Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 558–575
The Journal of Systems and Software
www.elsevier.com/locate/jss

Software architecture reliability analysis using failure scenarios ☆

Bedir Tekinerdogan *, Hasan Sozer, Mehmet Aksit

Department of Computer Science, University of Twente, P.O. Box 217 7500, AE Enschede, The Netherlands

Received 14 July 2006; received in revised form 11 October 2007; accepted 30 October 2007
Available online 17 November 2007

Abstract

With the increasing size and complexity of software in embedded systems, software has now become a primary threat for the reliability. Several mature conventional reliability engineering techniques exist in literature but traditionally these have primarily addressed failures in hardware components and usually assume the availability of a running system. Software architecture analysis methods aim to analyze the quality of software-intensive system early at the software architecture design level and before a system is implemented. We propose a Software Architecture Reliability Analysis Approach (SARAH) that benefits from mature reliability engineering techniques and scenario-based software architecture analysis to provide an early software reliability analysis at the architecture design level. SARAH defines the notion of failure scenario model that is based on the Failure Modes and Effects Analysis method (FMEA) in the reliability engineering domain. The failure scenario model is applied to represent so-called failure scenarios that are utilized to derive fault tree sets (FTS). Fault tree sets are utilized to provide a severity analysis for the overall software architecture and the individual architectural elements. Despite conventional reliability analysis techniques which prioritize failures based on criteria such as safety concerns, in SARAH failure scenarios are prioritized based on severity from the end-user perspective. SARAH results in a failure analysis report that can be utilized to identify architectural tactics for improving the reliability of the software architecture. The approach is illustrated using an industrial case for analyzing reliability of the software architecture of the next release of a Digital TV.

Keywords: Reliability analysis; Scenario-based architectural evaluation; FMEA; Fault trees

1. Introduction

Three important trends can be observed in the development of embedded systems. First, the demand for products with more and more functionality has increased due to the high industrial competition and the advances in hardware and software technology. Second, the functionality provided by embedded systems is more and more shifting from hardware to software. Third, embedded systems become more and more open. Increasingly, the functionality is host to multiple parties and is not solely developed by just one manufacturer anymore. Furthermore, embedded systems are integrated in networked environments that affect these systems in ways that might not have been foreseen during their construction.

These three trends complicate the design and implementation of embedded systems and as such put serious challenges on assuring the desired software quality factors such as security, reliability, performance and availability. Since embedded systems are now largely defined and controlled by software, the required quality factors are likewise more dependent on the quality of the adopted software and to a lesser degree on the hardware.

An important quality factor in the development of embedded systems is reliability. Given the above trends of increased functionality, complexity and openness, it is expected that the risk of failures in embedded systems can increase to a mission critical level. Obviously, to minimize this risk it is required that appropriate reliability

☆ This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.
* Corresponding author. Tel.: +31 53 489 5683; fax: +31 53 489 3247.
E-mail address: bedir@cs.utwente.nl (B. Tekinerdogan).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.10.029

analysis and design techniques are provided so that potential failures can be predicted in time and the system can be designed to be recoverable from these failures.

Several useful conventional reliability engineering techniques exist already in literature to analyze and design reliable systems. A comprehensive overview of these approaches is, for example, given in Dugan (1996) and Avizienis et al. (2001). From a close study to these approaches we can derive two key observations. First of all, these conventional reliability analysis techniques have primarily focused on failures in hardware components and less on software components. In fact this is not so strange because historically, software formed only a small part of embedded systems and basically hardware components defined the quality of the system. The second observation shows that conventional reliability analysis techniques very often assume a completed running system and do not focus on reliability analysis before the system is actually implemented.

It is now widely recognized that reliability analysis should not be limited to hardware components but should also cover software components. In addition, there is an increasing agreement that reliability analysis should not just be performed at the code level but also earlier during the development of the system. In this context, the system that needs to be implemented is typically the next release of a product in a given product line. The new release usually includes an enhanced set of features or has improved quality. Thus, early analysis here should be interpreted as the analysis of the next release of a system rather than the analysis of a system that will be developed completely from scratch.

The early analysis of software quality has been in particular promoted in the software architecture analysis domain. Because implementing the software architecture is a costly process it is important to predict the quality of the system and identify potential risks, before committing enormous organizational resources (Dobrica and Niemela, 2002). Similarly, it is of importance to analyze the potential failures that might reduce reliability at the architecture design level before the system is implemented.

For providing an early reliability analysis that covers software components, it is worthwhile to utilize both the results from software architecture analysis and conventional reliability analysis approaches.

Software architecture analysis (Clements et al., 2002) usually includes static analysis of formal architectural models (Medvidovic and Taylor, 2000) or scenario-based architecture analysis methods as described in Dobrica and Niemela (2002). Scenario-based analysis approaches have been widely applied and validated over the past several years. Several scenario-based architecture analysis methods have been developed each focusing on particular quality attributes (Dobrica and Niemela, 2002). In general, scenario-based analysis methods take as input a model of the architecture and measure the impact of the predefined scenarios on it in order to identify the potential risks and the sensitive points of the architecture. A scenario is generally considered to be a brief description of some anticipated or desired use of the system (Clements et al., 2002). Hereby, it is implicitly assumed that scenarios correspond to the particular quality attributes that need to be analyzed.

In the conventional reliability analysis domain, a well-known and established mature approach is the Failure Mode and Effect Analysis (FMEA) method. The purpose of FMEA is to identify possible failure modes of the system components and evaluate their impact on the system performance. FMEA is usually utilized together with fault tree analysis (FTA) that is developed based on the results of the FMEA. In FTA so-called fault trees define causal and logical relationships between faults and their causes. Although both FMEA and FTA are established techniques in reliability engineering they have been basically used to analyze failures of hardware components.

We propose the software architecture reliability analysis (SARAH) approach that benefits from both reliability engineering and scenario-based software architecture analysis to provide an early reliability analysis of next product releases. SARAH defines the notion of failure scenario model that is based on the FMEA and FTA. The failure scenario model is applied to represent so-called failure scenarios that indicate potential failures in the software system. These failure scenarios are utilized to derive a fault tree set (FTS), which shows the causal and logical connections among the failures.

To a large extent SARAH integrates the best practices of the conventional and stable reliability analysis techniques with the scenario-based software architecture analysis approaches. Besides this SARAH provides another distinguishing property by focusing on user perceived reliability. Conventional reliability analysis techniques prioritize failures usually based on criteria such as safety concerns since safety critical systems (airplanes, nuclear power plants, etc.) have been the main focus for reliability analysis. However, we focus on the consumer electronics domain in which safety is less/not an issue. Instead, user perception of failures is adopted as the primary concern to enhance reliability. The underlying assumption here is that failures that are not perceived by the user are less important and should be tolerated as much as possible. In SARAH the focus on user perceived reliability manifests itself in the prioritization and analysis of failure scenarios based on a user perception model. As a result SARAH aims to improve in particular the user-perceived reliability.

SARAH results in a failure analysis report that defines the sensitive elements of the architecture and provides information on the type of failures that might frequently happening. The reliability analysis forms the key input to identify architectural tactics (Bachman et al., 2003) for adjusting the architecture and improving its reliability, which forms the last phase in SARAH.

We thus provide a reliability analysis approach aimed for analyzing the reliability of future releases before they are developed with the basic focus on reliability from a user perspective. SARAH is illustrated using an industrial case

for analyzing user-perceived reliability of future releases of Digital TVs. Besides presenting SARAH as an integrated and novel software architecture reliability approach, an additional goal of this paper is to present the results and the experiences of applying this method within an industrial context. As such while explaining the approach we will also discuss our experience and obstacles in applying the approach.

The remainder of this paper is organized as follows. In Section 2, we present the industrial case in which Digital TV architecture is introduced. This example will be used throughout the paper to illustrate the steps of SARAH and to discuss our experiences in the industrial context. Sections 3–6 explain the steps of SARAH and apply this to the reliability analysis of the software architecture of the Digital TV. Section 3 presents the top-level process. Section 4 presents the definition of the software architecture and the scenarios from FMEA. Section 5 presents the analysis of the architecture and the definition of the failure analysis report. Section 6 discusses the identification of architectural tactics and the refactoring of the architecture based on the architectural analysis results. Section 7 discusses the lessons learned. Section 8 provides the related work and finally Section 9 concludes the paper.

2. Industrial case: Digital TV (DTV) software architecture

At the Embedded Systems Institute (ESI), the TRADER (Television Related Architecture and Design to Enhance Reliability) project is carried out together with NXP Semiconductors, Philips Consumer Electronics, and several other academic and industrial partners (Trader project web site, 2005). The objective of the project is to develop methods and tools for ensuring reliability of digital television (DTV) sets. Similar to the general trends that can be observed for embedded systems, one can also observe three important trends for the DTV. Firstly, the TV is equipped with an increasing number of features after each release. Secondly, functionality is shifting from hardware to software implementations. The amount of software that is embedded in the TV is dramatically increasing leading to an increase in software complexity. Thirdly, the DTV will become open and be integrated in networks and connected to other devices.

Albeit reliability has always played a role in the design of TV systems, this has become a key concern within the current trends. To keep the reliability at least at the current levels it is required that reliability analysis techniques need to be enhanced. For a long period the reliability of the TV has been basically addressed at hardware level using run-time error handling techniques. However, in face of the current developments it has now been recognized that the error handling techniques in the TV system should also cover software failures, and reliability analysis techniques should be applied before the system has been implemented. The latter is required to anticipate on potential failures that cannot be solely addressed at the code level. Another important observation in the design of embedded systems and in particular the DTV is that it is not practically feasible to aim for a perfect system design that does not include any failures at all. Instead, it is recognized that the system can include some failures and the system needs to provide fault tolerance for this by defining appropriate reliability analysis and detection techniques. In the DTV design in the TRADER project, failures that can be directly observed by the user require a special attention. Failures that are not directly perceived by the user can be tolerated to some extent. Because TRADER aims to anticipate also on failures in future releases it is important that reliability analysis and detection techniques are defined at the design level. From this perspective one of the key aims in TRADER is the design of fault tolerant software architecture with respect to the perception of TV users.

