The Journal of Systems and Software 142 (2018) 115–135

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

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] Humboldt-University of Berlin, Software Engineering, Germany
[b] University of Stuttgart, Institute of Software Technology, Germany
[c] Humboldt-University of Berlin, Model-Driven Software Development, Germany
[d] University of Ulm, Institute of Software Engineering and Programming Languages, Germany

ARTICLE INFO

Article history:
Received 12 February 2017
Revised 14 February 2018
Accepted 1 April 2018
Available online 20 April 2018

Keywords:
System architecture
Fault trees
Safety
Model co-evolution
Model transformation

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.

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 tolerable with an erroneous design since they are playing fundamental roles in human lives. Therefore quality evaluation 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 (Grunske and Han, 2008).

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 (Mens et al., 2005): 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 can-

* Corresponding author.
E-mail addresses: getir@informatik.hu-berlin.de (S. Getir), grunske@informatik.hu-berlin.de (L. Grunske), van.hoorn@informatik.uni-stuttgart.de (A.v. Hoorn), timo.kehrer@informatik.hu-berlin.de (T. Kehrer), noller@informatik.hu-berlin.de (Y. Noller), matthias.tichy@uni-ulm.de (M. Tichy).

https://doi.org/10.1016/j.jss.2018.04.001
0164-1212/

not 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 Getir et al., 2013). At best, developers may be supported by *recommending* possible synchronization actions, e.g. as in the model-based (co-)evolution framework known as CoWolf (Getir et al., 2015). To achieve consistent co-evolution, CoWolf follows a rule-based approach where incremental 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 Getir et al. (2013) on the evolution of the so-called Pick and Place Unit (PPU) (Legat et al., 2013), a case study from the automation engineering domain which is commonly used in the German priority program "Design for Future – Managed Software Evolution" (Goltz et al., 2015). 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, ISO/IEC, 2001) 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 (Getir et al., 2015) 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.

[1] (Getir et al., 20018).

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 (Grunske and Han, 2008). According to the current standards for the development of safe systems in different application domains (CENELEC EN 50126,128,129, 2000; IEC61508, 1998; ISO26262, 2009), common fault trees (Adler et al., 2007; Bondavalli et al., 1999; Boulanger and Van Quang, 2008; Bretschneider et al., 2004; de Miguel et al., 2008; Giese et al., 2004; Grunske, 2006; Grunske and Kaiser, 2005; Joshi et al., 2007; Papadopoulos and Maruhn, 2001; Papadopoulos et al., 2001; Rae and Lindsay, 2004; Szabo and Ternai, 2009) 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) (Amari et al., 2003; Bechta-Dugan et al., 1992; Dehlinger and Dugan, 2008; Ganesh and Bechta-Dugan, 2002), Generalized Stochastic Petri Nets (GSPNs) (Rugina et al., 2007), State-Event Fault Trees (SEFTs) (Grunske et al., 2005; Kaiser, 2005; Kaiser et al., 2007) or Markov models (Bozzano et al., 2009; 2011). 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 (Cichocki and Górski, 2000; 2001; David et al., 2008; Papadopoulos et al., 2004). To construct the safety artifacts, the system architecture models are often annotated with failure propagation models (Domis and Trapp, 2008; Elmqvist and Nadjm-Tehrani, 2007; Grunske, 2006; Kaiser et al., 2003; Kaiser, 2005; Papadopoulos et al., 2001). These failure propagation models are commonly combinatorial in nature (Papadopoulos et al., 2001; Priesterjahn et al., 2013) thus producing static fault trees.

Beside annotating an architecture specification, there are also approaches to construct a safety artifact via model checking techniques (Grunske et al., 2007; Güdemann and Ortmeier, 2010; Gündemann et al., 2007; Heimdahl et al., 2005; Lipaczewski et al., 2012). 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 (Grunske, 2006) or quasi-automatic (Giese and Tichy, 2006; Giese et al., 2004; Papadopoulos et al., 2001; Priesterjahn et al., 2013) 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 (Ruscio et al., 2011). 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) (Greenyer and Kindler, 2010; Schürr, 1994), PMT (Tratt, 2008), Atlas Transformation Language (ATL) (Jouault and Kurtev, 2006), Groove (Rensink, 2004), Query/View/Transformation (QVT) (OMG, 2011), and the Janus Transformation Language (JTL) (Cicchetti et al., 2010) (see also Schürr and Rensink, 2014 for a recent special issue on current model transformation approaches). Bergmann et al. (2012) 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 Madari et al. (2009), Milovanovic and Milicev (2015) and Wimmer et al. (2012). Madari et al. (2009) 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 (Arendt et al., 2010; Strüber et al., 2017) as one of the technical foundations of our approach. However, in contrast to ours, existing approaches 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 and Milicev (2015) 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 inter-related models is generally known as model repair. Several approaches have been developed which deal with inconsistencies by constructing repair actions (Egyed et al., 2008; Finkelstein et al., 1994; Nentwich et al., 2003). 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 (2012) 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 (Brambilla et al., 2012; Taentzer et al., 2012), model transformations (Ruscio et al., 2011), or syntactic and semantic constraints (Demuth et al., 2013) 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 (Arendt et al., 2010) 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) (Taylor et al., 2009) and fault tree modeling (Vesely et al., 1981), respectively, are used. We use the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009) 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. Fig. 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 Fig. 1), expressed in OCL, completes the specification of the SA meta-model.

3.1.2. Failure model and fault trees

Fig. 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 (2004)) taxonomy. 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, the FT meta-model is completed by a set of OCL constraints (not shown in Fig. 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 Fig. 3). As we can see in Fig. 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.

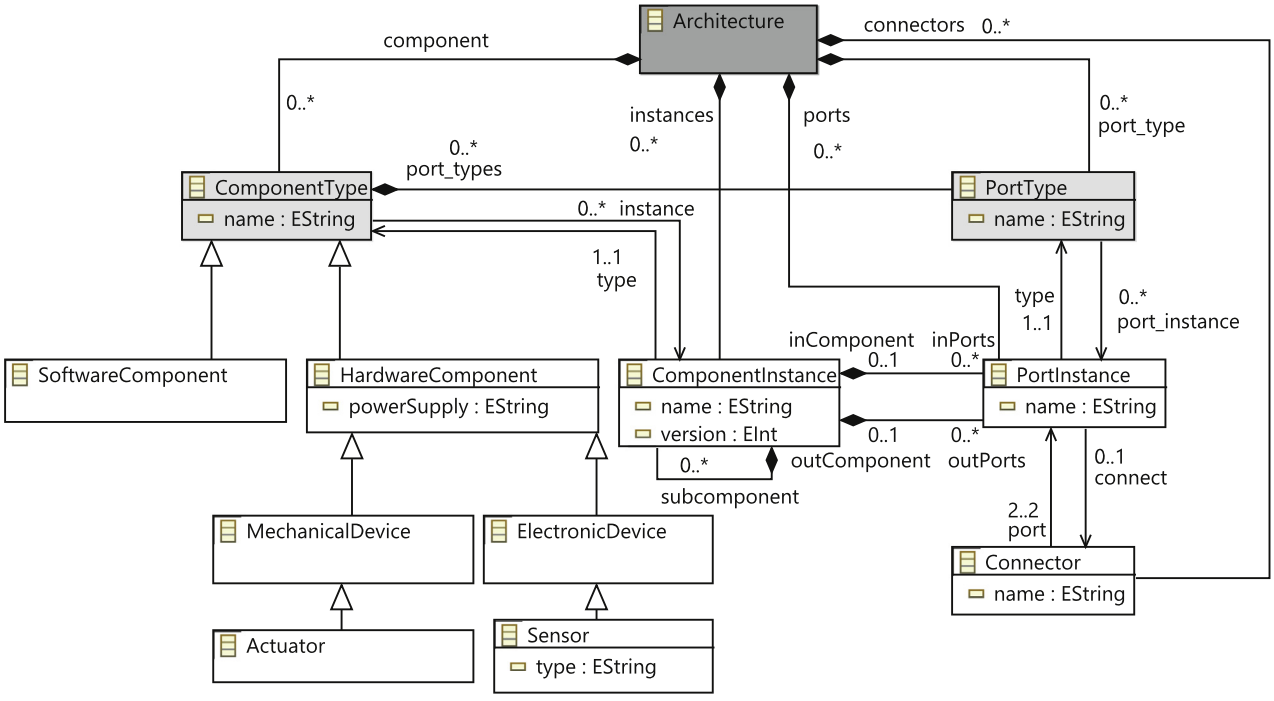


Fig. 1. Classes and relations in the SA meta-model.

