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

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

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

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

1 Introduction

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

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

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

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

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

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

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

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

2 Modeling Languages and PPU Case Study System

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

2.1 Modeling Languages

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

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

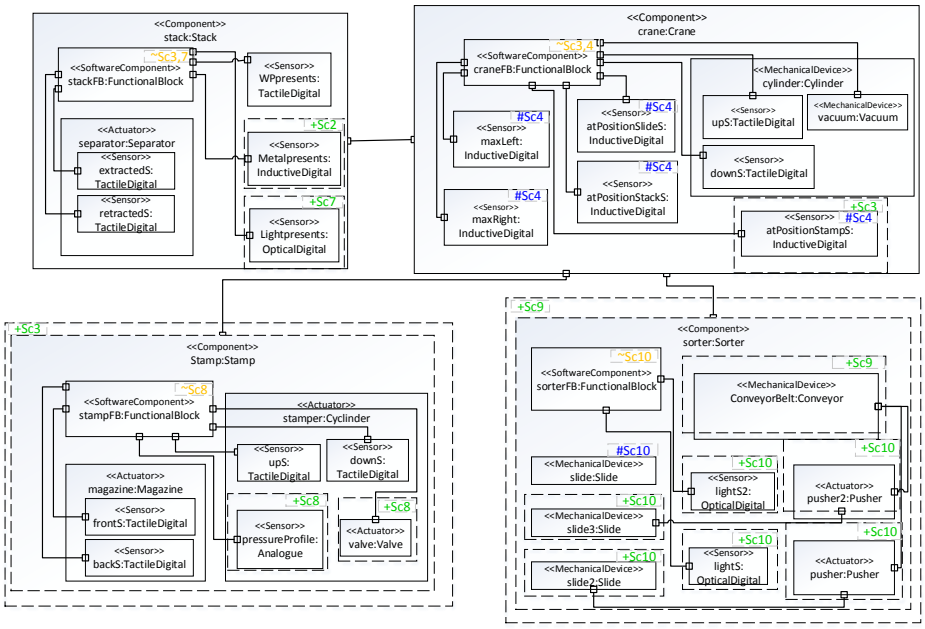
Our second modeling language FT allows the definition of a failure model and a set of corresponding fault trees. A failure model includes the definition of error types and failure types and their instances based on [1]. To exemplify the difference between instance- and type-level, a sensor error is an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

2.2 Case Study: Evolution Scenarios

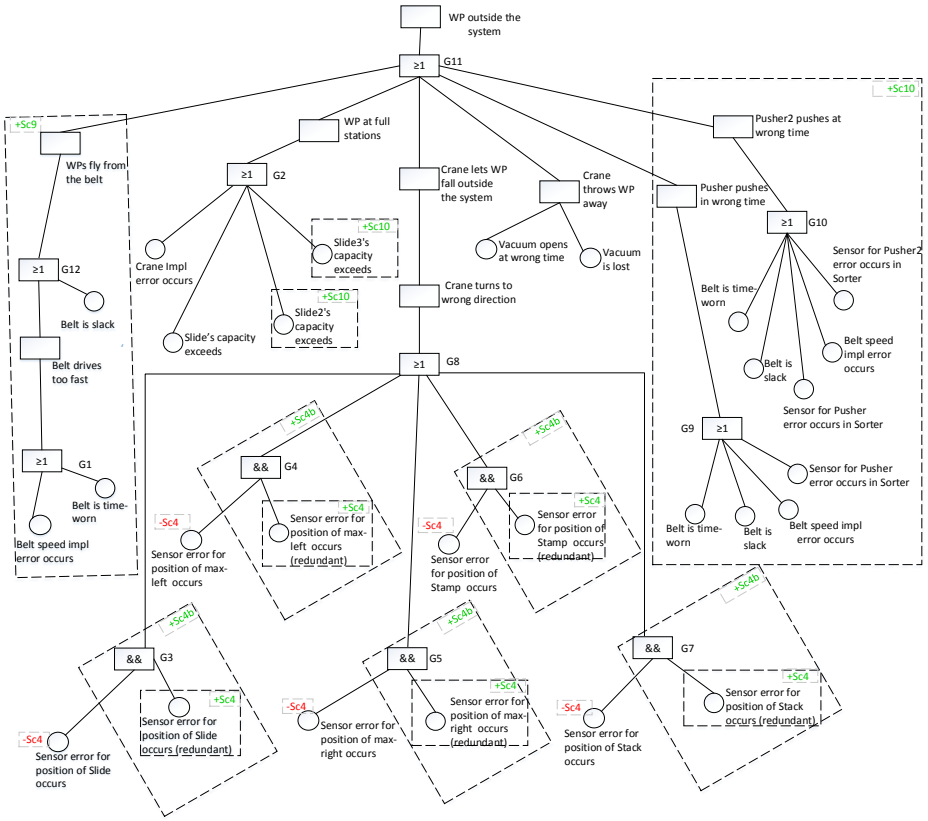
The case study system is a laboratory plant, called pick and place unit (PPU). The PPU mimics an industrial plant by moving so-called work pieces (WPs) between different working positions where they are stored or processed. Out of the 14 evolution scenarios that have been defined for the PPU [4], we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

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

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



(a) SA model of the PPU



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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Identified Relations Between Model Changes

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

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

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

Table 1. Mapping of scenarios and model changes

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

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

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

4 Conclusion and Future Work

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

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

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

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

References

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