Process Components for Quality Evaluation and Quality Improvement

Lars Grunske[1] and Roland Neumann[2]

[1]Boeing Postdoctoral Research Fellow at the School of Information Technology & Electrical Engineering, University of Queensland, Brisbane, QLD 4072, Room 458 IT Building
grunske@itee.uq.edu.au
[2]Research Assistant at the Department of Software Engineering and Quality Management, Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam, Prof.-Dr.-Helmert-Straße 2-3, D-14482 Potsdam (Germany)
roland.neumann@hpi.uni-potsdam.de

Abstract. Processes and methods used for software construction have a high influence on the quality of the resulting software product. Therefore, the research in the field of method engineering should be more focused on quality aspects. Due to this reason, we propose in this paper, process components for quality evaluation and quality improvement. These process components can be stored in a repository and the instances of these process components can be used in process frameworks (e. g. OPEN Process Framework).

1 Introduction

Constructing software systems with reusable components has become a popular approach for several reasons, including economical ones like cost reduction and shorter time to market. Similar to the software components, the tasks (phases, stages, cycles, etc.) that are used for the construction of the software system can be specified by process components. These process components can be stored in a repository, as described in the OPEN (Object-oriented Process Environment and Notation) Process Framework [12, 13], and can be used to construct a personalized process model. For this purpose, process components must be selected from the repository and plugged together.

In this paper, we will focus on such a personalized process model for mission-critical systems, like systems in the automotive, avionic, medicine or railway sector. These systems have important requirements relating to quality characteristics (or non-functional properties NFPs) as standardized in [15]. For technical software, the NFPs safety, availability, reliability and temporal correctness are most important [20]. As recommended in [4, 28] the quality assurance of these quality requirements must be part of the development process. Furthermore, due to economical reasons, activities to improve the quality of the system under construction must be made as early as possible in the construction process of the system.

The best phase is obviously immediately after the construction of the software/hardware architecture, because at this point developers are able to assess or forecast the quality properties for the first time. Consequently, we will introduce in this paper a

process component for the quality evaluation and improvement in the architectural design phase. This process component will be specified in detail in this paper. Furthermore, the activities and techniques, which are relevant for the practical application, are reviewed.

The rest of the paper is organized as follows: In Section 2 we introduce the preliminaries for component-based software engineering and the basic concepts of the quality characteristics: safety, availability, reliability, and temporal correctness. Thereafter in section 3, a process component for the quality improvement of a software system at an architectural level is introduced.

This process component contains two work units "Quality Evaluation" and "Quality Improvement by Architecture Refactoring" which are described in Section 4 and Section 5. Furthermore, in these sections the basic activities and techniques, which are used within the process components, are reviewed. In Section 6, we present a tool called BALANCE that supports the application of the introduced process component. Finally, section 7 concludes and discusses some remaining issues for future research.

2 Preliminaries

2.1 Component Based Software Engineering

Building software systems with self-contained and exchangeable components is a precondition for efficient modeling and reuse, which are both key elements of mature engineering disciplines [26,27]. Thus, many current design approaches divide systems recursively into components (sometimes called actors or capsules), which are instances of component-classes.

Components are encapsulated entities that hide their internal details and communication with their environment is only possible via ports. The selected model implicitly determines the kind of information that is transferred via ports; examples are discrete event signals, continuous data streams, or any kind of service requests and the corresponding responses. In many models a transmission of different messages or services across the same port is allowed .

Ports or their associated services may be directed (in or out). In this case, they must be connected as complementary pairs (input to output, service provider to service consumer etc.) and the associated semantics is that information flows from a source component to a target component or that a service is required by a client component and provided by a server component.

The architecture of the system, graphically depicted by the structure diagram, specifies how a higher-level component is built of lower-level components and how these can interact during the runtime of the system. Therefore, it must be described which ports of the components must be connected. Two basic connection mechanisms can be distinguished, connection and binding. The difference is that a connection interconnects two ports on the same hierarchical level, whereas a binding interconnects two ports in different hierarchical levels, i.e. a port of a subcomponent with a port of its enclosing component [26].