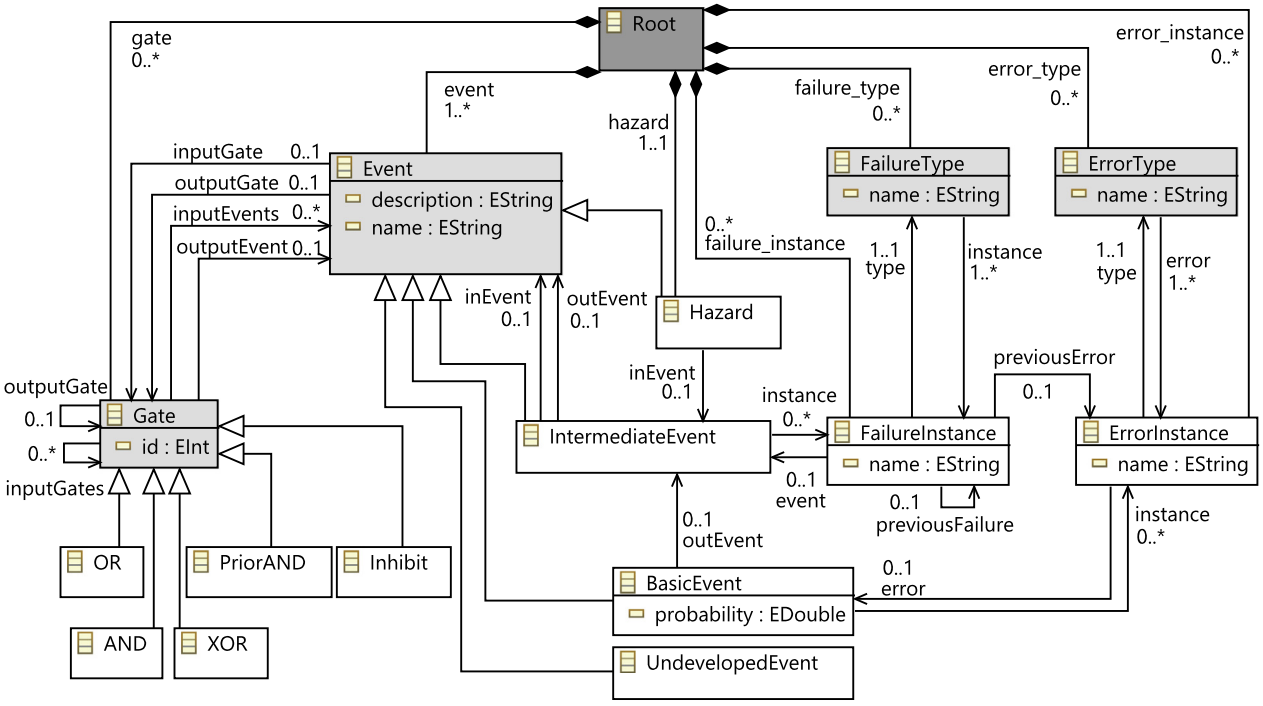


Fig. 2. Classes and relations in the FT meta-model.

3.2. Henshin model transformations

Henshin (Arendt et al., 2010) is a high-level graph rewriting and model transformation language and tool targeting models defined in the Eclipse Modeling Framework (EMF) (Steinberg et al., 2009). Henshin is based on the foundations of algebraic graph transformation (Rozenberg, 1997). 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.

Fig. 4 shows a simple Henshin transformation rule, typed over the previously presented SA meta-model, 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 pa-

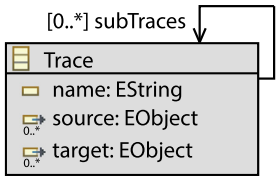


Fig. 3. Trace meta-model (Henshin Trace Metamodel, 2016) used for modeling SA/FT interrelations.

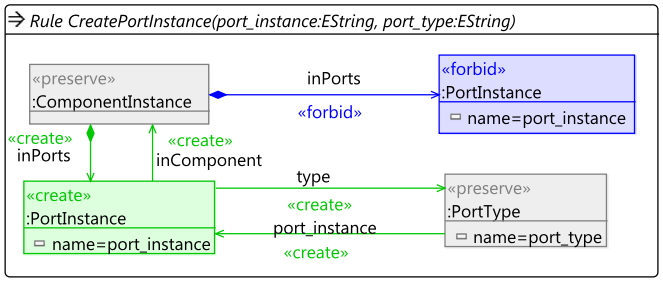


Fig. 4. Henshin rule specifying the creation of a PortInstance.

rameter *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 Fig. 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 and 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.

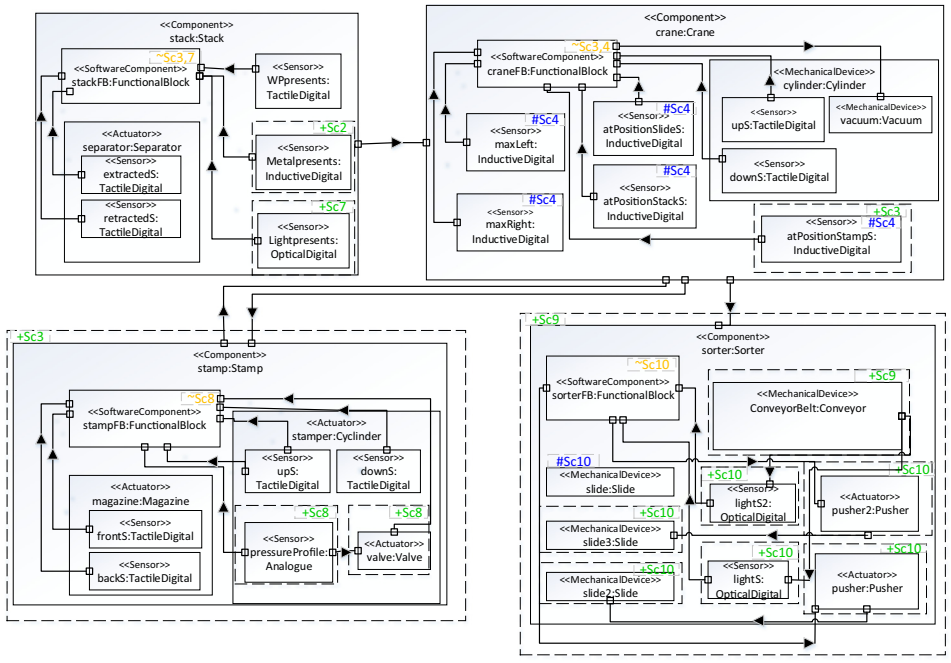
The evolution scenarios for the PPU have been described in Legat et al. (2013) and we selected a subset of 12 scenarios for our study Getir et al. (2013) out of these 14 evolution scenarios—those that were identified as including system changes affecting the system's safety properties. For each scenario, 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 (Kehrer et al., 2012b; Kolovos et al., 2009). 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