A conceptual architecture of DTV is depicted in Fig. 1,[1] which will be referred to throughout the paper. The design mainly comprises two layers. The bottom layer, namely the streaming layer, involves modules taking part in streaming of audio/video information. The upper layer consists of applications, utilities and modules that control the streaming process. In the following, we briefly explain some of the important modules that are part of the architecture. For brevity, the modules for decoding and processing audio/video signals are not explained here.

Application Manager (*AMR*) initiates and controls execution of both resident and downloaded applications in the system. It keeps track of application states, user modes and redirects commands/information to specific applications or controllers accordingly.

Audio Controller (*AC*), controls audio features like volume level, bass and treble based on commands received from AMR.

Command Handler (*CH*) interprets externally received signals (i.e. through keypad or remote control) and sends corresponding commands to AMR.

Communication Manager (*CMR*) employs protocols for providing communication with external devices.

Conditional Access (*CA*) authorizes information that is presented to the user.

Content Browser (*CB*) presents and provides navigation of content residing in a connected external device.

Electronic Program Guide (*EPG*) presents and provides navigation of electronic program guide regarding a channel.

Graphics Controller (*GC*) is responsible for generation of graphical images corresponding to user interface elements.

Last State Manager (*LSM*) keeps track of last state of user preferences such as volume level and selected program.

Program Installer (*PI*) searches and registers programs together with channel information (i.e. frequency).

Program Manager (*PM*) tunes to a specific program based on commands received from AMR.

[1] Due to space limitation and confidentiality, we present a simplified DTV architecture and a representative set of scenarios only.

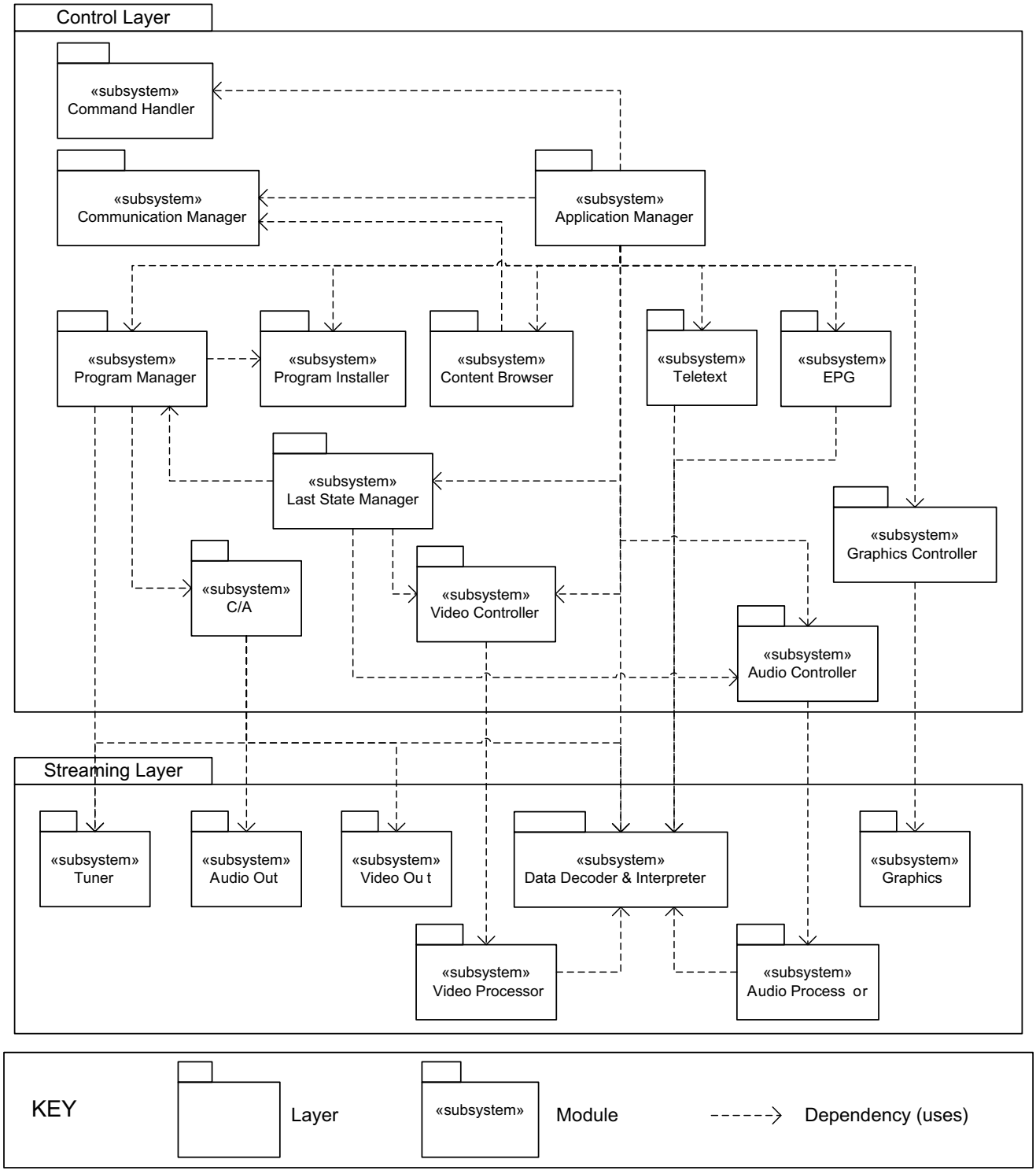


Fig. 1. Conceptual architecture of Digital TV.

Teletext (TXT) handles acquisition, interpretation and presentation of teletext pages.

Video Controller (*VC*) controls video features like scaling of the video frames based on commands received from AMR.

3. Top-level process of the analysis approach

For understanding and predicting quality requirements of the architectural design Bachman et al. identify four important requirements: (1) provide a specification of the

quality attribute requirements, (2) enumerate the architectural decisions to achieve the quality requirements, (3) couple the architectural decisions to the quality attribute requirements and finally (4) provide the means to compose the architectural decisions into a design (Bachman et al., 2003). SARAH is in alignment with these key assumptions. The focus in SARAH is the specification of the reliability quality attribute, the analysis of the architecture based on the provided model and the identification of the architectural tactics to adjust the architecture.

In SARAH, reliability is considered as the ability to cope with failures. The notion of failure has been extensively studied in the reliability engineering domain. Hereby, a *failure* is generally defined as an event that occurs when the delivered service of a system deviates from a correct service (Avizienis et al., 2001). A correct service is delivered when the service implements the system function. A service failure may occur because it does not comply with the functional specification, or because the specification did not adequately describe the required system function. An *error* is defined as the system state that is liable to lead to a failure and the cause of an error is called a *fault* (Avizienis et al., 2001).

SARAH adopts the view of failures from the reliability engineering domain. The steps of SARAH are presented as a UML activity diagram in Fig. 2. The approach consists of three basic processes: (1) *Definition*, (2) *Analysis* and (3) *Adjustment*. In the *definition process* the software architecture, the failure domain model, the failure scenarios, the fault trees and the severity values for failures are defined. Based on this input, in the *analysis process*, an architectural level analysis and an architectural element level analysis are performed. The results are presented in the failure analysis report. The failure analysis report is utilized in the *adjustment process* to identify the architectural tactics and adapt the software architecture. An architectural tactic is defined as a characterization of architectural decisions that are needed to achieve a desired quality attribute response (Bachman et al., 2003). In SARAH we are mainly interested in identifying architectural tactics to enhance reliability. In the following sections the main steps of the method will be explained in detail using the industrial case study.

4. Definition of the software architecture and failure scenarios

4.1. Describe the architecture

Similar to existing software architecture analysis methods SARAH starts with defining the software architecture. The definition includes the architectural elements and their relationships. Currently, the method itself does not presume to provide a particular architectural view (Clements et al., 2002) but in our project we have basically applied it to the module view. The architecture that we analyzed is depicted in Fig. 1.

4.2. Develop failure scenarios

SARAH is a scenario-based architectural analysis method, that is, scenarios are the basic means to analyze the architecture. SARAH defines the concept of *failure scenario* to analyze the architecture with respect to reliability. Failure scenarios are potential failures that could occur due

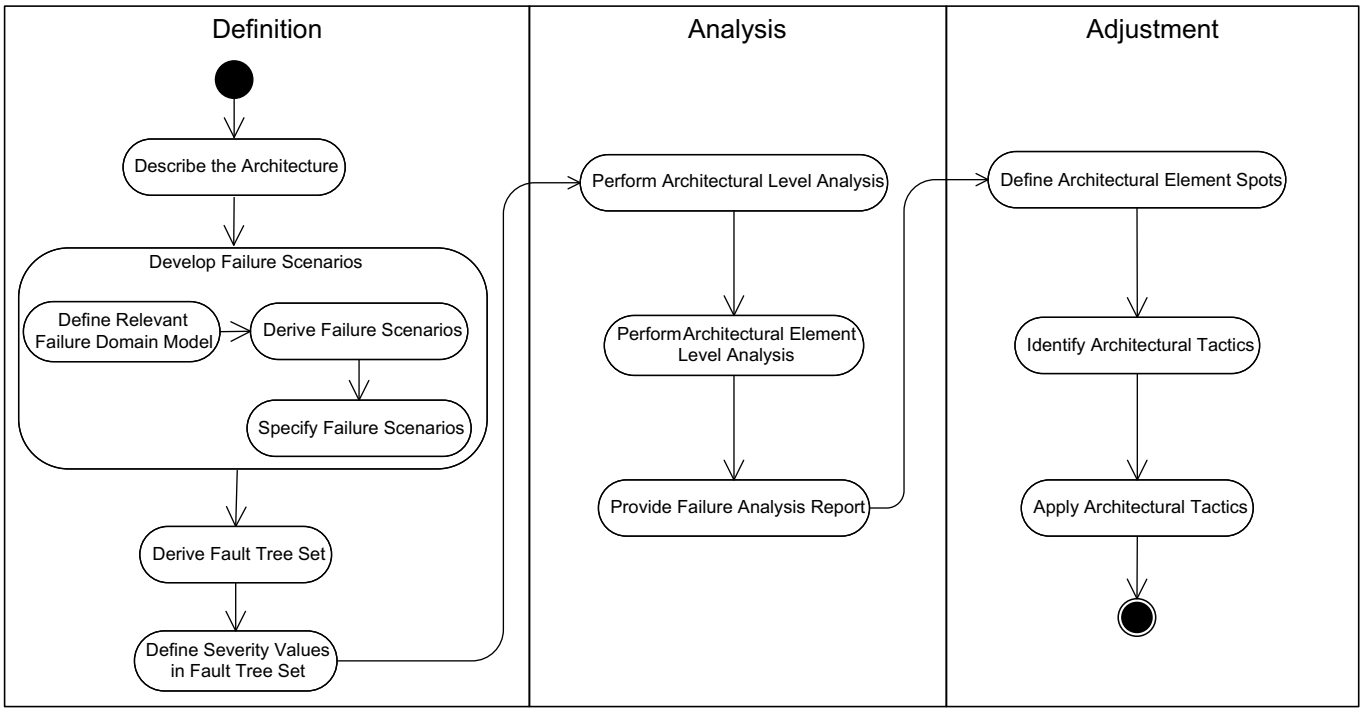


Fig. 2. Activity diagram of the software architecture reliability analysis method.

