev_{i \to i+1}$. 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 edit operations does not imply that the respective edit steps are actually coupled. In other words, they can have a dependency by coincidence such that none of the involved arguments are actually related by a trace. Hence, we also provide a second analysis function which is capable of identifying *coupled changes*. Such an analysis can provide knowledge about user's modeling intentions enhancing correlation analysis results, for example learning of the loop intention by the user, as provided in Figure 2 with transition (6).

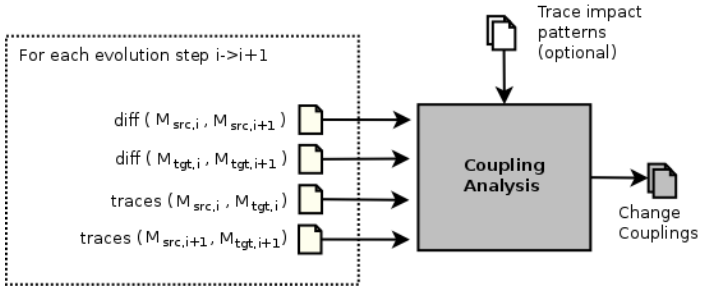


Fig. 4. Coupling analysis: basic proceeding, input and configuration parameter

In general, a coupled change identifies a pair of edit steps which have happened in the same evolution scenario. It also identifies the changed model elements which are connected (either directly or indirectly) to each other and were not just coincidentally changed in the same evolution scenario. We assume here that trace links identify related model elements of the source and target model.

These are, together with the model differences for each evolution scenario, provided as additional input parameters of the coupling analysis (see Figure 4).

Let $args(s)$ be the set of arguments of an edit step s. Basically, a pair of edit steps (s_{src}, s_{tgt}) is considered to be a coupled change, if we can find a pair of arguments (a_{src}, a_{tgt}), with $a_{src} \in args(s_{src})$ and $a_{tgt} \in args(s_{tgt})$, which are connected via a trace link.

Additionally, domain-specific *trace impact patterns* can be specified as optional inputs of the coupling analysis. These patterns allow to extend the search for coupled edit steps to the "neighborhood" of elements which are directly connected by a trace link. Consider for instance our running example shown in Figure 2. Here, trace links are only provided for related states and components. However, component connectors and state transitions are also to be considered as related if the connected components/states are related. This can be specified by a trace impact pattern as shown in Figure 5, i.e. the component connector labelled as *cc* and the state transition labelled as *st* are implicitly related. Consequently, a pair of edit steps modifying occurrences of *cc* and *st*, respectively, are to be considered as coupled.

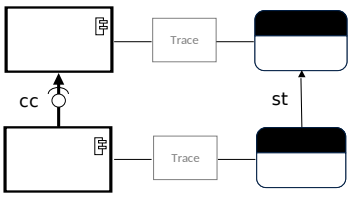


Fig. 5. Example of a trace impact pattern

Coupled changes are summarized over all evolution scenarios of the history as follows: We construct a s-$\times$-t matrix where s denotes the number of edit operations available for instances of MM_{src} and t denotes the number of edit operations available for instances of MM_{tgt}. A variable $a_{i,j}$ ($i \in \{1, ..., s\}$, $j \in \{1, ..., t\}$) is computed as the fraction of coupled edit steps of types i and j (i.e. edit steps invoking edit operations represented by i and j, respectively) with respect to all edit steps of type i being observed in the source model history.

4 Tool Support

We have prototypically implemented the analysis framework proposed in Section 3 on the widely used Eclipse Modeling Framework (EMF) and the model differencing engine SiLift [8, 9]. It is made available to the general public at the SiLift website[1] in order to enable other researchers to study the co-evolution of any EMF-based models.

Adaption of the generic framework. In order to adapt the generic framework to new modeling languages, i.e., to adapt it to a given source and target domain, one has to configure the SiLift differencing tool chain. Primarily, suitable edit operations for the source and target domain have to be provided. In SiLift, we use the model transformation language and system Henshin [1] to implement edit operations as declarative transformation rules, to which we refer to as *edit*

[1] http://pi.informatik.uni-siegen.de/Projekte/SiLift/coevolution.php

rules. Domain engineers can make use of the EMF-based meta-tool SERGe (SiDiff Edit Rule Generator) [10] in order to generate basic edit rules, which can be derived from MM_{src} and MM_{tgt}, respectively. Basic edit rules can be complemented by semantically rich complex edit rules such as refactoring operations. Typically, many complex edit rules can be composed of basic edit rules generated by SERGe.

Optionally, a set of trace impact patterns can be specified as additional input for the coupling analysis. Trace impact patterns are also specified in Henshin. We refer to these pattern specifications as *trace impact rules*. Trace impact rules do not implement in-place transformations, but serve as specifications of graph patterns which are to be found by the Henshin matching engine. Obviously, trace impact rules have to be specified manually by a domain engineer.

PPU Case Study. In order to demonstrate the feasibility of our approach, we have adapted the analysis framework to be used in the PPU(Pick and Place Unit) case study [11], which provides several evolution scenarios of a laboratory plant. In our previous work [4], we modeled each of the scenarios from two different viewpoints using two types of modeling languages: A simple architecture description language (SA) was used to model the system architecture, fault trees (FT) were used to model undesired system states and their possible causes.