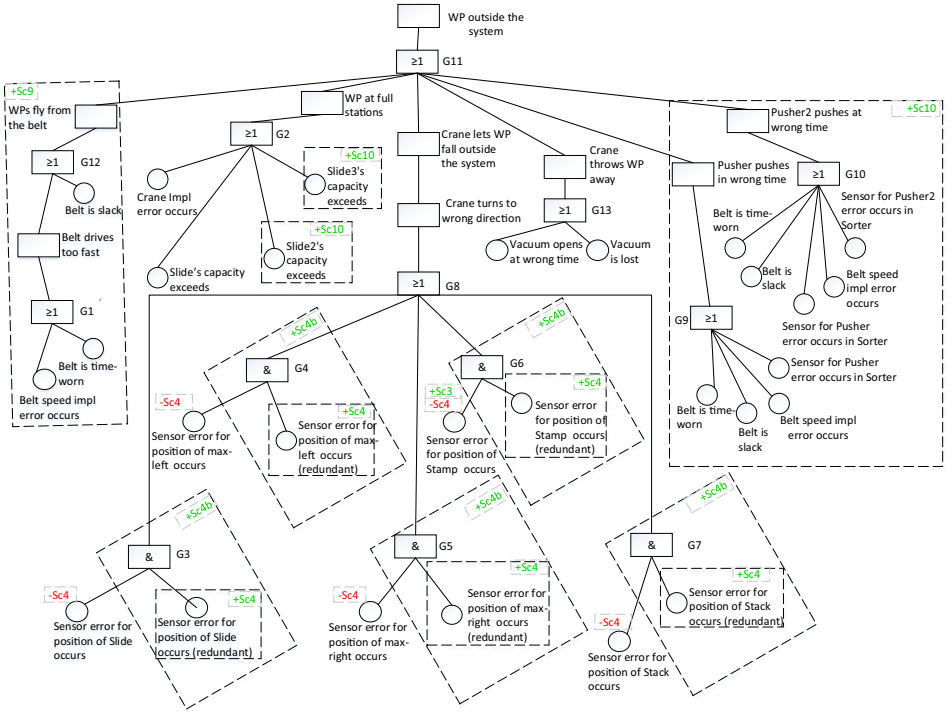
Section 4.1.1 includes a compact description of the PPU's initial scenario (SC0), including the corresponding SA and FT model. A brief summary of the changes in the subsequent evolution scenarios is provided by Section 4.1.2. Note that in the latter, changes to the SA and FT models—as detailed in Section Appendix A—are not covered for the sake of brevity. Fig. 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 (Figs. A.13–A.15). Note that a similar description of the PPU scenarios is also provided in Legat et al. (2013) (without considering SA and FT models) and Getir et al. (2013). However, we decided to include it to make this paper self-contained. Table 1 provides a quantitative summary 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

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

	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



(a) SA component model of the PPU



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

Fig. 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.)

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 a 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 (Fig. 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. Fig. 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 Fig. 5(a) called *atPositionStackS* is related to the basic event called "Sensor error for positions of Stack occurs" in fault tree model given in Fig. 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).

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 Fig. 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 Fig. 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 Fig. 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 Fig. 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.

				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

Fig. 6. Analysis results. (+) addition; (−) deletion.

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. Fig. 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 Fig. 6(a) and (b) and *trgt* refers to a column. We use n_{src} for the number of all changes for a change type *src*. The cells in Fig. 6(b) contain then the fraction

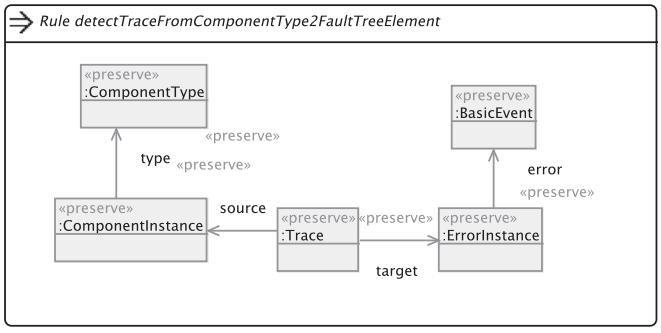


Fig. 7. Mining pattern for trace elements between ComponentType and BasicEvent.

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 (Getir et al., 2013) 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 Herrmannsdörfer et al. (2010). 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 (Getir et al., 2015) 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 *(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 Fig. 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 Fig. 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.

Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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 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

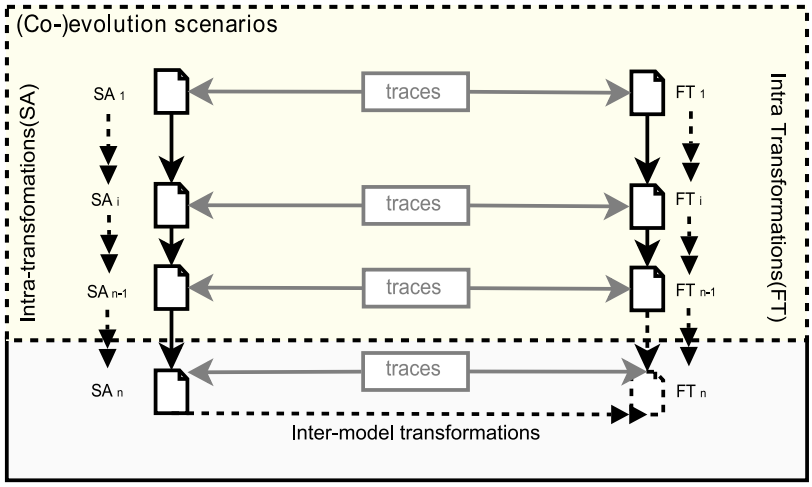
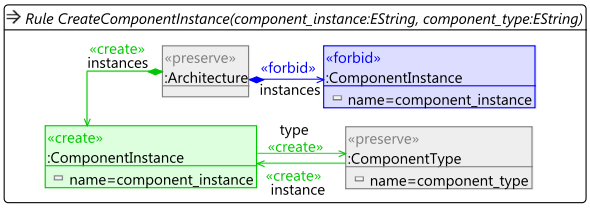
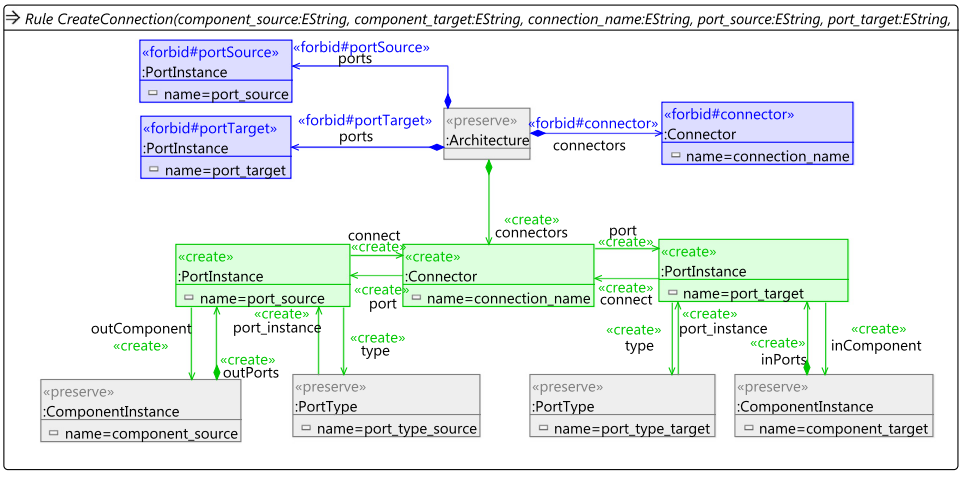


Fig. 8. The role of transformations supporting model (co-)evolution.



(a) Component instance creation



(b) Connection creation

Fig. 9. Intra-model transformations for system architecture.