Table 1
Template for defining failure scenarios

FID	A numerical value to identify the failures (i.e. Failure ID)
AEID	An acronym defining the architectural element for which the failure scenario applies (i.e. Architectural Element ID)
Fault	The cause of the failure defining both the description of the cause and its features
Error	Description of how the element fails or the state of the element that leads to the failure together with its features
Failure	The effect of the failure defining both the description of the effect and its features

to internal or external causes within a given context. To specify the failure scenarios in a uniform and consistent manner a failure scenario template, as defined in Table 1 is adopted for specifying failure scenarios. The template is inspired from FMEA (Dugan, 1996). FMEA is a well known reliability analysis method for eliciting and evaluating potential risks of a system. In FMEA, five attributes of a failure scenario are identified; *failure id*, *related component*, *failure cause*, *failure mode* and *failure effect*. A *failure mode* is defined as the manner in which the element fails. A *failure effect* is the (undesirable) consequence of a failure mode. For clarity in SARAH *fault*, *error* and *failure* are used for the concepts *failure cause*, *failure mode* and *failure effect*, respectively. In SARAH failure scenarios are derived in two steps. First the relevant failure domain model is defined, then failure scenarios are derived from this failure domain. The following subsections describe these steps in detail.

4.2.1. Define relevant failure domain model

The failure scenario template can be adopted to derive scenarios in an ad hoc manner using free brainstorming sessions. However, it is not trivial to define the fault, error, or failure types. Hence, there is a high risk that several potential and relevant failure scenarios are missed or that other irrelevant failure scenarios are included. To define the space of relevant failures SARAH defines relevant domain model for faults, errors and failures using a systematic domain analysis process (Arrango, 1994). These domain models provide a first scoping of the potential scenarios. In fact, several researchers have already focused on modeling and classifying failures for embedded systems. Avizienis et al. (2001), for example, provide a nice overview of this related work and provide a comprehensive classification of faults, errors and failures. The provided domain classification by Avizienis et al., however, is rather broad,[2] and one can assume that for a given reliability analysis project not all the potential failures in this overall domain are relevant. Therefore, the given domain is further scoped by focusing only on the faults, errors and failures that are considered relevant for the actual project. Fig. 3, for example,

[2] Due to space limitations we do not show this domain model and refer the interested reader to the corresponding publication.

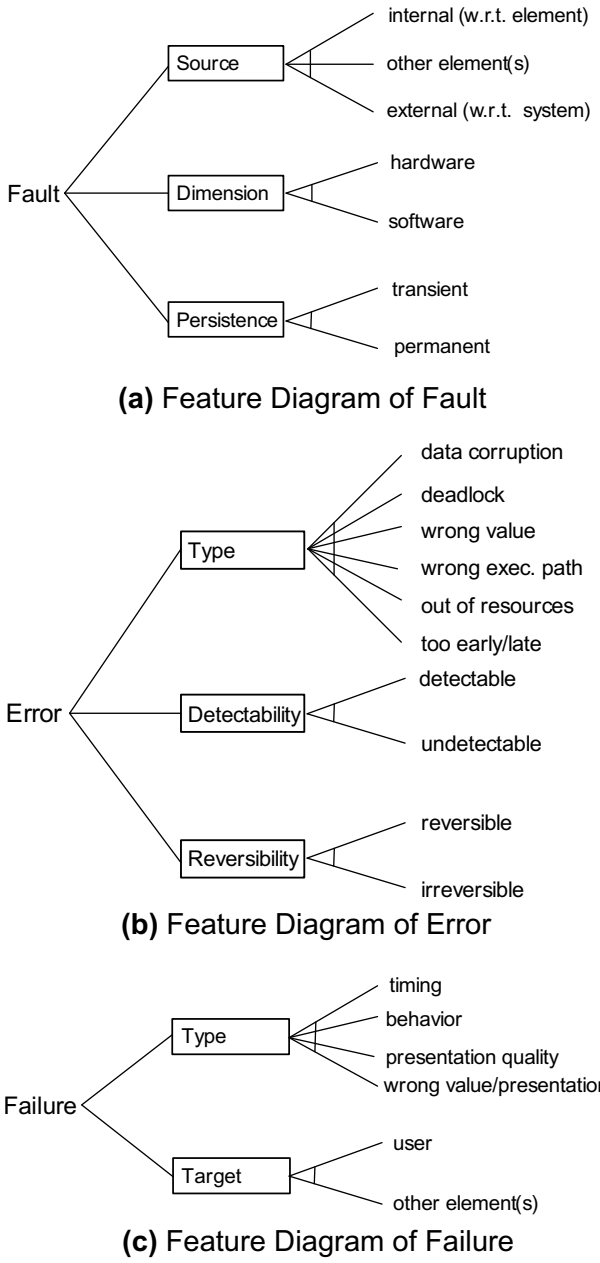


Fig. 3. Failure domain model.

defines the derived domain model that is considered relevant for the DTV project.

In Fig. 3a, a feature diagram of fault is presented, in which faults are identified according to their *source*, *dimension* and *persistence*. In SARAH, failure scenarios are defined per architectural element. For that reason, the source of the fault can be either (1) internal to the element in consideration, (2) caused by other element(s) of the system or (3) caused by external entities with respect to the system. Faults could be caused by software or hardware, and be transient or persistent. In Fig. 3b, the relevant features of an error are shown, which comprise the *type* of error together with its *detectability* and *reversibility* properties. Fig. 3c defines the features for failure, which includes

the features *type* and *target*. The *target* of a failure can be the user or other element(s) of the system.

The failure domain model of Fig. 3 has been derived after a thorough domain analysis and in cooperation with the domain experts in the project. In principle, for different project requirements one may come up with a slightly different domain model, but as we will show in the next sections this does not impact the steps in the analysis method itself. The key issue here is that failure scenarios are defined based on the FMEA model, in which their properties are represented by domain models that provide the scope for the project requirements.

4.2.2. Define failure scenarios

The domain model defines a system-independent specification of the space of failures that could occur. The number and type of failure scenarios is implicitly defined by the failure domain model, which define the scope of the relevant scenarios. In the fault domain model (feature diagram in Fig. 3a), for example, we can define faults based on three features, namely *Source, Dimension* and *Persistence*. The feature *Source* can have three different values, the features *Dimension* and *Persistence* 2 values. This means that the fault model captures $3 \times 2 \times 2 = 12$ different faults. Similarly, from the error domain model we can derive $6 \times 2 \times 2 = 24$ different errors, and $4 \times 2 = 8$ different failures are captured by the failure domain model. Since a scenario is a composition of selection of features from the failure domain model, we can state that for the given failure domain model in Fig. 3 in theory, $12 \times 8 \times 24 = 2304$ failure scenarios can be defined. However, not all the combinations instantiated from the failure domain are possible in practice. For example, in case the error type is "too late" then the error cannot be "reversible". If the failure type is "presentation quality" then the failure target can only be "user". To specify such kind of constraints for the given model in Fig. 3 usually *mutex* and *requires* predicates are used (Kang et al., 1990). For example, the given two example constraints are specified as follows:

```
Error.type.too_late mutex
Error.type.reversibility.reversible
Failure.type.presentation_quality
requires
Failure.target.user
```

Once the failure domain model together with the necessary constraints have been specified we can start defining failure scenarios for the architectural elements. For each architectural element we check the possible failure scenarios and define an additional description of the specific fault, error or failure. Table 2 provides, for example, a list of nine selected failure scenarios that have been derived for the reliability analysis of the DTV. In fact, for the example we have utilized 44 failure scenarios but due to space limitations we do not show these all. In Table 2 the five elements *FID*, *AEID*, *fault*, *error* and *failure* are represented as columns headings. Failure scenarios are represented in rows.

The failure ids represented by FID do not have a specific ordering but are only used to identify the failure scenarios. The column AEID includes acronyms of names of architectural elements to which the identifiers refer. The type of the architectural elements that are analyzed can vary depending on the architectural view utilized (Clements et al., 2002). In case of a deployment view, for instance, the architectural elements that will be analyzed will be the nodes. For component and connector view, the architectural elements will be components and connectors. In this paper, we focused on module view of the architecture, where the architectural elements are the implementation units (i.e. modules). In principle if an architectural element is represented as a first-class entity in the model and it affects the failure behavior, then it can be used in the analysis of SARAH.