All configuration artifacts which are needed to adapt the analysis framework to SA and FT models are available at the EnSure website[2]. In summary, we identified 82 edit rules available for FT models, 69 of them could be generated with SERGe. For SA models, we identified 42 suitable edit rules of which only one had to be specified manually, all other 41 edit rules could be generated with SERGe. In addition, we specified 6 trace impact rules serving as additional input of the coupling analysis. Consequently, we were able to automatically generate the results that have been produced by a manual analysis in our previous work [4].

5 Related work

Most approaches to model co-evolution address the migration of different types of MDE artifacts in response to meta-model adaptions. MDE artifacts which have to be migrated are, for example, instance models [7], model transformations [14], or syntactic and semantic constraints [3].

Only a few approaches address the evolution of multi-view models, which is most often considered as a model synchronization problem. Solutions are often based on the principle of bidirectional model transformations which are used to derive incremental change propagation rules, e.g., [5, 16, 6]. Among them, the approaches of Giese et al. [5] and Hermann et al. [6] are based on Triple Graph Grammars (TGGs). TGG rules describe correspondences between elements of source and target models together with the according forward and backward

[2] http://www.iste.uni-stuttgart.de/rss/projects/ensure/co-evolution

editing behavior. Bergmann et al. [2] present a novel type of model transformation to which they refer to as change-driven transformations. Change-driven transformations are directly triggered by complex model changes and thus can be utilized to specify sophisticated co-evolution patterns. A similar approach is presented by Wimmer et al. [15].

In contrast to our approach, TGG rules and change-driven transformations must be specified manually, whereas we intend to generate our co-evolution rules. In fact, we believe that existing approaches based on TGGs, change-driven transformations or similar techniques, can be also supported by our co-evolution analysis framework. Up to the best of our knowledge, we are not aware of any approach providing a framework to empirically study co-evolution by analyzing the history of co-evolving models.

6 Conclusion and Future Work

Many approaches for consistently co-evolving models and other related MDE artifacts have been proposed recently. Some are tailored to fixed source and target domains while others are more generic and adaptable.

However, correlation and coupling of changes has not been researched in-depth for many types of co-evolving (sub-)models. In order to close this research gap, we focus on establishing a co-evolution analysis framework to analyze the history of co-evolving models of arbitrary types. This will provide the foundation for synthesizing co-evolution rules in an automated way. Although the analysis framework still needs some configuration data as input, we conclude from the PPU case study that this adaption to a dedicated source and target domain can be done with moderate effort. Currently, the co-evolution rules which we finally intend to use as input of a co-evolution framework (see Figure 1) still have to be manually synthesized based on the information which is produced by the analysis framework. Larger case studies are needed to evaluate how far we can push the generation of co-evolution rules and how much training data is needed to derive appropriate co-evolution rules.

On the one hand, these co-evolution rules can be used to configure existing model synchronization frameworks in cases of domains where the co-evolution process can be fully automated. On the other hand, co-evolution rules serve as basis for a recommender system, which is able to handle semi-automated co-evolution of models. The latter one is subject to our future work.

Acknowledgments. This work is supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution. The authors would like to thank André van Hoorn, Matthias Tichy and Lars Grunske for the initial discussions and valuable reviews.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Intl. Conf. on Model Driven Engineering Languages and Systems, pp. 121–135 (2010)
2. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations - change (in) the rule to rule the change. Software and System Modeling 11(3), 431–461 (2012), http://dx.doi.org/10.1007/s10270-011-0197-9
3. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: Intl. Conf. on Model Driven Engineering Languages and Systems. pp. 287–303 (2013)
4. Getir, S., Van Hoorn, A., Grunske, L., Tichy, M.: Co-evolution of software architecture and fault tree models: An explorative case study on a pick and place factory automation system. In: Intl. Workshop on Non-functional Properties in Modeling: Analysis, Languages, Processes. pp. 32–40 (2013)
5. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Int. Conf. on Model Driven Engineering Languages and Systems, pp. 543–557 (2006)
6. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Software & Systems Modeling pp. 1–29 (2013)
7. Herrmannsdörfer, M., Wachsmuth, G.: Coupled evolution of software metamodels and models. In: Evolving Software Systems, pp. 33–63 (2014)
8. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Intl. Conf. on Automated Software Engineering. pp. 163–172 (2011)
9. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: Intl. Conf. on Automated Software Engineering. pp. 191–201 (2013)
10. Kehrer, T., Rindt, M., Pietsch, P., Kelter, U.: Generating edit operations for profiled UML models. In: Intl. Workshop on Models and Evolution. pp. 30–39 (2013)
11. Legat, C., Folmer, J., Vogel-Heuser, B.: Evolution in industrial plant automation: A case study. In: Proc. of IECON 2013. IEEE (2013)
12. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Intl. Workshop on Principles of Software Evolution. pp. 13–22 (2005)
13. Ruhroth, T., Gärtner, S., Bürger, J., Jürjens, J., Schneider, K.: Versioning and evolution requirements for model-based system development. In: Intl. Workshop on Comparison and Versioning of Software Models (2014)
14. Taentzer, G., Mantz, F., Lamo, Y.: Co-transformation of graphs and type graphs with application to model co-evolution. In: Graph Transformations, pp. 326–340. Springer (2012)
15. Wimmer, M., Moreno, N., Vallecillo, A.: Viewpoint co-evolution through coarse-grained changes and coupled transformations. In: Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings. pp. 336–352 (2012), http://dx.doi.org/10.1007/978-3-642-30561-0_23
16. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Synchronizing concurrent model updates based on bidirectional transformation. Software & Systems Modeling 12(1), 89–104 (2013)