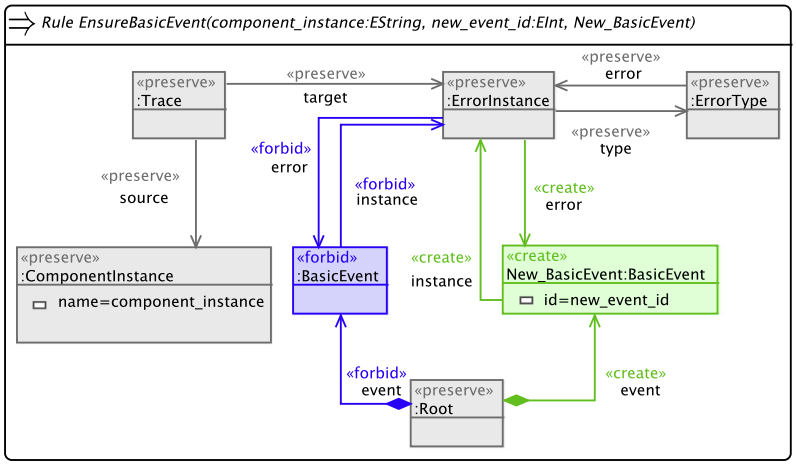
refer to as *consistency*, *coupling/connecting*, *decoupling* and *propagation*, respectively.

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 Fig. 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.

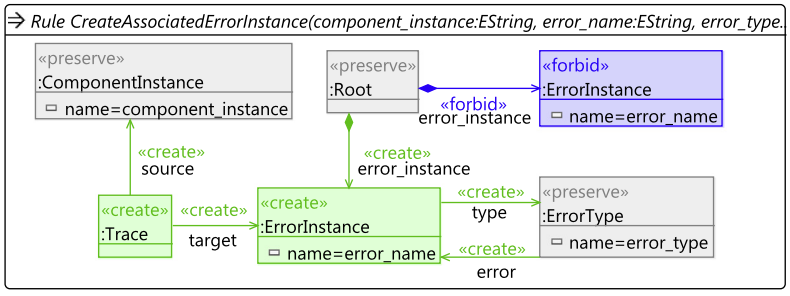
Secondly, *coupling/connecting* transformation rules aim at creating new trace objects to relate the corresponding elements between M_{SA} and M_FT. In other words, they specifically add new

Table 2
Summary of inter-model transformations.

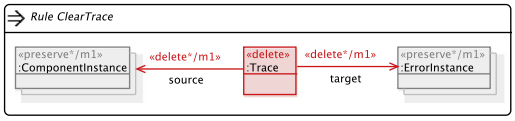
Category		#	Name	+Node	+Edge	−Node	−Edge
Consistency	Addition	1	EnsureBasicEvent	1	2	0	0
	Addition	2	EnsureIntermediateEvent	1	2	0	0
	Deletion	3	RemoveConnectedBasicEvent	0	0	1	3
	Deletion	4	RemoveConnectedIntermediateEvent	0	0	1	3
Coupling/connecting		5	CreateAssociatedErrorInstance	2	5	0	0
		6	CreateAssociatedFailureInstance	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



(a) Creation of a BasicEvent for the corresponding ComponentInstance



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



(c) Removing the Trace element

Fig. 10. Inter-model transformations between system architecture and fault trees.

inter-model relationships to the models. An example is demonstrated with the rule *CreateAssociatedErrorInstance* in Fig. 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 Fig. 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 Fig. 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.

5.3. Running example showing evolution actions

In Fig. 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 *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 Fig. 11(b), we illustrate how the changes of Fig. 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 Fig. 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 Fig. 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 Fig. 11(b).

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_ip, FT_ip) 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 (ISO/IEC, 2001).

6.1. 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., Wenzel, 2008; Yazdi et al., 2013; Yazdi et al., 2014) in order to reason about quality aspects *(i)* and *(ii)* of a transformation rule set.

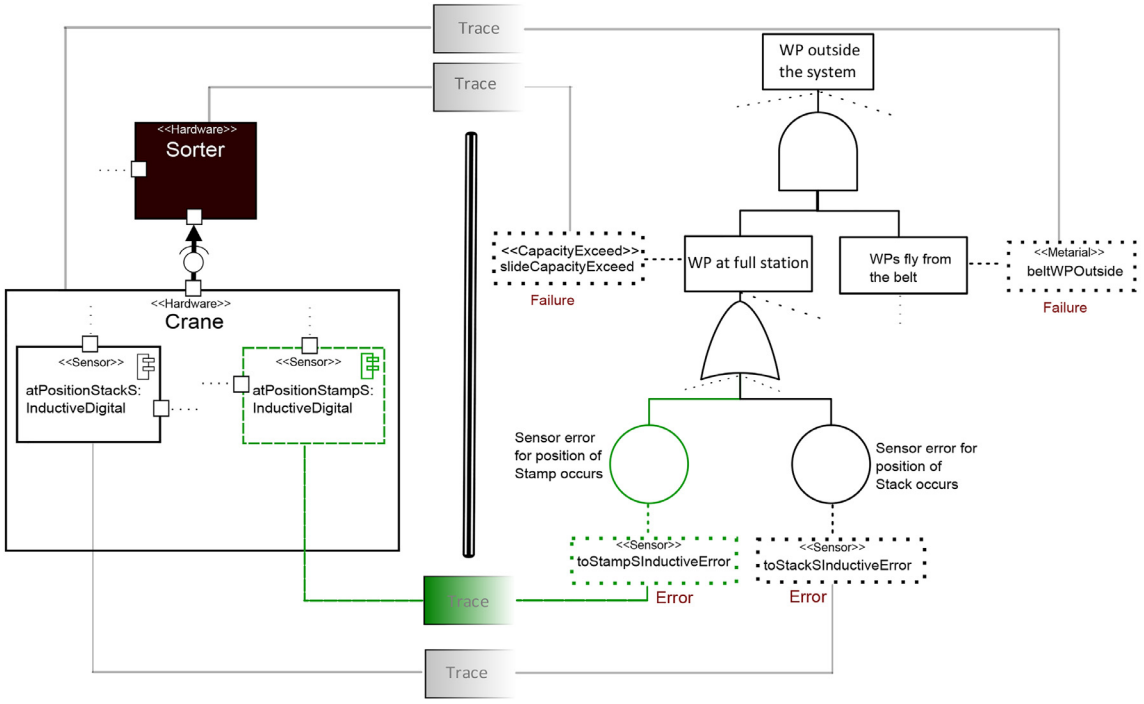
In our study, we use the SiLift model differencing framework (Kehrer et al., 2012a), which has been specifically designed for the comparison of graph-structured models. In addition to the two input models M_1 and M_2, which are to be compared with each other, the differencing engine takes a set R of transformation rules as additional configuration parameter as input (cf. Fig. 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 (Kehrer et al., 2013). 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.

Our study design is described in Section 6.2, quality metrics are introduced in Section 6.3. Results are summarized by Section 6.4, and Section 6.5 finally discusses threats to validity.