The columns *fault*, *error* and *failure* describe the specific faults, errors and failures. Note that the separate features of the corresponding domain models are represented as keywords in the cells. For example, *fault* includes the features *source*, *dimension* and *persistence* as defined in Fig. 3a, and likewise these are represented as keywords. Besides the different features, each column also includes the keyword *description*, which is used to explain the domain specific details of the failure scenarios. Typically these descriptions are obtained from domain experts. The columns *fault* and *failure* include the fields *source* and *target* respectively. Both of these refer to failure scenarios of other architectural elements. For example, in failure scenario F_2, the fault source is defined as CMR (F_5), indicating that the fault in F_2 occurs due to a failure in CMR as defined in F_5. The source of the fault can be caused by a combination of failures. This is expressed by logical connectives. For example, the source of F_1 is defined as CH (F_4) OR CMR (F_6) indicating that F_1 occurs due to a failure in CH as defined in F_4 or due to a failure in CMR as defined in F_6.

4.3. Derive fault tree set from failure scenarios

A close analysis of the failure scenarios shows that they are connected to each other. For example, the failure scenario F_1 is caused by the failure scenario F_4 or F_6. To make all these connections explicit, in SARAH *fault trees* are defined. A fault tree is a model for representing the cause–effect relations of failures and faults. The root of a fault tree represents a failure and the leaf nodes represent faults. Since a failure can be logically caused by a set of faults, the nodes of the fault tree are interconnected with logical connectives. Typically, a given set of failure scenarios leads to a set of fault trees, which are together defined as a *fault tree set* (FTS). Formally, FTS is a graph $G(V, E)$ consisting of the set of fault trees. G has the following properties:

Table 2
Selected concrete failure scenarios derived from general scenarios for analysis of DTV architecture

FID	AEID	Fault	Error	Failure
F_1	AMR	*Description*: Reception of irrelevant signals *Source*: CH (F_4) OR CMR (F_6) *Dimension*: Software *Persistence*: Transient	*Description*: Working mode is changed when it is not desired *Type*: Wrong path *Detectability*: Undetectable *Reversibility*: Reversible	*Description*: Switching to an undesired mode *Type*: Behavior *Target*: User
F_2	AMR	*Description*: Cannot acquire information *Source*: CMR (F_5) *Dimension*: Software *Persistence*: Transient	*Description*: Information can not be acquired from the connected device *Type*: Too early/late *Detectability*: Detectable *Reversibility*: Irreversible	*Description*: Cannot provide information *Type*: Timing *Target*: CB (F_3)
F_3	CB	*Description*: Cannot acquire information *Source*: AMR (F_2) *Dimension*: Software *Persistence*: Transient	*Description*: Information can not be presented due to lack of information *Type*: Too early/late *Detectability*: Detectable *Reversibility*: Irreversible	*Description*: Cannot present content of the connected device *Type*: Behavior *Target*: User
F_4	CH	*Description*: Software fault *Source*: Internal *Dimension*: Software *Persistence*: Permanent	*Description*: Signals are interpreted in a wrong way *Type*: Wrong value *Detectability*: Undetectable *Reversibility*: Reversible	*Description*: Provide irrelevant information *Type*: Wrong value/presentation *Target*: AMR (F_1)
F_5	CMR	*Description*: Protocol mismatch *Source*: External *Dimension*: Software *Persistence*: Permanent	*Description*: Communication cannot be sustained with the connected device *Type*: Too early/late *Detectability*: Detectable *Reversibility*: Irreversible	*Description*: Cannot provide information *Type*: Timing *Target*: AMR (F_2)
F_6	CMR	*Description*: Software fault *Source*: Internal *Dimension*: Software *Persistence*: Permanent	*Description:* Signals are interpreted in a wrong way *Type*: Wrong value *Detectability*: Undetectable *Reversibility*: Reversible	*Description*: Provide irrelevant information *Type*: Wrong value/presentation *Target*: AMR (F_1)
F_7	DDI	*Description*: Reception of out-of-spec signals *Source*: External *Dimension*: Software *Persistence*: Transient	*Description*: Scaling information can not be interpreted from meta-data *Type*: Wrong value *Detectability*: Detectable *Reversibility*: Reversible	*Description*: Cannot provide data *Type*: Wrong value/presentation *Target*: VP (F_8)
F_8	VP	*Description*: Inaccurate scaling ratio information *Source*: DDI (F_7) and VC (F_9) *Dimension*: Software *Persistence*: Transient	*Description*: Video image cannot be scaled appropriately *Type*: Wrong value *Detectability*: Undetectable *Reversibility*: Reversible	*Description*: Provide distorted video image *type*: Presentation quality *Target*: User
F_9	VC	*Description*: Software fault *Source*: Internal *Dimension*: Software *Persistence*: Permanent	*Description*: Correct scaling ratio cannot be calculated from the video signal *Type*: Wrong value *Detectability*: Detectable *Reversibility*: Reversible	*Description*: Provide inaccurate information *Type*: Wrong value/presentation *Target*: VP (F_8)

1. $V = F \cup A$
2. F is the set of failure scenarios each of which is associated with an architectural element.
3. $F_u \subseteq F$ is the set of failure scenarios comprising failures that are perceived by the user (i.e. system failures). Vertices residing in this set constitute root nodes of fault trees.
4. A is the set of gates representing logical connectives.
5. $\forall g \in A$,
   - [a] *outdegree* $(g) = 1 \wedge$
   - [b] *indegree* $(g) \geqslant 1$
6. $A = A_{AND} \cup A_{OR}$ such that,
   - [a] A_{AND} is the set of AND gates.
   - [b] A_{OR} is the set of OR gates.
7. E is the set of directed edges (u, v) where $u, v \in V$.

For the example case, based on the failure scenarios in Table 2 we can derive the FTS as depicted in Fig. 4.

Here, the FTS consists of three fault trees. On the left the fault tree shows that F_1 is caused by F_4 or F_6. The middle column indicates that failure scenario F_3 is caused by F_2 which is on its turn caused by F_5. Finally, in the last column the fault tree shows that F_8 is caused by both F_7 and F_9.

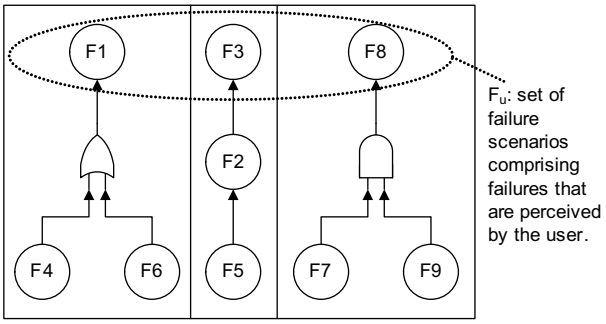


Fig. 4. Fault trees derived from failure scenarios in Table 2.

4.4. Define severity values in fault tree set

Usually, in FTA probability values are assigned to the leaf nodes of a fault tree based on the frequency of fault occurrences (Dugan, 1996). The probability of the root failure is then computed based on these probability values of the lower level nodes. The fault tree is thus processed in a bottom-up manner starting from the leaf nodes.

As we have indicated before we are focused on failure analysis in the context of consumer electronics domain. As such we need to analyze in particular those failures which have the largest impact on the user perception. For example, a complete black screen will have a larger impact on the user than a temporary distortion in the image. These different user perceptions on the failures have a clear impact on the way how we process the fault trees. Before computing the failure probabilities of the individual leaf nodes, we first assign *severity* values to the root failures based on their impact on the user. In our case the severity values are based on the prioritization of the failure scenarios as defined in Table 3.

The severity degrees range from 1 to 5 and are provided by domain experts. For example, the severity values for failures F_1, F_3 and F_8 are depicted in Fig. 5. Here we can see that F_5 has been assigned the severity value 5 indicating that it has a very high impact on the user perception. In fact, the impact of a failure can differ from user to user. In this paper, we consider the average user type. To define user perceived failure severities, an elaborate user model can be developed based on experiments with subjects from different age groups and education levels. We consider the user model as an external input and any such model can be incorporated to the method.

The severity values of the root failures are used to determine the severity values of the other, lower nodes of the FTS. These values are calculated based on the following equation:

Table 3
Prioritization of severity categories

Severity	Type	Annoyance description
1.	Very low	User hardly perceives failure
2.	Low	A failure is perceived but not really annoying
3.	Moderate	Annoying performance degradation is perceived
4.	High	User perceives serious loss of functions
5.	Very high	Basic user-perceived functions fail. System locks up and does not respond

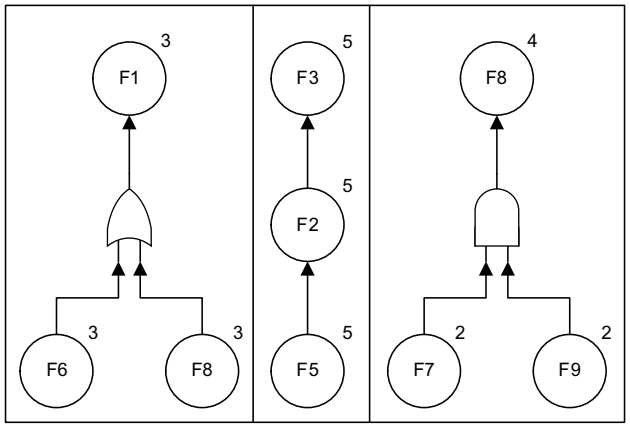


Fig. 5. Fault trees with severity values.

$$\forall u \in F_u, s(u) = s_u$$
$$\forall f \in F \wedge f \notin F_u,$$
$$s(f) = \sum_{\substack{\forall v \\ (f,v)\in E}}^{\text{s.t.}} s(v) \times P(v|f) \quad (1)$$

The first part of the equation defines the assignment of severity values to the root failures (1). The second part indicates the calculation of the severity values for the lower nodes. Here, $P(v|f)$ denotes the probability that the occurrence of f will cause the occurrence of v (Dugan, 1996). We multiply this value with the severity of v to calculate the severity of f. According to the probability theory, $P(v|f) = P(v \cap f)/P(f)$. If v is an OR gate, then the output of v will always occur whenever f occurs. That is, $P(v \cap f) = P(f)$. As a result, $P(v/f) = 1$. If v is an AND gate, $P(v \cap f) = \prod P(x)$ for all vertices x that is connected to v. To calculate $P(v|f)$ we need to know $P(x)$ for all x except f. For example, F_8 has been assigned the severity value 4. Due to the use of AND gates F_7 and F_9 are assigned the value $s(F_8) \times P(F_8|F_7) = 4 \times P(F_9)$ and $s(F_8) \times P(F_8|F_9) = 4 \times P(F_7)$, respectively.