As an example, Figure 1 depicts the structure specification of a component class C with two ports. This component contains two subcomponents Sub1 of class C1 and Sub2

of class C2, both of them possessing two ports. The edges between the ports represent the communication links. Thereby the edge between the ports Sub1.p1 and the port Sub2.p1 is a connection and the edges between the other ports are bindings, because they link different hierarchy levels.

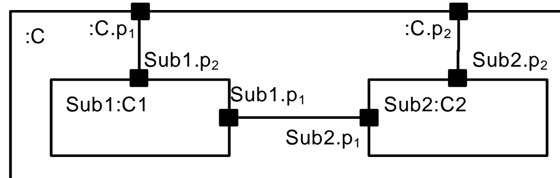

Fig. 1. Structure Specification Example

2.2 Quality Characteristics: Safety, Availability, Reliability, Maintainability and Temporal Correctness

Based on the definition in [1, 3, 17], availability is the readiness for correct service, reliability is the continuity of correct service, safety is the absence of catastrophic failures, maintainability is the ability to undergo repairs to return to the correct service and temporal correctness is the timeliness of a correct service. All these definitions refer to the correct service as specified in the requirement specification and the deviations from this correct service, which are failures.

However, the absence of these failures and the correctness of complex software-intensive technical systems cannot be guaranteed over the whole lifetime of the system [1, 18, 21]. The reasons for this are (a) the complexity of the software and hardware parts, which exponentiates the effort to prove the correctness and other similar quality assurance techniques (as for instance model checking) and (b) random or wear-out failures in the hardware parts that can result in an improper system behavior. Due to this, the tolerable probability of these failures must be specified by the quality requirements with stochastic measures. As an example, a safety requirement describes a safety critical failure (which can lead to a catastrophic accident) or a hazard together with its tolerable failure probability (TFP) or tolerable hazard probability (THP). As another example, reliability and maintainability requirements can be specified by the desired mean time to failure (MTTF) and the desired mean time to repair (MTTR).

In the development process, the fulfillment of the quality requirements depends mostly on the architecture of the system. Thereby especially the quality of the selected components and the usage of redundancy structures, failure handling and failure detection concepts are important.

3 The Quality Improvement Cycle

To improve the quality characteristics of software architectures we propose a cyclic process in this section. It is depicted in Figure 2 (for a detailed description of this process, we refer to [2] and [9].).

The precondition for the process application is an architectural specification that fulfils all functional requirements. Based on this functional correct specification, the quality characteristics are determined and checked against their quality requirements by a work unit called quality evaluation. If the architectural specification does not meet its quality requirements, the software architecture must be restructured by the application of transformation operators. These transformation operators, also called architectural refactorings, should improve the desired quality characteristics without changing the functional behavior. Thus, the architectural specification is still correct after the transformation. However, due to the complexity of today's software systems and the complexity relations between the quality characteristics, the influence of the transformation operator to all quality characteristics cannot be forecasted. Due to this, the quality can only be determined, with a quality evaluation of the new architecture specification. As a result, that transformation may be needed to revert, if the quality characteristics not improve. Another problem could be a negative influence of transformation operators to other unfocussed quality attributes. Due to this, an optimization problem must be solved within the process application, where the producers (quality assurance and architecture team) try to find a specification that fulfills most quality requirements.

The presented quality improvement cycle can be terminated, if all quality characteristics meet their corresponding requirements. If the cyclic process is terminated in this way, the resulting work product is an architectural specification that fulfills all functional and quality requirements and the system development can proceed with the next process step or development phase. However, due to the reason that some quality requirements cant be fulfilled by the system under development or that some quality requirement are in conflict to each other, the process must be terminated in an alternative way, like e.g. a fixed number of iterations or by the identification of a local optima.