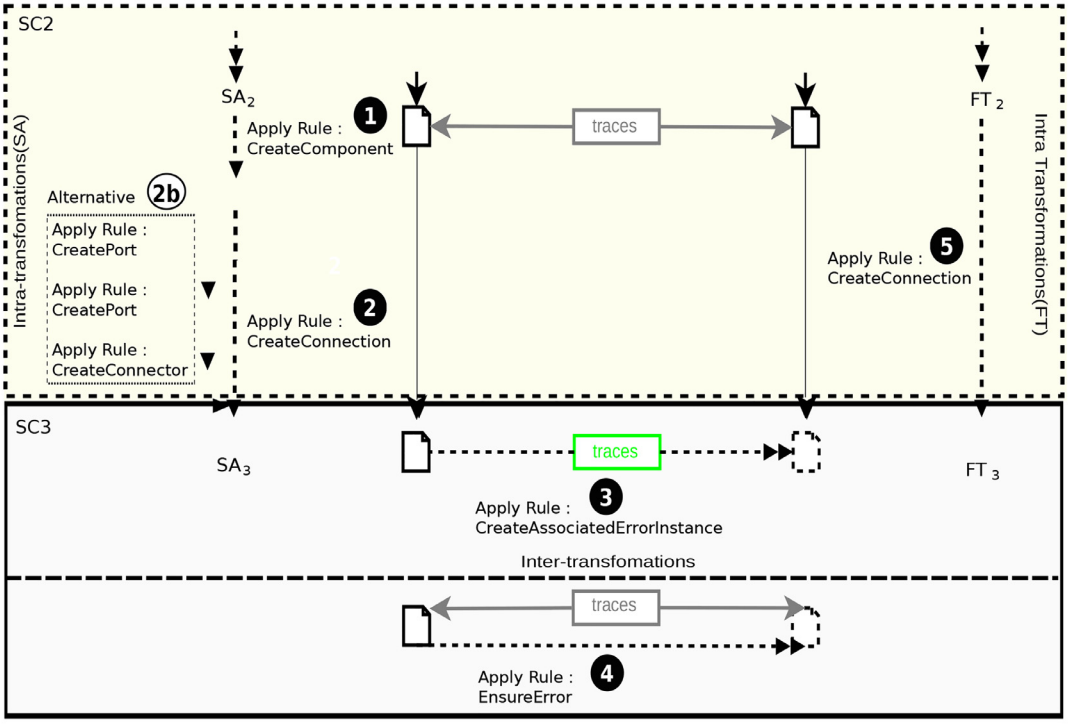
6.2. Study design

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



(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)

Fig. 11. Example of an evolution step.

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. Figs. 1 and 2) using the approach presented in Kehrer et al. (2016), which is implemented in the SiDiff Edit Rule Generator (SERGe) (Rindt et al., 2014). As argued in Kehrer et al. (2016), transformation rules generated with SERGe are similar to the edit operations provided by typical editors for visual modeling languages.

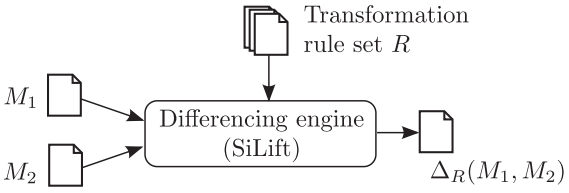


Fig. 12. SiLift model differencing engine: in- and output parameters.

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.

6.3. Measures

6.3.1. 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.

6.3.2. 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

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

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 $f_{red}(Inter/Intra, Generated, SCi, SCi+1)$.

6.4. 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 $SCi \rightarrow SCi+1$ 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 *EnsureBasicEvent* 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.

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

	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 4
Distribution of inter-model transformations over evolution steps $SCi \rightarrow SCi{+}1$.

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	

6.5. Threats to validity

Our conclusions are subject to several threats to validity (Wohlin et al., 2000), the major ones will be discussed in the remainder of this section.

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 evaluating recommender systems in both online and offline experiments (Shani and Gunawardana, 2011). 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 (Kehrer et al., 2011; Langer et al., 2013).

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. (2013) 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 (Getir et al., 2013).

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 and 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 (Ayav and Sözer, 2016).

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.

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. Figs. 5 and 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).

Fig. 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 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.

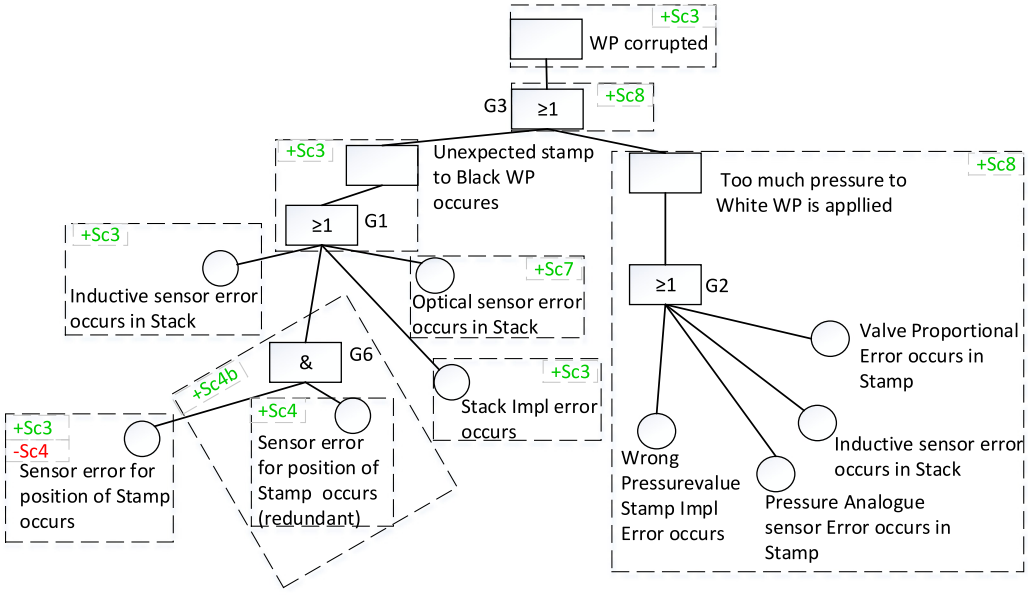


Fig. A.13. FT for the hazard that a WP is corrupted (FT2).

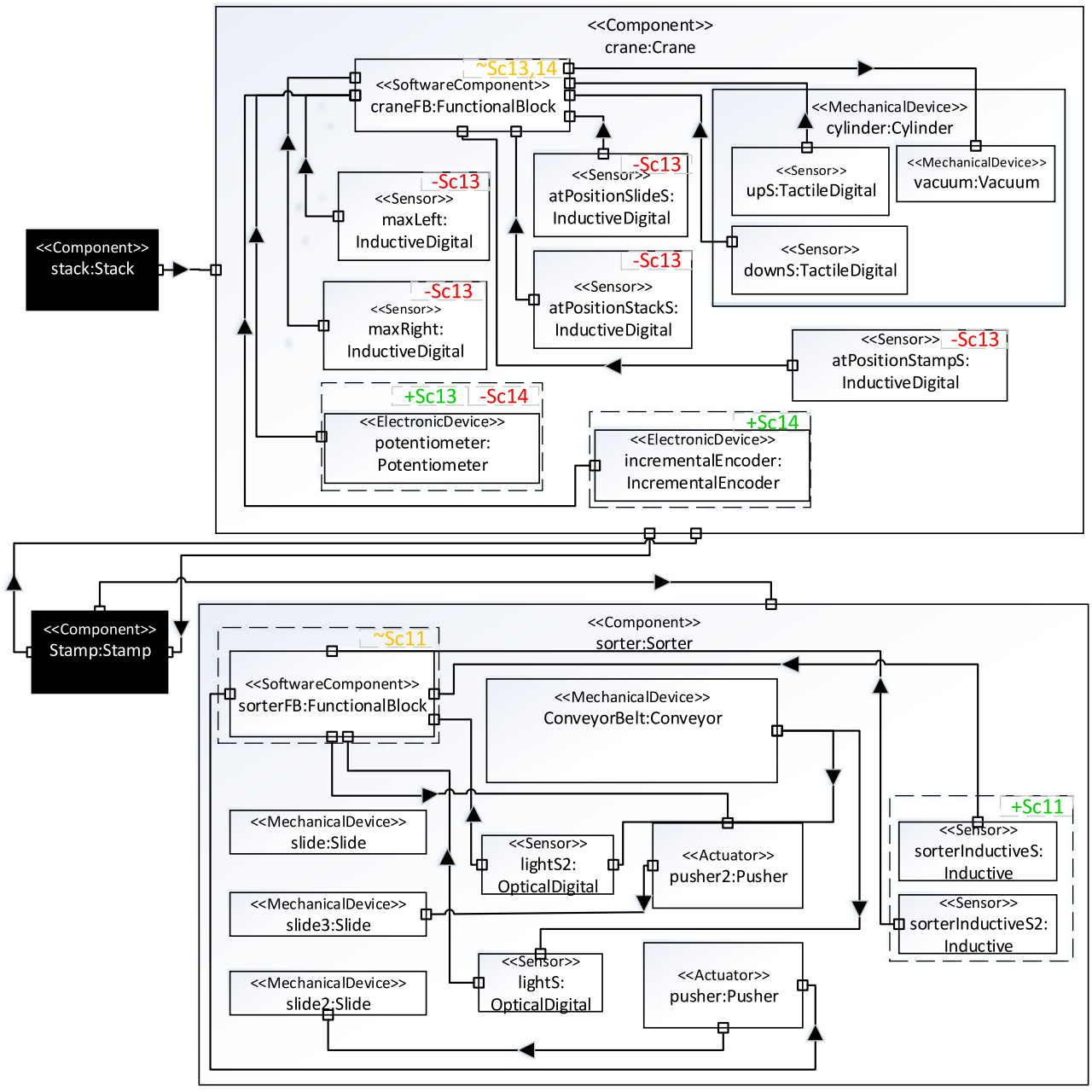


Fig. A.14. Partial SA component model for the PPU scenarios 11,13,14.