To define the final severity values obviously we need to know the probability of each failure. In principle there are three strategies that can be used to determine these required probability values:

- *Using Fixed values*:

All probability values can be fixed to a certain value. An example is to assume that each failure will have equal weight and likewise the probabilities of individual failures are basically defined by the AND and OR gates.

- *What-if analysis*

A range of probability values can be considered, where fault probabilities are varied and their effect on the probability of user perceived failures are observed.

- *Measurement of actual failure rate*

The actual failure rate can be measured based on historical data or execution probabilities of elements that are obtained from scenario-based test runs (Yakoub et al., 2004).

In SARAH all these three strategies can be used depending on the available knowledge on probabilities. In case the probabilities are not known or they do not have a relevant impact on the final outcome then fixed probability values can be adopted. If probabilities are not equal or have different impacts on the severity values then either the second or third strategy can be used. In the second strategy, the what-if analysis can be useful if no information is available on an existing system. The measurement of actual failure rates is usually the most accurate way to define the severity values. However, the historical data can be missing or not accessible, and deriving execution probabilities is cumbersome.

To identify the appropriate strategy a preliminary *sensitivity analysis* can be performed. In conventional FTA, sensitivity analysis is based on cut-sets in a fault tree and the probability values of fault occurrences (Dugan, 1996). However, this analysis leads to complex formulas and it requires that the probability values are known priori. In our case we propose the following model for estimating the sensitivity with respect to a fault probability even if the probability values are not known

$$\begin{aligned} &root \in F_u, node \in F,\\ &\forall n \in F \quad \wedge n \neq node \wedge n \neq root, P(n) = p,\\ &P(node) = p', P(root) = f(p', p),\\ &sensitivity(node) = \int_0^1 \left(\frac{\partial}{\partial p'} P(root)\right) \mathrm{d}p \end{aligned} \tag{2}$$

The equation above, shows the calculation of the sensitivity of a user perceived failure ($P(root)$) to the probability of a *node* ($P(node)$) of the fault tree. Here, we assign p' to $P(node)$ and fix the probability values of all the other nodes to p. Thus $P(root)$ turns out to be a function of p and p'. For example, if we are interested in the sensitivity of $P(F_8)$ to $P(F_9)$, $P(F_8) = P(F_7) \times P(F_9) = p' \times p$. Then, we take a partial derivative of $P(root)$ with respect to p'. This gives the rate of change of $P(root)$ with respect to p' ($P(F_9)$). For our example, this will yield to $\partial/\partial p' P(F_8) = p$. Finally, the result of the partial derivation is integrated with respect to p for all possible probability values ([0–1]) to calculate the overall impact. For the example case, $\int(\partial/\partial p' P(F_8))\mathrm{d}p = \int p \mathrm{d}p = p^2/2$. So, the result of the integration from 0 to 1 will be 0.5. In fact, in this model we use the basic sensitivity analysis approach, where we change a parameter one at a time and fix the others (Yakoub et al., 2004). Additionally, we use derivation to calculate the rate of change and we use integration to take the range of probability values into account. The resulting formulas are simple enough to be computed with spreadsheets.

For the analysis presented in this paper, we skip the sensitivity analysis and assume equal probabilities (i.e. $P(x) = p$ for all x), which simplifies the severity assignment formula for AND gates as $s(v) \times P(v|f) = s(v) \times p^{\mathrm{INDEGREE}(v)-1}$. Based on this assumption, the severity calculation for intermediate failures is as shown in the following equation:

$$s(f) = \sum_{\substack{\forall v \quad \text{s.t.} \\ (u,v)\in E \wedge \\ (v\in F \vee v \in A_{OR})}} s(v) + \sum_{\substack{\forall v \quad \text{s.t.} \\ (u,v)\in E \wedge \\ v \in A_{AND}}} s(v) \times p^{INDEGREE(v)-1} \tag{3}$$

In our analysis, we fixed the value of p to be 0.5. In case of F_7 and F_9, for instance, the severity value is 4/2 because F_8 has the severity value of 4 and it is connected to an AND gate with $INDEGREE = 2$. On the other hand, F_1 has the assigned severity value of 3 and this is also assigned to the failures F_6 or F_8 that are connected to F_1 through an OR gate. A failure scenario can be connected to multiple gates and other failures, in which the severity value is derived as the sum of the severity values calculated for all these connections.

5. Analysis of software architecture

In our example case, we defined a total of 44 failure scenarios including the scenarios presented in Table 2. We completed the definition process by deriving the corresponding fault tree set and calculating severity values as explained in Section 4. The results that were obtained during the definition process are utilized by the analysis process as described in the subsequent sections.

5.1. Perform architecture-level analysis

The first step of the analysis process is architecture-level analysis in which we pinpoint critical elements of the architecture with respect to reliability. Here, we can consider two types of critical elements 1) *unreliable elements* 2) *sensitive elements*. Unreliable elements are the elements, which are the sources of the majority of the failure scenarios (i.e. root causes). Sensitive elements are the elements, which are associated with the majority of the failure scenarios. These include not only the failure scenarios caused by internal faults but also the failure scenarios caused by other elements. In the architectural-level analysis step, we aim at identifying sensitive elements with respect to the most critical failures from the user perception. In this context sensitivity analysis actually precedes reliability analysis and as such forms a starting point for tackling unreliable elements. Later, in the architectural element level analysis, the fault, error types and the actual sources of their failures (internal or other unreliable elements) are identified. Sensitive elements provide a starting point for considering the relevant fault tolerance techniques and the elements to improve with respect to reliability (see Section 6). In this way the effort that is provided for reliability analysis is scoped with respect to failures that are directly perceivable by the users.

As a primary and straightforward means of comparison, we consider the percentage of failures (PF) that are associated with elements. For each element c the value for PF is calculated as follows:

$$\mathrm{PF}_c = \frac{\#\text{ of failures associated with } c}{\#\text{ of failures}} \times 100 \quad (4)$$

This means that simply all the number of failures related to an element are summed and divided by the total number of failures (in this case 44). The results are shown in Fig. 6. A first analysis of this figure already shows that the *Application Manager (AMR)* and *Teletext (TXT)* modules have to cope with a higher number of failures than other modules.

This analysis treats all failures equally. To take the severity of failures into account we define the Weighted Percentage of Failures (WPF) as given in Eq. (3).

$$WPF_c = \frac{\sum_{\forall u \in F \quad \text{s.t.} \atop \mathrm{AEID}(u)=c} s(u)}{\sum_{\forall u \in F} s(u)} \times 100 \quad (5)$$

For each element, we collect the set of failures associated with them and we add up their severity values. After averaging this value with respect to all failures and calculating the percentage, we obtain the WPF value. The result of the analysis is shown in Fig. 7.

Although weighted percentage presents different results compared to the previous one, the *Application Manager* (AMR) and Teletext (TXT) modules again appears to be very critical.

From the project perspective it is not always possible to focus on the total set of possible failures due to the related cost for reliability. To optimize the cost usually one would like to consider the failures that have the largest impact on the system. For this, in SARAH the architectural elements are ordered in ascending order with respect to their WPF values. The elements are then categorized based on the proximity of their WPF values. Accordingly, elements of the same group have WPF values that are close to each other. The results of this prioritization and grouping are provided in Table 4, which also shows the sum of the WPF values of elements for each group. Here we can see that, for example, group 4 consists of two modules AMR and TXT. The reason for this is that their WPF values are the highest and close to each other. The sum of their WPF values is $25 + 14 = 39\%$.

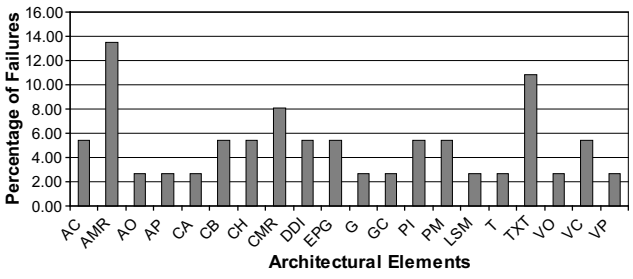


Fig. 6. Percentage of failure scenarios impacting architectural elements.

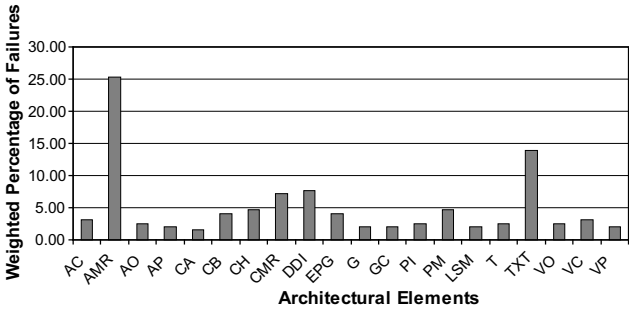


Fig. 7. Weighted percentage of failures impacting architectural elements.

Table 4
Architectural elements grouped based on WPF values.

Group #	Modules	WPF (%)
1	AC, AO, AP, CA, G, GC, PI, LSM, T, VO, VC, VP	28
2	CB, CH, EPG, PM	18
3	DDI, CMR	15
4	AMR, TXT	39

To highlight the difference in impact of the groups of architectural elements we define a Pareto chart as presented in Fig. 8.

In the Pareto chart, the groups of architectural elements shown in Table 4 are ordered along the x-axis with respect to the number of elements they include. The percentage of elements that each group includes is depicted with bars. The y-axis on the left hand shows the percentage values from 0 to 100 and is used for scaling the percentages of architectural elements whereas the y-axis on the right hand side scales WPF values. The plotted line represents the WPF value for each group. In the figure we can, for example, see that group 4 (consisting of two elements) represents 10% of the all elements but has a WPF of 39%. The chart helps us in this way to focus on the most important set of elements which are associated with the majority of the user perceived failures.