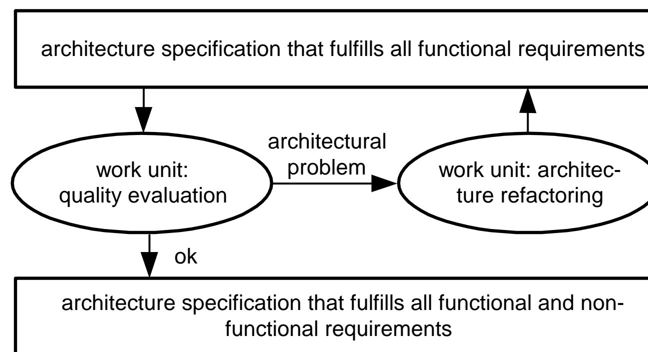

Fig. 2. The High-Level Quality Improvement Cycle

4 Work Unit: Quality Evaluation

The goal of the work unit: "Quality Evaluation" is to determine (forecast) the quality characteristics of an architectural specification and to check if these quality characteristics will meet their quality requirements. Due to this, the result of this work unit is only a Boolean value, describing whether or not the architecture fulfills the quality requirement.

To technically apply the work unit to a component-based system, each contained component must be annotated with an evaluation model that only specifies the relevant aspects and assumption of the quality characteristics of the component and how these quality characteristics are influenced by the environment. A selection of suitable evaluation models for the quality characteristics safety, reliability, maintainability, availability, and temporal correctness are presented in table 1. These models represent the current state-of-the-art (c.p. [1, 5, 7, 8, 19, 22, 24, and 25]). For component-based systems especially modular evaluation models like Component Fault Trees (CFT) [16], Reliability Block Diagrams (RBD) [19] and Scheduling Models (e.g. RMA, EDFA) [8] are suited for the quality evaluation. Based on these modular evaluation models an evaluation model for the complete system can be constructed based on the information flow and the architectural specification (c.p. [11], where such an algorithm is presented for CFTs). This system level evaluation model can be used to forecast the quality characteristics of the complete architecture and can be used to decide whether a system build with this architecture will meet their quality requirements.

Table 1. Quality characteristics and relevant evaluation models

Quality characteristic	Evaluation models
Safety	Fault Trees, Component Fault Trees (CFT), …
Reliability, maintainability, availability	Fault Trees, Component Fault Trees (CFT), Reliability Block Diagrams (RBD), Markov Chains, …
Temporal correctness	Scheduling Models (e.g. RMA, EDFA), End-to-End analysis, …

5 Work Unit: Quality Improvement by Architecture Refactoring

The basic activity of the work unit "Quality Improvement by Architecture Refactoring" is to transform the architecture by the application of a transformation operator. These transformation operators describe structural changes of the architecture in an abstract way. Thereby, a transformation operator can be seen as a pattern or an architectural refactoring that capsules domain-specific knowledge. For the goal-oriented application of such an architectural refactoring, a so-called "bad smell" must be detected first.

In the case of safety critical systems, components with a high failure probability and a high influence to safety critical system functions possess such a "bad smell". If fault trees are used as evaluation models [14], these components can be determined by a minimal cut-set analysis.

After the identification of the "bad smell", the transformation operator must be applied. Thereby, (a) components or connections can be removed or added, (b) connections can be redirected or (c) components can be replacement by components of another component type. To get an impression of the architecture refactorings we now introduce two transformation operators, which are used to improve the safety properties of a system in the case study in section 6 . A more detailed description of these and other transformation operators can be found in [10].