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. Fig. 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 corre-

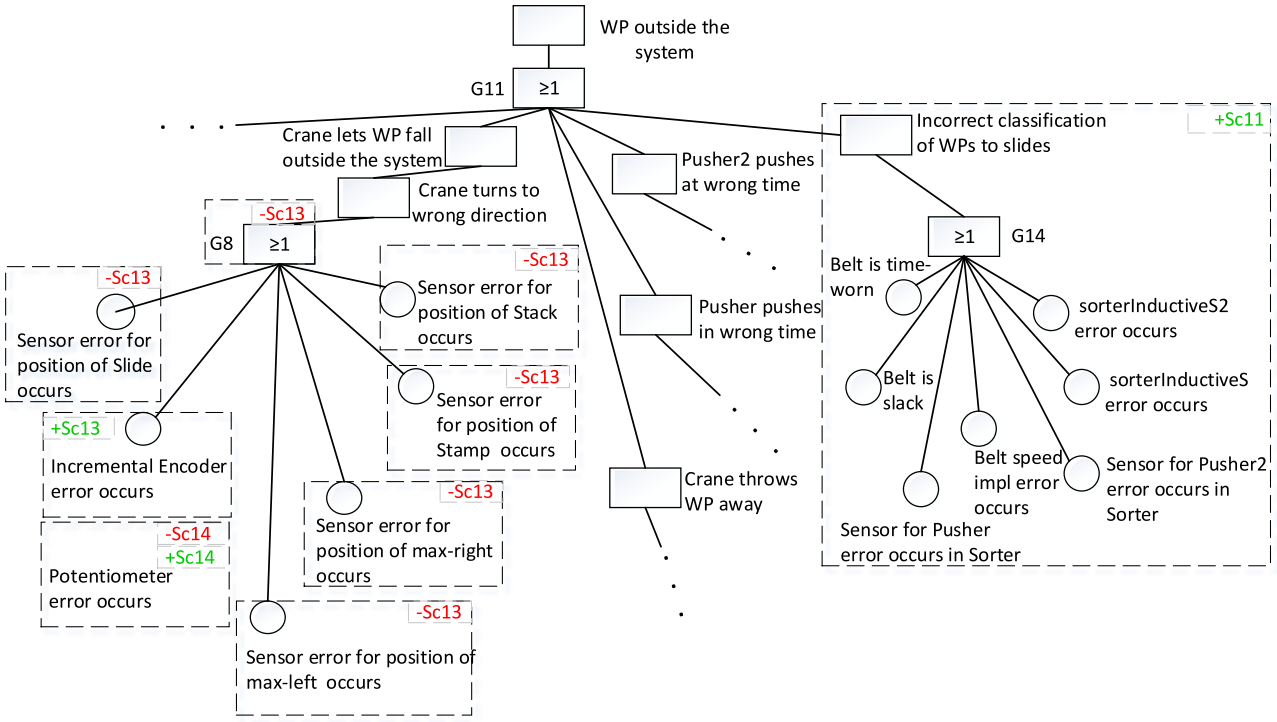


Fig. A.15. Partial FT model (FT1) with changes only for the scenarios 11,13,14.

sponding 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 Fig. 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 (*G*3–7 in Fig. 5(b)) are added, each having two basic events as input and leading to the already-existing OR gate (*G*8). 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 (*G*1, *G*12) 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 *G*9, *G*10) 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 Figs. 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 *G*8 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 is replaced by the new basic event and error instance corresponding to the potentiometer in FT1.

References

Adler, R., Förster, M., Trapp, M., 2007. Determining configuration probabilities of safety-critical adaptive systems. In: Proceedings of the Twenty-First International Conference on Advanced Information Networking and Applications (AINA 2007). IEEE Computer Society, pp. 548–555.

Amari, S., Dill, G., Howald, E., 2003. A new approach to solve dynamic fault trees. Proceedings of the 2003 Annual Reliability and Maintainability Symposium, pp. 374–379.

Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G., 2010. Henshin: advanced concepts and tools for in-place EMF model transformations. In: Proceedings of the Thirteenth International Conference on Model Driven Engineering on Languages and Systems, MODELS 2010, Part I, pp. 121–135.

Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.E., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. 1 (1), 11–33.

Ayav, T., Sözer, H., 2016. Identifying critical architectural components with spectral analysis of fault trees. Appl. Soft Comput. 49, 1270–1282.

Bechta-Dugan, J., Bavuso, S., Boyd, M., 1992. Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans. Reliab. 41 (3), 363–377. September.

Bergmann, G., Ráth, I., Varró, G., Varró, D., 2012. Change-driven model transformations. Softw. Syst. Model. 11 (3), 431–461.

Bondavalli, A., Majzik, I., Mura, I., 1999. Automated dependability analysis of UML designs. In: Proceedings of the 1999 IEEE International Symposium on Object-Oriented Real-time distributed Computing, 2.

Boulanger, J. L., Van Quang, D., 2008. Experiences from a Model-Based Methodology for Embedded Electronic Software in Automobile. April. 1–6, https://ieeexplore.ieee.org/document/4530259/.

Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M., 2009. The compass approach: correctness, modelling and performability of aerospace systems. In: Buth, B., Rabe, G., Seyfarth, T. (Eds.), Proceedings of the Eighth International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2009. In: LNCS, 5775. Springer, pp. 173–186.

Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M., 2011. Safety, dependability and performance analysis of extended AADL models. Comput. J. 54 (5), 754–775.

Brambilla, M., Cabot, J., Wimmer, M., 2012. Model-driven software engineering in practice. Synthesis Lectures on Software Engineering. Morgan & Claypool.

Bretschneider, M., Holberg, H.J., Bode, E., Bruckner, I., 2004. Model-based safety analysis of a flap control system. In: Proceedings of the Fourteenth Annual INCOSE Symposium.

CENELEC EN 50126,128,129, 2000. 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, http://www.era.europa.eu/core-activities/ertms/pages/mandatory-en-std.aspx.

Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A., 2010. JTL: a bidirectional and change propagating transformation language. In: Proceedings of the Third International Conference on Software Language Engineering, SLE 2010. In: Lecture Notes in Computer Science, 6563. Springer, pp. 183–202.

Cichocki, T., Górski, J., 2000. Failure mode and effect analysis for safety-critical systems with software components. In: Koornneef, F., van der Meulen, M. (Eds.), Proceedings of the Nineteenth International Conference on Computer Safety, Reliability and Security, SAFECOMP 2000. In: Lecture Notes in Computer Science, 1943. Springer, pp. 382–394.

Cichocki, T., Górski, J., 2001. Formal support for fault modelling and analysis. In: Voges, U. (Ed.), Proceedings of the Twentieth International Conference on Computer Safety, Reliability and Security, SAFECOMP 2001. In: Lecture Notes in Computer Science, 2187. Springer, pp. 190–199.

David, P., Idasiak, V., Kratz, F., 2008. 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. ESREL 2008 and 17th SRA-EUROPE annual conference. Taylor & Francis Group, pp. 2259–2266. Jan.