5.2. Perform architectural element level analysis

The architectural level analysis provides only a quantitative analysis of the impact of failure scenarios on the given.

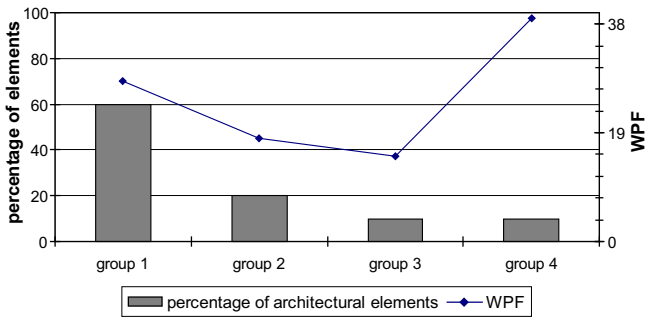


Fig. 8. Pareto chart showing the largest impact of the smallest set of architectural elements.

However, for failure management and recovery it is also necessary to define the *type* of failures that might occur in the identified sensitive elements. This is analyzed in the architectural element level analysis in which the features of faults, errors and failures that impact an element are determined. For the example case, in the architectural-level analysis it appeared that elements residing in the 4th group (see Table 4) had to deal with largest set of failure scenarios. Therefore, in architectural element level analysis, we will focus on members of this group, namely *Application Manager* and *Teletext* modules.

Following the derivation of the set of failure scenarios impacting an element, we group them in accordance with the features presented in Fig. 3. This grouping results in the distribution of fault, error and failure categories of failure scenarios associated with the element.

For example, the results obtained for Application Manager and Teletext modules are shown in Fig. 9a and Fig. 9b, respectively. If we take a look at fault features presented on those figures for instance, we see that most of the faults impacting *Application Manager* Module are caused by other modules. On the other hand, *Teletext* Module has internal faults as much as faults stemming from the other modules. As such, distribution of features reveals characteristics of faults, errors and failures associated with individual elements of the architecture. This information is later utilized for architectural adjustment (See Section 6).

5.3. Provide failure analysis report

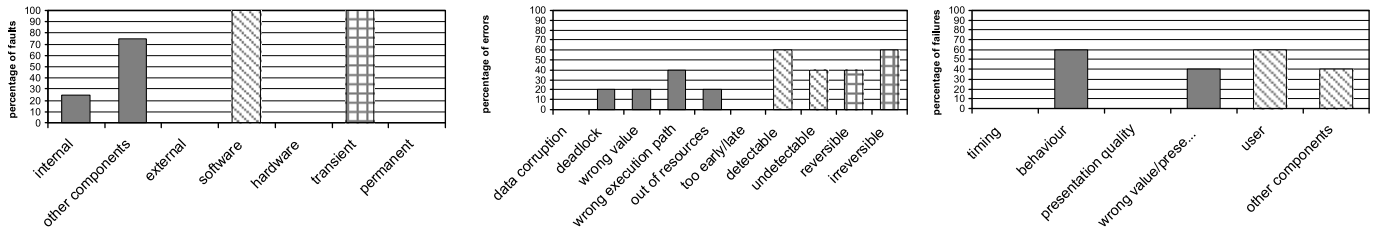
SARAH defines a detailed description of the fault tree sets, the failure scenarios, the architectural level analysis and the architectural element level analysis. These are described in the *failure analysis report* that summarizes the previous analysis results and provides hints for recovery. Sections comprised by the failure analysis report are listed in Table 5, which are in accordance with the steps of SARAH. The first section describes the project context, information sources and specific considerations (e.g. cost-effectiveness). The second section describes the software architecture. The third section presents the domain model of faults, errors and failures which include features of interest to the project. The fourth section contains list of failure scenarios annotated based on this domain model. The fifth section depicts the fault tree set generated from the failure scenarios together with the severity values assigned to each. The sixth and seventh sections include analysis results as presented in Sections 5.1 and 5.2 of this paper, respectively. Additionally, the sixth section includes the distribution of fault, error and failure features for all failure scenarios as depicted in Fig. 10. Finally, the report includes a section on first hints for architectural recovery as titled *architectural tactics* (Bachman et al., 2003). This is explained in the next section.

Table 5
Sections of the failure analysis report

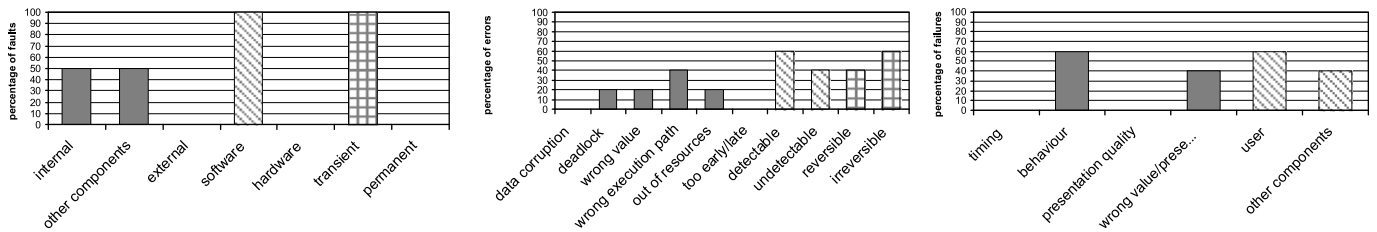
1. Introduction
2. Software architecture
3. Failure domain model
4. Failure scenarios
5. Fault tree set
6. Architecture level analysis
7. Architectural element level analysis
8. Architectural tactics

6. Architectural adjustment

The failure analysis report that is defined during the reliability analysis forms an important input to the architec-



(a) Fault, error and failure features associated with Application Manager Module



(b) Fault, error and failure features associated with Teletext Module

Fig. 9. Fault, error and failure features of failure scenarios associated with elements in 4th group of the Pareto chart.

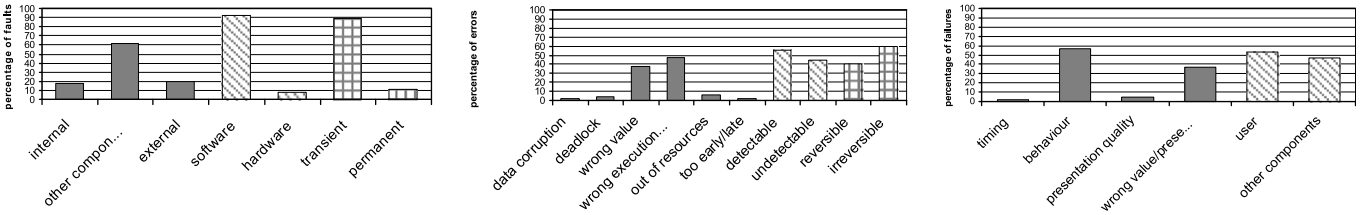


Fig. 10. Feature distribution of fault, error and failures for all failure scenarios.

tural adjustment. Hereby the architecture will be enhanced to cope with the identified failures. This requires the following three steps: (1) defining the elements to which the failure relates (2) identifying the architectural tactics and (3) application of the architectural tactics.

6.1. Define architectural element spots

Architectural tactics have been introduced as a characterization of architectural decisions that are used to support a desired quality attribute (Bachman et al., 2003). In SARAH we apply the concept of architectural tactics to derive architectural design decisions for supporting reliability. The previous steps in SARAH result in a prioritization of the most sensitive elements in the architecture together with the corresponding failures that might occur. Thus, SARAH prioritizes actually the design fragments (Bachman et al., 2003) to which the specific tactics will be applied and to improve reliability. In general, for applying a recovery technique to an element we need also to consider the other elements coupled with it. This is because local treatments of individual elements might directly impact the dependent elements. For this reason, we define *architectural element spot* for a given element as the set of elements with which it interacts. This draws the boundaries of the design fragment (Bachman et al., 2003) to which the design decisions will be applied. The coupling can be statically defined by analyzing the relations among the elements. Table 6 shows architectural element spots for each element in the example case. For example, Table 6 shows the elements that are in the architectural element spot of AMR as AC, AO, CB, CH, CMR, DDI, EPG, GC, LSM, PI, PM, TXT, VC and VO. These elements should be considered while incorporating mechanisms to AMR or any refactoring action that includes alteration of AMR and its interactions.

Table 6
Architectural element spots

AEID	Architectural element spot
AMR	AC, AO, CB, CH, CMR, DDI, EPG, GC, LSM, PI, PM, TXT, VC, VO
AC	AMR, AP, LSM
AP	AC, DDI
AO	AMR, CA
CA	AMR, PM, T, AO, VO
CB	AMR, CMR
CH	AMR
CMR	AMR, CB
DDI	AMR, AP, CA, EPG, TXT, VP
EPG	AMR, DDI
G	GC
GC	AMR, G
LSM	AMR, AC, PM, VC
PI	AMR, PM
PM	AMR, CA, LSMR, T
T	CA, PM
TXT	AMR, DDI
VC	AMR, LSM, VP
VO	AMR, CA
VP	DDI, VC

6.2. Identify architectural tactics

Once we have identified the sensitive elements, the element spots and the potential set of failure scenarios, we proceed with identifying the architectural tactics for reliability. The number of reliability techniques in the literature is quite broad. In Avizienis et al. (2001), the means of achieving reliability is categorized as *fault forecasting*, *fault removal*, *fault prevention* and *fault tolerance* (See Fig. 11). The analysis report of SARAH defines the potential set of failure scenarios for the analyzed system, which can be considered as fault forecasting. Fault forecasting supports the other means of reliability since it points out potential faults that are needed to be tolerated, prevented or removed. In SARAH, we particularly focus on fault tolerance techniques.