Multi Channel Redundancy with Voting The idea of the transformation operator Multi Channel Redundancy with Voting is to replace a component that does not fulfil its safety, availability or reliability requirements by multiple components which are executed on different hardware platforms and a comparator (*m-out-of-v voter*) that gets the inputs from the environment and generates multiple messages for the redundant components. Based on these messages the components compute the results and send them back to the comparator, which in turn chooses the message to be sent to the environment by a majority voting (*m-out-of-v*). For the realization of this pattern, often 3 components and a 2-out-of-3 voting are used [5, 6]. Due to the redundant structure, random and single point failures of the hardware platforms can be detected. Thus, the reliability of the system will be improved if the hardware platform of the comparator is more reliable than the hardware platforms of the redundant components.
Two Channel Redundancy The transformation operator Two Channel Redundancy replaces a component with two redundant channels operating in parallel on different hardware platforms. Each channel contains the replaced component and a component that is called *channel validation.* Both channels are getting all information from the environment, but one of them is active and one is passive. The active channel checks its operation with a *channel validation* component. The results of this validation are sent to the *channel validation* component of the passive channel. If a failure occurs, an error is detected, or if the active channel omits to send the results, the passive channel becomes active and replaces the former active channel. To check the correct operation of one channel the utilization of several strategies are possible [10]. By the application of this pattern, one channel can be still available in case of random or wear-out failures of the hardware platform of the other channel. Thus, availability can be increased by the application of this transformation operator.

6 Tool Support and Case Study

Supporting the proposed process, the tool Balance has been developed at the Hasso-Plattner-Institute in cooperation with the Siemens AG Cooperate Technology. This tool is introduced in this section and its applicability is presented with a case study.

6.1 Short Description of Balance

The tool Balance (as depicted in Fig. 3) can be used to model architectural specifications in ROOM (UML RT)[26], a subset of the UML 2.0, and COOL (an architecture description language that has be developed for embedded systems) [9]. These specifications can be directly designed by the user or they can be extracted from a commercial tool like IBM/Rational Rose RT. To further apply the above described process components the software and hardware components must be annotated with modular evaluation models for the relevant quality characteristics and the quality requirements must be specified. Based on this information the tool evaluates the quality characteristics of the system and proposes a set of transformations (refactorings) that will improve the architecture.

6.2 Case Study: Level Crossing Control System

The practicability of the tool is now presented by modeling a simplified version of a control system for a level crossing (c.p. Fig. 3). This control system contains the components *LevelCrossingControl, GateControl, TrainSignalControl, GateSensorManager* and *TrainSensorManager* which are not further decomposed. The *LevelCrossingControl* component is the controller of the level crossing system. This component can send messages to the *GateControl* component to open or close the Gates and to the *TrainSignal-Control* component to allow or deny the passage for an arriving train. To get the information from the environment the *LevelCrossingControl* component utilizes two sensors: One sensor determines the state of the gates and the other detects an arriving train and checks its progress through the level crossing section. These sensors are controlled by the *GateSensorManager* and *TrainSensorManager* component.

The objective of this case study is to improve the safety properties of the level crossing control system. These safety properties are evaluated by annotating each component with a CFT (component fault tree) [16]. Such CFT describes the failure behavior of the component. It contains a set of outputs, also called as output failure ports, defining all concrete failure types that can be caused by the component. The output failures can be caused either by an internal fault or by an external failure of the environment or another component. The external failures that can influence the correct behavior of the component are specified with a set of inputs, also called as input failure ports. The internal structure of an encapsulated fault tree is specified similar to normal fault trees as a Boolean function.