Dehlinger, J., Dugan, J.B., 2008. Analyzing dynamic fault trees derived from model-based system architectures. Nucl. Eng. Technol. Int. J. Korean Nucl. Soc. 40 (5), 365–374.

Demuth, A., Lopez-Herrejon, R.E., Egyed, A., 2013. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: Proceedings of the 2013 International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 287–303.

Domis, D., Trapp, M., 2008. Integrating safety analyses and component-based design. In: Proceedings of the Twentieth International Conference on Computer Safety, Reliability and Security, SAFECOMP 2008, pp. 58–71.

Egyed, A., Letier, E., Finkelstein, A., 2008. Generating and evaluating choices for fixing inconsistencies in UML design models. In: Proceedings of the Twenty-Third IEEE/ACM International Conference on Automated Software Engineering (ASE 2008). IEEE, pp. 99–108.

Elmqvist, J., Nadjm-Tehrani, S., 2007. Safety-oriented design of component assemblies using safety interfaces. In: Proceedings of the 2007 Formal Aspects of Component Software, pp. 57–72.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., 1994. Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. 20 (8), 569–578.

Ganesh, P., Bechta-Dugan, J., 2002. Automatic synthesis of dynamic fault trees from UML systemmodels. In: Proceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE).

S. Getir, L. Grunske, A. van Hoorn, T. Kehrer,Y. Noller, and M. Tichy, (2018), Supporting Semi-Automatic Co-Evolution of Architecture and Fault Tree Models, Mendeley Data, v1, https://data.mendeley.com/datasets/6kkh54xbpj/1.

Getir, S., Grunske, L., Bernasko, C.K., Käfer, V., Sanwald, T., Tichy, M., 2015. Cowolf – a generic framework for multi-view co-evolution and evaluation of models. In: Proceedings of the Eighth International Conference on Theory and Practice of Model Transformations, ICMT 2015, Held as Part of STAF 2015, pp. 34–40.

Getir, S., van Hoorn, A., Grunske, L., Tichy, M., 2013. Co-evolution of software architecture and fault tree models: an explorative case study on a pick and place factory automation system. In: Proceedings of the Fifth International Workshop Non-functional Properties in Modeling: Analysis, Languages and Processes Co-located with Sixteenth International Conference on Model Driven Engineering Languages and Systems (MODELS 2013 (NIM-ALP 2013), 1074, pp. 32–40. CEUR Workshop Proceedings.

Giese, H., Tichy, M., 2006. Component-based hazard analysis: optimal designs, product lines, and online-reconfiguration. In: Proceedings of the SAFECOMP 200. In: LNCS, 4166. Springer, pp. 156–169.

Giese, H., Tichy, M., Schilling, D., 2004. Compositional hazard analysis of UML component and deployment models. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (Eds.), Proceedings of the Twenty-Third International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2004. In: LNCS, 3219. Springer, pp. 166–179.

Goltz, U., Reussner, R.H., Goedicke, M., Hasselbring, W., Märtin, L., Vogel-Heuser, B., 2015. Design for future: managed software evolution. Comput. Sci.-Res. Dev. 30 (3–4), 321–331.

Greenyer, J., Kindler, E., 2010. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. Softw. Syst. Model. 9 (1), 21–46.

Grunske, L., 2006. Towards an integration of standard component-based safety evaluation techniques with save CCM. In: Hofmeister, C., Crnkovic, I., Reussner, R.