After identifying the sensitive elements we analyze the fault and error features related to these sensitive elements (Fig. 10). Faults can be either internal or external to the sensitive element. If the source of the fault is internal this means that the sensitive element itself is unreliable and as such the fault tolerance techniques will be applied to the sensitive element itself. However, if the source of the fault is external, this means that the failures are caused by the other elements that are unreliable. In that case, we might need to consider applying fault tolerance techniques to these unreliable elements instead. These unreliable elements can be traced with the help of the architectural element spot (Section 6.1) and fault tree set. In case faults are *transient* we might also apply fault tolerance techniques to the sensitive elements although they are not the causes of the failures. Examples of these fault tolerance techniques are exception handling, restarting, check-pointing and roll-back recovery (Huang and Kintala, 1995). If the faults

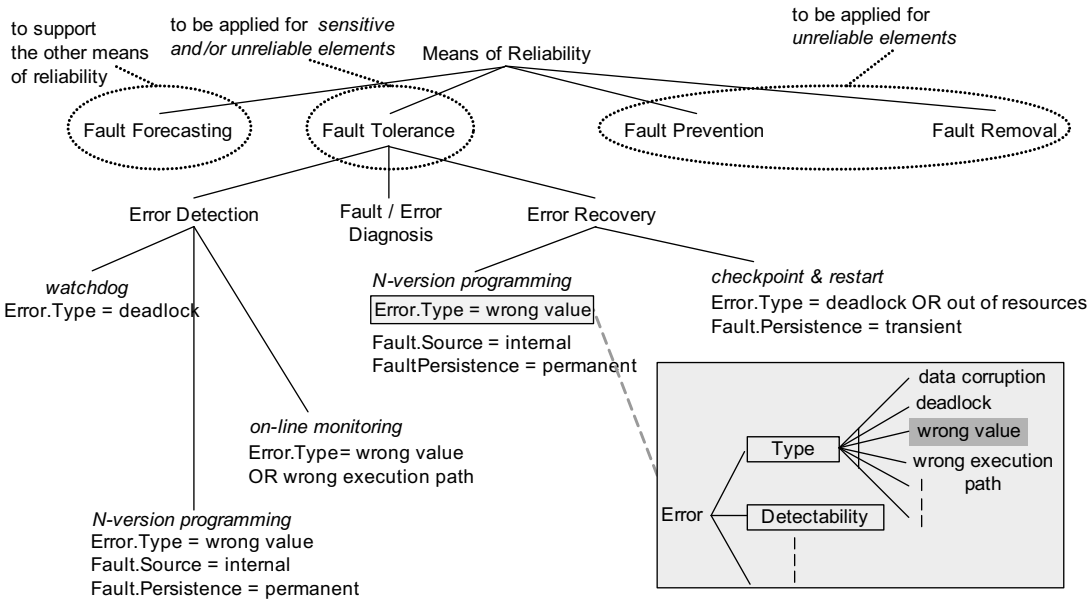


Fig. 11. Partial categorization of means of reliability with corresponding fault and error features.

are *internal* and *permanent*, then one should apply design diversity (e.g. recovery blocks, N-version programming (McAllister and Vouk, 1996) on the sensitive element to tolerate such faults.

In Fig. 11, some examples for fault tolerance techniques have been given. Here, fault tolerance techniques are further divided into detection, diagnosis and recovery steps. Each technique provides a solution for a particular set of problems. To make this explicit, each fault tolerance technique is tagged with the fault and error types that it aims to detect and/or recover. As an example, in Fig. 11, *watchdog* (Huang and Kintala, 1995) is applied to detect *deadlock* errors for tolerating faults resulting in such errors. On the other hand, *N-version programming* (McAllister and Vouk, 1996) is used to both detect and recover from *wrong value* errors that are caused by *internal* and *persistent* faults.

Fig. 11 provides only a part of the reliability techniques as an example. Derivation of all fault tolerance tactics is out-of-scope of this paper.

6.3. Apply architectural tactics

As the last step of architectural adjustment, selected architectural tactics should be applied to adjust the architecture if necessary.

The possible set of architectural tactics is determined as described in the previous section. Additionally, other criteria need to be considered including cost and limitations imposed by the system (resource constraints). For the given case, Table 7 shows, for example, the potential means of reliability that can be applied for AMR and TXT modules. The last column of the table shows the selected candidate techniques.

Although SARAH identifies tactics that could be applied to enhance reliability it does not explicitly state how to realize the selected tactics. Obviously, the realization of an architectural tactic and measuring its properties (reengineering and performance overhead, availability, performability) for a system requires dedicated analysis and additional research. To identify the appropriate techniques we adopt the existing literature on the selected tactic. For example, in Laprie et al. (1995) fault tolerance techniques that are based on diversity (e.g. recovery blocks, N-version programming) are evaluated with respect to dependability and cost. In Candea et al. (2004) availability improvement achieved by component-level restarts is evaluated in the Internet services domain.

Table 7
Treatment approaches for sensitive components

AEID	Potential means	Selected means
AMR	On-line monitoring, watchdog, checkpoint & restart, resource checks, interface checks	On-line monitoring, interface checks
TXT	On-line monitoring, watchdog, n-version programming, checkpoint & restart, resource checks, interface checks	On-line monitoring, checkpoint & restart, resource checks

In our case, on-line monitoring was selected as a candidate error detection technique to be applied to the AMR and TXT modules. Similar to other tactics it appears that there are different ways to realize on-line monitoring as it is described in literature. Schroeder, for example, provides a classification of on-line monitoring and explains its characteristics (Schroeder, 1995). Based on this classification scheme we have defined one solution for on-line monitoring. Hereby, the design is enhanced with on-line monitoring facilities including sensing based on sampling, embedded event interpretation, and explicit recovery action specifications (Schroeder, 1995).

7. Discussion

The method and its application provide new insight in the scenario-based architecture analysis methods. A number of lessons can be learned from this study.

- *Early reliability analysis*

Similar to other software architecture analysis methods based on our experiences with SARAH we can state that early analysis of quality at the software architecture level has a practical and important benefit (Dobrica and Niemela, 2002). In our case the early analysis relates to the analysis of the next release of a system (e.g. Digital TV) in a product line. Although, we did had access to the next release implementation of the Digital TV the reliability analysis with SARAH still provided useful insight in the critical failures and elements of the current architecture. For example, we were able to identify the critical modules *Application Manager, Teletext* and also got an insight in the failures, their causes and their effects. Without such an analysis it would be hard to denote the critical modules that have the risk to fail. For the next releases of the product this information can be directly utilized in deciding for the reliability techniques to be used in the architecture.

- *Utilizing a quality model for deriving scenarios*

The use of an explicit domain model for failures has clearly several benefits. Actually, in the initial stages of the project we first tried to directly collect failure scenarios by interviewing several stakeholders. In our experience this has clear limitations because (1) the set of failure scenarios for a given architecture is in theory too large and (2) even for the experts in the project it was hard to provide failure scenarios. To provide a more systematic and stable approach we have done a thorough analysis on failures and defined a fault domain model that represents essentially the reliability quality attribute. This model does not only provide systematic means for deriving failure scenarios but also defined the stopping criteria for defining failure scenarios. Basically we have looked at all the elements of the architecture, checked the failure domain and expressed the failure scenarios using the failure scenario template that we have presented in Section 4.2. During the whole process we were supported by TV domain experts.

- *Impact of project requirements and constraints*

From the industrial project perspective it was not sufficient to just define a failure domain model and derive the scenarios from it. A key requirement of the industrial case was to provide a reliability analysis that takes the user-perception as the primary criteria. This requirement had a direct impact on the way how we proceeded with the reliability analysis. In principle, it meant that all the failures that could be directly or indirectly perceived by the end-user had to be prioritized before the other failures. In our analysis this was realized by weighing the failures based on their severities from the user-perspective. In fact, from a broader sense, the focus on user-perceived failures could just be considered an example. SARAH provides a framework for reliability analysis and the method could be easily adapted to highlight other types of properties such as for example hardware/software failures.

- *Calculation of probability values of failures*

One of the key issues in the reliability analysis is the definition of the probability values of the individual failures. In Section 4.4 we have described three well-known strategies that can be adopted to define the probability values. In case more knowledge on probabilities is known in the project the analysis will be more accurate accordingly. As described in Section 4.4 SARAH does not adopt a particular strategy and can be used with any of these strategies. We have illustrated the approach using fixed values.

- *Inherent dependence on domain knowledge*

Obviously, the set of selected failure scenarios and values assigned to their attributes (severity) directly affect the results of the analysis. As it is the case with other scenario-based analysis methods both the failure scenario elicitation and the prioritization are dependent on subjective evaluations of the domain experts. To handle this inherent problem in a satisfactory manner SARAH guides the scenario elicitation by using the relevant failure domain model as described in Section 4.2.1 and the use of failure scenario template in Section 4.2. The initial assignment of severity values for the user-perceived failures is defined by the domain experts, but the independent calculation of the severities for intermediate failures is defined by the method itself as defined in Eq. (1) and (2) in Section 4.4.

- *Extending the fault tree concept*

In the reliability engineering community, fault trees have been used for a long time in order to calculate the probability that a system fails (Dugan, 1996). This information is derived from the probabilities of fault occurrences by processing the tree in a bottom-up manner. The system or system state can be represented by a single fault tree where the root node of the tree represents the failure of the system. In this context, failure means a crash-down in which no more functionality can be provided further. The fault tree can be also processed in a top-down manner to find the root causes of this failure. One of the contributions of this paper from the reliability engineering perspective is the fault tree set (FTS) concept, which is basically a set of fault trees. The difference of FTS from a single fault tree is that an intermediate failure can participate in multiple fault trees and there exists multiple root nodes each of which represents a system failure perceived by the user. This refinement

enables us to discriminate different types of system failures (e.g. based on severity) and infer what kind of system failures that an intermediate failure can lead to.