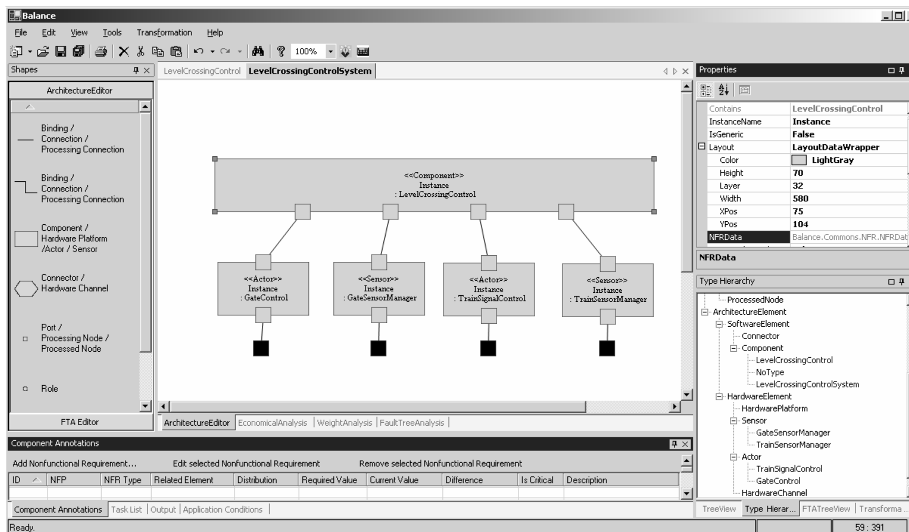

Fig. 3. Structure specification of the level crossing example within the tool Balance

Starting from the structural specification of the system an encapsulated fault tree can be constructed by taking a closer look at the messages that can be sent or received by a component. Each received message can be an input failure port and each sent message can be an output failure port of the encapsulated fault tree. The internal structure of fault

trees can be determined by structured techniques like (IF)-FMEA, HiP-HOPS or HAZOPS (c.p. [19, 24]). Based on the encapsulated fault trees, a fault tree for the complete architecture can be constructed with composition based techniques, as presented in [11]. The resulting fault tree of the level crossing control system is presented in Figure 4. It contains the encapsulated fault trees of the component and connects them with respect to the message flow in the system.