(Eds.), Proceedings of the Second International Conference on Quality of Software Architectures, QoSA 2006. In: LNCS, 4214. Springer, pp. 199–213.
Grunske, L., Colvin, R., Winter, K., 2007. Probabilistic model-checking support for FMEA. In: Proceedings of the Fourth International Conference on the Quantitative Evaluaiton of Systems (QEST 2007). IEEE Computer Society, pp. 119–128.
Grunske, L., Han, J., 2008. A comparative study into architecture-based safety evaluation methodologies using AADL'S Error annex and failure propagation models. In: Proceedings of the Eleventh IEEE High Assurance Systems Engineering Symposium, HASE 2008. IEEE Computer Society, pp. 283–292.
Grunske, L., Kaiser, B., 2005. Automatic generation of analyzable failure propagation models from component-level failure annotations. In: Proceedings of the Fifth International Conference on Quality Software (QSIC 2005). IEEE, pp. 117–123.
Grunske, L., Kaiser, B., Papadopoulos, Y., 2005. Model-driven safety evaluation with state-event-based component failure annotations. In: Proceedings of the Eighth International Symposium on Component-Based Software Engineering, CBSE 2005, pp. 33–48.
Güdemann, M., Ortmeier, F., 2010. A framework for qualitative and quantitative formal model-based safety analysis. In: Proceedings of the Twelfth IEEE High Assurance Systems Engineering Symposium, HASE 2010. IEEE Computer Society, San Jose, CA, USA, pp. 132–141. November 3–4, 2010.
Gündemann, M., Ortmeier, F., Reif, W., 2007. Using deductive cause-sequence analysis (DCCA) with SCADE. In: Proceedings of the SAFECOMP 2007. In: LNCS, 4680, pp. 465–478.
Heimdahl, M.P.E., Choi, Y., Whalen, M.W., 2005. Deviation analysis: a new use of model checking. Autom. Softw. Eng. 12 (3), 321–347.
Henshin Trace Metamodel, 2016. http://wiki.eclipse.org/Henshin_Trace_Model, last access: Dec.
Herrmannsdörfer, M., Ratiu, D., Wachsmuth, G., 2010. Language evolution in practice: the history of GMF. In: Software Language Engineering. Springer, pp. 3–22.
IEC61508, 1998. International Standard IEC 61508. International Electrotechnical Commission (IEC).
ISO26262, 2009. ISO 26262 Road Vehicles and Functional Safety, https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en.
ISO/IEC, 2001. Software Engineering – Product Quality – Part 4, Quality in Use, https://www.iso.org/standard/35733.html.
Joshi, A., Vestal, S., Binns, P., 2007. Automatic generation of static fault trees from AADL models. In: Proceedings of the DSN Workshop on Architecting Dependable Systems. In: Lecture Notes in Computer Science. Springer.
Jouault, F., Kurtev, I., 2006. Transforming models with ATL. In: Bruel, J.M. (Ed.), Proceedings of the Satellite Events at the MoDELS 2005 Conference. In: LNCS, 3844. Springer, pp. 128–138. Revised Selected Papers.
Kaiser, B., 2005. State/Event Fault Trees: A Safety and Reliability Analysis Technique for Software-Controlled Systems. Technische Universität Kaiserslautern, Fachbereich Informatik Ph.D. thesis.
Kaiser, B., Gramlich, C., Förster, M., 2007. State/event fault trees–a safety analysis model for software-controlled systems. Reliab. Eng. Syst. Saf. 92 (11), 1521–1537. SAFECOMP 2004, the 23rd International Conference on Computer Safety, Reliability and Security.
Kaiser, B., Liggesmeyer, P., Mäckel, O., 2003. A new component concept for fault trees. In: Proceedings of the Eighth Australian Workshop on Safety Critical Systems and Software, SCS '03. Australian Computer Society, Inc, Darlinghurst, Australia, pp. 37–46.
Kehrer, T., Kelter, U., Ohrndorf, M., Sollbach, T., 2012. Understanding model evolution through semantically lifting model differences with SiLift. In: Proceedings of the Twenty-Eighth IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 638–641.
Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M., 2012. Adaptability of model comparison tools. In: Proceedings of the Twenty-Seventh IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 306–309.
Kehrer, T., Kelter, U., Taentzer, G., 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the Twenty-Sixth IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 163–172.
Kehrer, T., Kelter, U., Taentzer, G., 2013. Consistency-preserving edit scripts in model versioning. In: Proceedings of the Twenty-Eighth IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 191–201.
Kehrer, T., Taentzer, G., Rindt, M., Kelter, U., 2016. Automatically deriving the specification of model editing operations from meta-models. In: Proceedings of the Ninth International Conference on Theory and Practice of Model Transformations, ICMT 2016. In: Lecture Notes in Computer Science, 9765. Springer, pp. 173–188.
Kolovos, D.S., Ruscio, D.D., Pierantonio, A., Paige, R.F., 2009. Different models for model matching: an analysis of approaches to support model differencing. In: Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models, CVSM'09. IEEE, pp. 1–6.
Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., Kappel, G., 2013. A posteriori operation detection in evolving software models. J. Syst. Softw. 86 (2), 551–566.
Legat, C., Folmer, J., Vogel-Heuser, B., 2013. Evolution in industrial plant automation: a case study. In: Proceedings of the IECON 2013. IEEE, pp. 4386–4391.
Lipaczewski, M., Struck, S., Ortmeier, F., 2012. Using tool-supported model based safety analysis – progress and experiences in SAML development. In: Proceedings of the Fourteenth International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012. IEEE Computer Society, pp. 159–166.
Madari, I., Angyal, L., Lengyel, L., 2009. Traceability-based incremental model synchronization. WSEAS Trans. Comput. 8 (10), 1691–1700.
Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., 2005. Challenges in software evolution. In: Proceedings of the IWPSE 2005. IEEE Computer Society, pp. 13–22.
de Miguel, M.A., Briones, J.F., Silva, J.P., Alonso, A., 2008. Integration of safety analysis in model-driven software development. Softw. IET 2 (3), 260–280. June.
Milovanovic, V., Milicev, D., 2015. An interactive tool for UML class model evolution in database applications. Softw. Syst. Model. 14 (3), 1273–1295.
Nentwich, C., Emmerich, W., Finkelstein, A., 2003. Consistency management with repair actions. In: Proceedings of the Twenty-Fifth International Conference on Software Engineering 2003. IEEE Computer Society, pp. 455–464.
Object Management Group (OMG), 2011. MOF 2.0 QVT Final Adopted Specification v1.1. OMG Adopted Specification formal/2011-01-01.
Papadopoulos, Y., Maruhn, M., 2001. Model-based automated synthesis of fault trees from Matlab.Simulink models. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks.
Papadopoulos, Y., McDermid, J., Sasse, R., Heiner, G., 2001. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Int. J. Reliab. Eng. Syst. Saf. 71 (3), 229–247.
Papadopoulos, Y., Parker, D., Grante, C., 2004. Automating the failure modes and effects analysis of safety critical systems. In: Proceedings of the International Symposium on High-Assurance Systems Engineering (HASE 2004). IEEE Computer Society, pp. 310–311.
Priesterjahn, C., Steenken, D., Tichy, M., 2013. Timed hazard analysis of self-healing systems. In: Proceedings of the 2013 ASAS. In: LNCS, 7740. Springer, pp. 112–151.
Rae, A., Lindsay, P., 2004. A behaviour-based method for fault tree generation. In: Proceedings of the Twenty-Second International System Safety Conference, pp. 289–298.
Rensink, A., 2004. The GROOVE simulator: a tool for state space generation. In: Proceedings of the 2004 Applications of Graph Transformations with Industrial Relevance (AGTIVE). In: Lecture Notes in Computer Science, 3062. Springer, pp. 479–485.
Rindt, M., Kehrer, T., Kelter, U., 2014. Automatic generation of consistency-preserving edit operations for MDE tools. In: Proceedings of the 2014 CEUR Workshop on Demonstrations Track of MoDELS 2014, 1255.
Rozenberg, G., 1997. Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations. World Scientific.
Rugina, A.E., Kanoun, K., Kaâniche, M., 2007. A system dependability modeling framework using AADL and GSPNs. In: Proceedings of the 2007 Architecting Dependable Systems IV. In: LNCS, 4615. Springer, pp. 14–38.
Ruhroth, T., Wehrheim, H., 2012. Model evolution and refinement. Sci. Comput. Program. 77 (3), 270–289.
Ruscio, D.D., Iovino, L., Pierantonio, A., 2011. What is needed for managing co-evolution in MDE? In: Proceedings of the IWMCP 2011. ACM, pp. 30–38.
Schürr, A., 1994. Specification of graph translators with triple graph grammars. In: Proceedings of the Twentieth International Workshop on Graph-Theoretic Concepts in Computer Science. In: Lecture Notes in Computer Science (LNCS), 903. Springer Verlag, pp. 151–163.
Schürr, A., Rensink, A., 2014. Software and systems modeling with graph transformations theme issue of the journal on software and systems modeling. Softw. Syst. Model. 13 (1), 171–172.
Shani, G., Gunawardana, A., 2011. Evaluating recommendation systems. In: Recommender Systems Handbook. Springer, pp. 257–297.
Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2009. EMF: Eclipse Modeling Framework, second ed. Addison-Wesley.
Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M., 2017. Henshin: a usability-focused framework for EMF model transformation development. In: Proceedings of the 2017 International Conference on Graph Transformation. Springer, pp. 196–208.
Szabo, G., Ternai, G., 2009. Automatic fault tree generation as a support for safety studies of railway interlocking systems. In: Proceedings of the IFAC Symposium on Control in Transportation Systems.
Taentzer, G., Mantz, F., Lamo, Y., 2012. Co-transformation of graphs and type graphs with application to model co-evolution. In: Proceedings of the ICGT 2012, pp. 326–340.
Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2009. Software Architecture: Foundations, Theory and Practice. John Wiley & Sons, Inc.
Tratt, L., 2008. A change propagating model transformation language. J. Object Technol. 7 (3), 107–124.
Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F., 1981. Fault tree handbook. Technical report, NUREG–0492. U.S. Nuclear Regulatory Commission.
Wenzel, S., 2008. Scalable visualization of model differences. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models. ACM, pp. 41–46.
Wimmer, M., Moreno, N., Vallecillo, A., 2012. Viewpoint co-evolution through coarse-grained changes and coupled transformations. In: Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Springer, pp. 336–352.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, USA.
Yazdi, H.S., Pietsch, P., Kehrer, T., Kelter, U., 2013. Statistical analysis of changes for synthesizing realistic test models. In: Software Engineering. Gesellschaft für Informatik, pp. 225–238.
Yazdi, H.S., Pietsch, P., Kehrer, T., Kelter, U., 2014. Synthesizing realistic test models. In: Computer Science-Research and Development. Springer, pp. 1–23.

Sinem Getir is a doctoral student at Humboldt University Berlin, Germany. She has research interests lie in the field of software engineering upon the reliable software systems, probabilistic and incremental verification, model driven development and of quality assurance of evolving software systems.

Lars Grunske is currently Professor at the Department of Computer Science at Humboldt University Berlin, Germany. He has active research interests in the areas modelling and verification of systems and software. His main focus is on automated analysis, mainly probabilistic and timed model checking and model-based dependability evaluation of complex software intensive systems.

André van Hoorn is a member of the Reliable Software Systems (RSS) group at the University of Stuttgart, Germany. His research interests are in the area of architecture-based software performance engineering and software reengineering. Particularly, he is interested in application performance monitoring, modelling, and management, as well as dynamic/hybrid analysis of legacy systems for architecture recovery and evolution.

Timo Kehrer is heading the Model-Driven Software Engineering Group at the Department of Computer Science at Humboldt-University of Berlin, Germany. He has active research interests in various areas of model-based software and system development with a particular focus on model evolution.

Yannic Noller is a doctoral student at Humboldt University Berlin, Germany. His research interests includes automated software verification, probabilistic software analysis and software architecture. He has a M.Sc. degree from the University of Stuttgart.

Matthias Tichy is currently Professor at the Institute of Software Engineering and Programming Languages at University of Ulm, Germany. His main research focus is on domain specific languages, particularly for mechatronic systems, graph transformations as underlying formal basis in many of these research activities. Furthermore, he employs empirical research methods to understand how humans use software as well as to evaluate technical contributions.