- *Architectural Tactics for reliability*

After the identification of the sensitive architectural elements, the related element spots and the corresponding failures in SARAH architectural tactics are defined. In the current literature on software architecture design this is not explicit and basically we had to rely on the techniques that are defined by the reliability engineering community. We have modeled a selected set of fault tolerance and fault prevention techniques and defined the relation of these techniques with the faults and errors in the failure domain model that had been defined before. In Bachman et al. (2003) derive architectural tactics by first listing concrete scenarios, matching these with general scenarios and deriving quality attribute frameworks. In a sense this could be considered as a bottom-up approach. In SARAH the approach is more top-down because it essentially starts at the domain models of failures and recovery techniques and then derives the appropriate matching. The concept of architectural tactics as an appropriate architectural design decision to enhance a particular quality attribute of the architecture remains of course the same.

8. Related work

Balsamo et al. (2004) provide a nice survey on model-based performance prediction methods. They classify the various approaches with respect to the phase in which the analysis is performed, the mapping between the software model and performance model (syntactic, semantic, same model) and the level of automation (low, medium, high). According to their classification scheme, our approach can be categorized as an early analysis approach that is applied at the software architecture design phase. We make use of multiple models which are semantically related. Failures associated with architectural components are expressed with a failure scenario model, from which the fault trees are directly derived. The automation in our case is provided by separate spreadsheets that define the failure scenarios and automatically calculate the severity values in the fault trees (after initial assignment of the user-perceived severities). This is a straightforward calculation and as such we have not elaborated on this issue.

Several software architecture reliability evaluation methods that employ quantitative models have been proposed (Goseva–Popstojanova et al., 2001). Most of these methods make use of variations of Markov models (DTMC, CTMC) in order to model the software architecture (Goseva–Popstojanova and Trived, 2001). These models are utilized to estimate the reliability of the system based on the reliability of its components. To achieve the same goal, the other approaches utilize scenarios (Zarras and Issarny, 2000), optimization algorithms (Roshandel and Medvidovic, 2004), dependency graphs and execution traces (Yakoub et al., 2004). In our method, we aim at pinpointing the sensitive elements of a software architecture with respect to reliability.

Application of FMEA to software has a long history (Reifer, 1979). Both FMEA and FTA have taken place in analysis of software systems and named as Software Failure Modes and Effects Analysis (SFMEA) and Software Fault Tree Analysis (SFTA), respectively. In SFMEA, failure modes for software components are identified such as *computational*, *logic* and *data I/O*. This classification resembles the failure domain model of SARAH. However, SARAH separates fault, error and failure concepts and provides a more detailed categorization for each. Also, note that the failure domain model can vary depending on the project requirements and the system. In general, efforts for applying reliability analysis to software mainly focus on the safety-critical systems, whose failure may have very serious consequences such as loss of human life and large-scale environmental damage. In our case, we focus on consumer electronics domain, where the systems are usually not safety-critical. For this reason, instead of safety, we take user-perception as the criteria to assign severity to failures.

9. Conclusion

Given the trends of increased software functionality, complexity and openness in embedded systems, it is expected that the risk of failures in embedded systems can increase to a mission critical level. To keep the current reliability levels, appropriate reliability analysis and design techniques are necessary so that potential failures can be predicted or corrected in time. Moreover, because implementing the software architecture is a costly process it is important to predict the quality of the system as early as possible before committing enormous organizational resources.

To meet these two goals we have introduced the software architecture reliability analysis method (SARAH) that has been developed within an industrial project on reliability analysis of software architectures for embedded systems (Trader project web site, 2005). SARAH actually integrates the best practices of conventional reliability (Dugan, 1996) with current scenario-based architectural analysis methods (Dobrica and Niemela, 2002). Conventional reliability engineering includes mature failure analysis and detection techniques. Software architecture analysis methods provide useful techniques for early analysis of the system at the architecture design level. The definition of fault, error failure models, the failure scenarios, fault tree set and the severity calculations are inspired from the reliability engineering domain (Avizienis et al., 2001, 1996). The overall scenario-based elicitation and prioritization is inspired from the work on software architecture analysis methods (Dobrica and Niemela, 2002). Despite most scenario-based analysis methods which usually do not focus on specific quality factors, SARAH is a specific purpose

analysis method focusing on the reliability quality attribute. Further, unlike conventional reliability analysis techniques which tend to focus on safety requirements SARAH prioritizes and analyses failures basically from the user perception due to the requirements of consumer electronics domain. To provide a reliability analysis based on user perception we have extended the notion of fault trees and refined the fault tree analysis approach. One of the critical issues in the analysis is the definition of the probability values of failures. However, just like existing reliability analysis approaches in the literature the accuracy of the analysis depends on the available knowledge in the project. SARAH does not adopt a fixed strategy but can be rather considered as a general process in which different strategies for defining probability values can be used.

SARAH has helped us to identify the sensitive modules for the Digital TV and provided an important input for the enhancement of the architecture. Besides of the outcome the process of doing such an explicit analysis has provided better insight in the potential risks of the system and clearly supported the enhancement of the architecture.

Our future work includes the definition of a tool which supports the reliability analysis process. In addition it is planned to apply the approach for the reliability analysis for different systems than DTV.

Acknowledgements

We thank the anonymous reviewers for their earlier valuable feedback to improve this work. We thank members of the TRADER project, for their feedback on this work and their input about the TV domain knowledge and reliability issues. Particularly, we would like to thank Rob Golsteijn from NXP Semiconductors, Paul L. Janson from Philips Research, Iulian Nitescu from Philips TASS for their contribution in deriving the conceptual architecture of DTV and possible failures. We also specially thank Christian Hofmann from University of Twente, Pierre van de Laar, Teun Hendriks and Jozef Hooman from ESI for reviewing and providing useful feedback to this paper.

References

Arrango, G., 1994. Domain Analysis Methods. In: Schäfer, Prieto-Díaz, R., Matsumoto, M. (Eds.), Software Reusability. Ellis Horwood, New York, pp. 17–49.

Avizienis, A., Laprie, J.-C., Randell, B., 2001. Fundamental concepts of dependability, LAAS Report No. 01145, LAAS-CNRS, France, April.

Bachman, F., Bass, L., Klein, M., 2003. Deriving architectural tactics: a step toward methodical architectural design, CMU/SEI-2003-TR-004, ADA413644, Pittsburgh, PA, March.

Balsamo, S., Di Marco, A., Inverardi, M., Simeoni, P., 2004. Model-based performance prediction in software development. IEEE Transactions on Software Engineering 30 (5), 295–310.

Candea, G., Cutler, J., Armando Fox, 2004. Improving availability with recursive microreboots: a soft-state system case study. Performance Evaluation 56 (1–4), 213–248.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures. Addison-Wesley.

Clements, P., Kazman, R., Klein, M., 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

Dobrica, L., Niemela, E., 2002. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28 (7), 638–654.

Dugan, J.B., 1996. Handbook of software reliability engineering. In: Lyu, M.R. (Ed.), Software System Analysis Using Fault Trees. McGraw-Hill, New York, pp. 615–659, Chapter 15.

Goseva–Popstojanova, K., Mathur, A.P., Trived, K.S., 2001. Comparison of Architecture-Based Software Reliability Models. In Proc. of the 12th International Symposium on Software Reliability Engineering, pp. 22–31.

Goseva–Popstojanova, K., Trived, K.S., 2001. Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45 (2–3), 179–204.

Huang, Y., Kintala, C., 1995. Software fault tolerance. In: Lyu, M.R. (Ed.), Software Fault Tolerance in the Application Layer. John Wiley & Sons Inc, New York, pp. 231–248, Chapter 10.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.S., 1990. Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, November.

Laprie, J.C., Arlat, J., Beounes, C., Kanoun, K., 1995. Software fault tolerance. In: Lyu, M.R. (Ed.), Architectural Issues in Software Fault Tolerance. John Wiley & Sons, Inc, New York, pp. 47–80, Chapter 3.

McAllister, D.F., Vouk, M.A., 1996. Handbook of software reliability engineering. In: Lyu, M.R. (Ed.), Fault-Tolerant Software Reliability Engineering. McGraw-Hill, New York, pp. 567–613, Chapter 14.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture Description Languages. IEEE Transactions on Software Engineering 26 (1), 170–193.

Reifer, D.J., 1979. Software failure modes and effects analysis. IEEE Transactions on Rel-28 3, 247–249.

Roshandel, R., Medvidovic, N., 2004. Toward architecture-based reliability estimation. In Proc. of the International Conference on Dependable Systems and Networks (DSN). Florence, Italy.

Schroeder, B.A., 1995. On-line monitoring: A tutorial. IEEE Computer, 72–78.

Trader project web site, http://www.esi.nl/site/projects/trader, accessed October 5, 2005.

Yakoub, S., Cukic, B., Ammar, H., 2004. A scenario-based reliability analysis approach for component-based software. IEEE Transactions on Reliability 53 (4), 465–480.

Zarras, A., Issarny, V., 2000. Assessing software reliability at the architectural level. In Proc. of the 4th International Software Architecture Workshop, June, Limerick, Ireland.

Bedir Tekinerdogan, holds an M.Sc. degree and a Ph.D. degree in Computer Science from the University of Twente. Currently, he is assistant professor at the Department of Computer Science in the Software Engineering group at the University of Twente where he is teaching courses on software architecture design and serving as daily supervisor for research projects on this topic. His basic research interests include software architecture design, software reliability analysis, model-driven software development, aspect-oriented software development and software product line engineering. He is also serving as a chairman of the steering committee of the biennial Turkish Software Architecture Design Conferences (UYMK).

Hasan Sozer is a Ph.D. student at the University of Twente, The Netherlands. He received the BS and MS degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. From August 2002 until January 2005, he worked as a software engineer at ASELSAN Inc. in Turkey. His research interests include software architecture design, software fault tolerance and wireless ad hoc networks.

Mehmet Aksit holds an M.Sc. degree from the Eindhoven University of Technology and a Ph.D.degree from the University of Twente. Currently, he is working as a full professor at the Department of Computer Science, University of Twente. He is the head of the Software Engineering chair and the leader of the Twente Research and Education on Software Engineering (TRESE) Group. His research interests include aspect-oriented software development, synthesis based software design, application of fuzzy logic to software design processes, and design algebra for managing large design spaces.