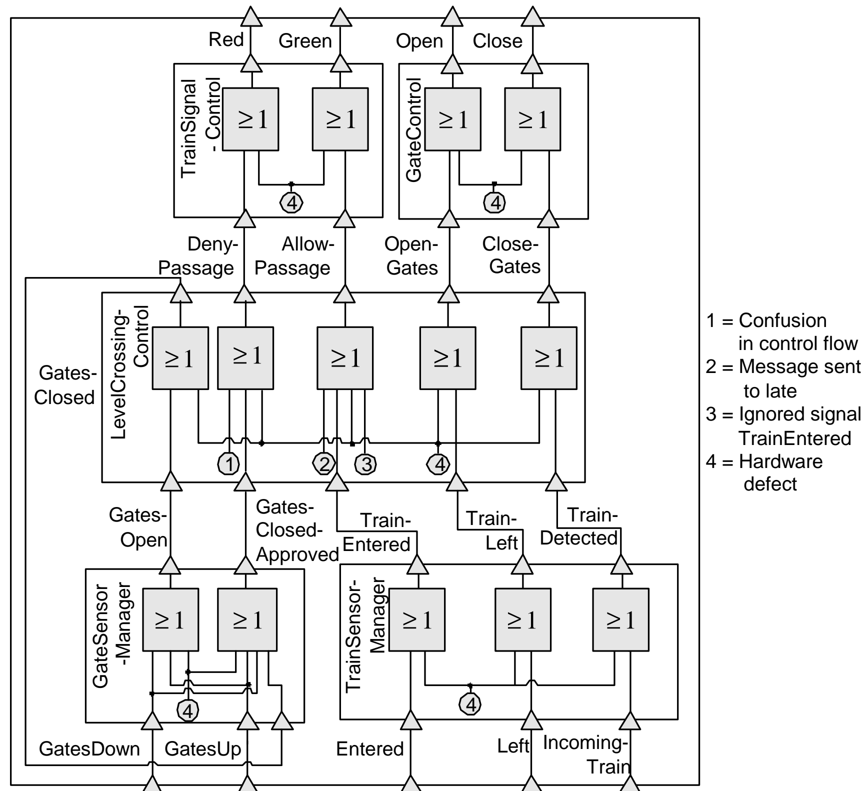

Fig. 4. Fault tree of the level crossing control system

Based on this fault tree, the system level failures responsible for accidents can be identified. In the level crossing control system the relevant safety critical system failures are (1) to send a faulty green signal to the train when the gates are open or (2) to open the gates when a train is in the level crossing section. The probabilities of these safety critical failures can be determined based on the fault tree and the probabilities of the internal faults as well as the system-level input failures given in Table 2. These probabilities are 1,099 $*10^{-5}$ for the first and 4,999 $*10^{-6}$ for the second critical failure. To reduce this failure probability the architecture specification must be restructured. In this manner, the patterns *Two Channel Redundancy* and *Multi Channel Redundancy with Voting* must be applied.

Table 2. Estimated probabilities of the internal faults and the system level input failures

Assumed internal faults or system level input failures	Failure or fault probability in 2000 hours mission time
TrainSignalControl.HardwareDefect	$1{,}0 * 10^{-6}$
GateControl.HardwareDefect	$1{,}0 * 10^{-6}$
LevelCrossingControl.ConfusionInControlFlow	$0{,}05 * 10^{-6}$
LevelCrossingControl.MessageSentToLate	$0{,}05 * 10^{-6}$
LevelCrossingControl.IgnoredSignalTrainEntered	$0{,}05 * 10^{-6}$
LevelCrossingControl.HardwareDefect	$1{,}0 * 10^{-6}$
GateSensorManager.HardwareDefect	$1{,}0 * 10^{-6}$
TrainSensorManager.HardwareDefect	$1{,}0 * 10^{-6}$
InputFailure.GatesDown	$2{,}0 * 10^{-6}$
InputFailure.GatesUp	$2{,}0 * 10^{-6}$
InputFailure.Entered	$2{,}0 * 10^{-6}$
InputFailure.Left	$2{,}0 * 10^{-6}$
InputFailure.IncomingTrain	$2{,}0 * 10^{-6}$

If we assume a constant probability of $0.5 * 10^{-6}$ for an internal fault in the *voter* or *validation* component, the probabilities of the critical failures will be reduced by the application of the transformation operators as presented in Table 3. Based on these results the best architecture transformation is the application of the Multi Channel Redundancy operator to the level crossing control component. The resulting architecture after this transformation is presented in Figure 5. The application of the Two Channel Redundancy does not reduce the probabilities of the safety critical failures. The reason for this is the combined probability of an internal fault of the *validation* component, which is nearly identical to the probability of an occurrence of a hardware defect in the target component.

Table 3. Application of the transformation operators to the level crossing control system

Transformation (Transformed component)	Probability of the safety critical failure “A green signal is sent to the train when the gates are open”	Probability of the safety critical failure “The gates are opened when a train is in the level crossing section
Original Architecture	$\sim 1{,}099 * 10^{-5}$	$\sim 4{,}999 * 10^{-6}$
Multi Channel Redundancy with Voting (TrainSignalControl)	$\sim 1{,}1 * 10^{-5}$	$\sim 4{,}5 * 10^{-6}$
Multi Channel Redundancy with Voting (GateControl)	$\sim 4{,}6 * 10^{-6}$	$\sim 5{,}0 * 10^{-6}$

Multi Channel Redundancy with Voting (LevelCrossingControl)	~4,5 $*10^{-6}$	~4,5 $*10^{-6}$
Multi Channel Redundancy with Voting (GateSensorManager)	~9,6 $*10^{-6}$	~5,0 $*10^{-6}$
Multi Channel Redundancy with Voting (TrainSensorManager)	~9,6 $*10^{-6}$	~4,5 $*10^{-6}$
Two Channel Redundancy (TrainSignalControl)	~1,1 $*10^{-5}$	~5,0 $*10^{-6}$
Two Channel Redundancy (GateControl)	~1,1 $*10^{-5}$	~5,0 $*10^{-6}$
Two Channel Redundancy (LevelCrossingControl)	~9,6 $*10^{-6}$	~1,1 $*10^{-5}$
Two Channel Redundancy (GateSensorManager)	~1,1 $*10^{-5}$	~5,0 $*10^{-6}$
Two Channel Redundancy (TrainSensorManager)	~1,1 $*10^{-5}$	~5,0 $*10^{-6}$

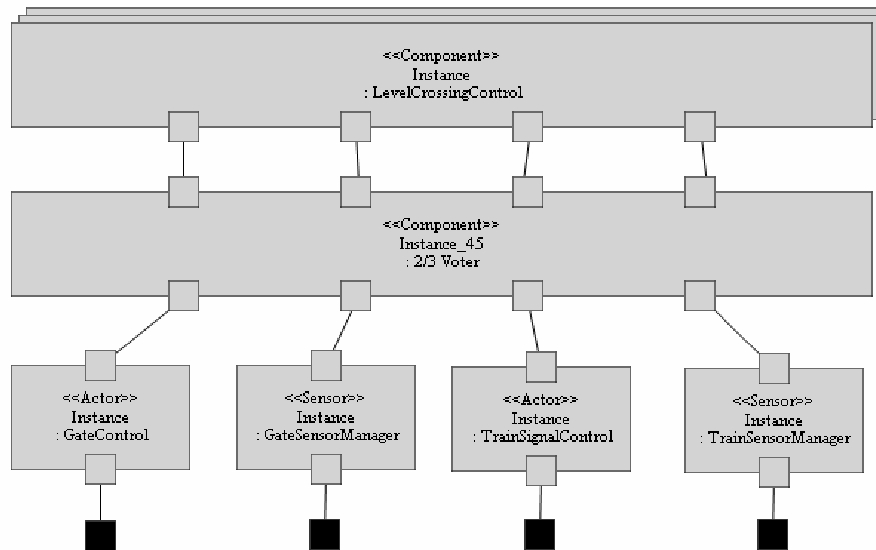

Fig. 5. Architecture specification after the application of the architecture pattern: Multi Channel Redundancy with Voting to the Level Crossing Control component

7 Conclusion and Future Work

In this paper, we have proposed a process component for the quality evaluation and quality improvement. For this process component, the relevant work units and techniques are presented and reviewed. Additionally, the practical applicability of this process component is shown with the case study of a level-crossing control system, which safety properties are improved supported by the tool Balance. Thereby the probabilities of the safety-critical failures (hazards) "A green signal is sent to the train when the gates are open" and "The gates are opened when a train is in the level crossing section" are reduced.

The proposed process component is especially focused for component-based software engineering processes. Due to this, it remains for future work how this process component can be used for object-oriented or agent-oriented software engineering methodologies.

Furthermore, the practical applicability of the process component and the tool Balance should be investigated in more case studies under real world conditions. Currently three industrial case studies for the Siemens AG, the FhG First and the DLR (German Aerospace Center) in the field of reactor control systems, satellite control systems and railway transportation systems are completed. Although technical aspects have to be further investigated, the presented process helps to improve the quality aspects of a software architecture.

References

1. Birolini A.: Reliability engineering: theory and practice, New York, Springer, 1999
2. Bosch J., P. Molin.: Software Architecture Design, Evaluation and Transformation. In IEEE Engineering of Computer Based Systems Symposium. (1999)
3. Bondavalli A., Simoncini L.: Failure Classification with Respect to Detection, in: Predictably Dependable Computing Systems, Task B, Vol. 2, May (1990)
4. CENELEC:, Railway applications The specification and demonstration of dependability, reliability, availability, maintainability and safety (RAMS), European Committee for Electrotechnical Standardisation, Brussels, Standard EN 50126, 128, 129, (2000-2002)
5. Douglas B. P.: Doing Hard Time, Reading, Massachusetts, Addison Wesley. 1999
6. Douglas, B. P.: Real-Time Design Patterns. Reading Massachusetts, Addison Wesley. 2002
7. Fenelon P., McDermid J.A., Nicholson M., Pumfrey D. J.: Towards Integrated Safety Analysis and Design, ACM Applied Computing Review, (1994).
8. Gomaa H. Designing Concurrent, Distributed, and Real-Time Applications with UML. Reading, Massachusetts: Addison-Wesley Publishing Company, (2000)
9. Grunske L.: Automated Software Architecture Evolution with Hypergraph Transformation, In Proceedings of the 7th International IASTED on Conference Software Engineering and Application (SEA 03), Marina del Ray, (2003,) pp. 613-621
10. Grunske L.: Transformational Patterns for the Improvement of Safety Properties in Architectural Specification. In J. Bargary, C. Haskins (Eds.), Proceedings of the Viking PLOP 03. Microsoft Business Press, (2003)
11. Grunske L.: Annotation of Component Specifications with Modular Analysis Models for Safety Properties, In Proceedings of the 1st International Workshop on Component Engineering Methodology, (WCEM 03), (2003), pp. 31-41
12. Henderson-Sellers B.: Method engineering for OO systems development. Commun. ACM 46(10) pp. 73-78, (2003)

13. Henderson-Sellers B.: Process Metamodelling and Process Construction: Examples Using the OPEN Process Framework (OPF). Ann. Software Eng. 14 (1-4), pp.341-362, (2002)
14. IEC 61025: International Standard IEC 61025 Fault Tree Analysis. International Electrotechnical Commission. Geneva, (1990)
15. ISO/IEC standard 9126-1.2: Information Technology – Software product quality – Part 1: Quality model, (1998)
16. Kaiser B., Liggesmeyer P. Mäckel, O.: A New Component Concept for Fault Trees. in Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Adelaide, (2003)
17. Laprie J.C.(ed.): Dependability: Basic Concepts and Associated Terminology. Vol.5, Dependable Computing and Fault-Tolerant Systems Series,Vienna: Springer (1992)
18. Leveson N. G.: Safeware: System Safety and Computers. Addison-Wesley, (1995)
19. Liggesmeyer P.: Qualitätssicherung softwareintensiver technischer Systeme. Heidelberg: Spektrum-Akademischer-Verlag. (2000)
20. Neumann, R., Grunske, L., Kaiser, B., Hierarchical software quality models – a step towards quantifying non-functional properties, proc. 12th Int.l Workshop on Software Measurement, Magdeburg, Germany, pp. 107-124, (2002)
21. Musa, J.D.; Iannino, A.; Okumoto, K.: Software Reliability - Measurement, Prediction, Application, McGraw-Hill International Editions, (1987)
22. Papadopoulos Y., McDermid J.A., Sasse R., Heiner G.: Analysis and Synthesis of the Behavior of Complex Programmable Electronic Systems in Conditions of Failure, Reliability Engineering and System Safety, 71(3), Elsevier Science, (2001) 229-247.
23. Papadopoulos Y., McDermid J. A.: Hierarchically Performed Hazard Origin and Propagation Studies, SAFECOMP '99, 18th Int. Conf. on Computer Safety, Reliability and Security, Toulouse, LNCS, 1698 (1999) 139-152
24. Pumfrey D. J.: The Principled Design of Computer System Safety Analyses, Dissertation, University of York, (1999).
25. Reussner,R., Schmidt, H., Poernomo, I.:. Reliability Prediction for Component-Based Software Architectures, Journal of Systems and Software, 66(3), Elsevier, The Nether-lands, (2003) 241--252
26. Selic B., Gullekson G., Ward P. T.: Real-Time Object-Oriented Modeling. Wiley, New York, (1994)
27. Szyperski C.: Component Software. Beyond Object-Oriented Programming. ACM Press/ Addison Wesley, (1998)
28. Vesely, W. E., Goldberg, F. F., Roberts, N. H.,. Haasl, D. F.: Fault Tree Handbook. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, (1981)