Software Architecture Reliability Analysis using Failure Scenarios

Bedir Tekinerdogan, Hasan Sözer, Mehmet Aksit
Department of Computer Science, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands
{bedir|sozerh|aksit}@cs.utwente.nl

Abstract

We propose an approach for analyzing software architectures with respect to reliability to improve fault tolerance. The approach defines a failure scenario model that is based on the established failure modes and effects analysis method (FMEA) in the reliability engineering domain. Failure scenarios are systematically derived from a fault domain model and expressed using a failure scenario model. The developed failure scenarios are utilized to derive so-called fault tree sets (FTS) that are prioritized based on severity from the user perspective. Severity analysis is provided for the top-level architecture and the most relevant fault categories are identified for the individual components. The method results in an impact analysis report that can be utilized for improving reliability of the software architecture with respect to user-perceived failures. The ideas are illustrated using an industrial case for analyzing reliability of the software architecture for a Digital TV.

Keywords
reliability analysis, scenario based architectural evaluation, FMEA, fault trees.

1. Introduction

Two important trends can be observed in the development of embedded systems. First, the demand for products with more and more functionality is increased due to the high industrial competition and the advances in hardware and software technology. Second, the amount and complexity of software in embedded systems is continuously increasing. These two trends complicate the design and implementation of embedded systems and as such put serious challenges on assuring quality. Since embedded systems are largely defined by software, the required quality is likewise basically dependent on the quality of the software.

A key quality factor in embedded systems is obviously reliability. Together with the increased size and complexity of software it is expected that the risk of failures will rise to a critical level for the software business if appropriate reliability analysis techniques are not applied. In this context, it should be noted that research on reliability analysis has primarily addressed failures in hardware components while the available research on software reliability analysis has basically focused on running systems requiring information regarding the behavioral properties of its components (i.e. average execution time, reliability) and the operational profile.

It is now currently recognized that reliability analysis should not be limited to hardware components but primarily needs to analyze software reliability. Further, reliability analysis can not just be performed at the code level but early prediction at the software architecture design is considered necessary. This is required to identify the potential risks of failures, and the verification whether the reliability requirements have been appropriately addressed. Likewise, numerous researchers have addressed the importance of analyzing software architectures in order to predict the quality of a system before it has been built. For analyzing software architectures usually either static analysis of formal architectural models [23] or a set of scenario-based architecture analysis methods as described in [10] are applied. In this paper, we focus on software architecture analysis methods that utilize scenarios for evaluating architectures. In general, these analysis methods take as input the architecture design and measure the impact of 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 [1]. Hereby, it is implicitly assumed that scenarios correspond to the particular quality attributes that need to be analyzed. Numerous scenario-based architecture analysis methods have been developed each focusing on a particular quality attribute or attributes. For example, SAAM has focused on analyzing modifiability of the architecture, SAAMCS [19] on *complexity*, SAAMER [20] on *reusability* and *evolution*, ALPSM [9] on *maintenance,* ESAAMI [21] on *reuse* from existing component libraries, and *ASAAM* [26] on identifying aspects for increasing *maintenance*. Some methods such as SAEM [13] and ATAM [17] have considered the need for a specific quality model for deriving the corresponding scenarios. ATAM has also discussed the problem of dealing with multiple quality attributes and the trade-off and optimization problems that emerge from these.

We think that for providing reliability analysis at the software architecture it is beneficial to utilize both results from research on conventional reliability analysis techniques and scenario-based software architecture analysis techniques. As a result we propose Software Architecture Reliability Analysis approach (SARA) that is a scenario-based approach integrating conventional reliability analysis techniques. SARA defines a failure scenario model that is based on the established failure modes and effects analysis method (FMEA) in the reliability engineering domain. Failure scenarios are systematically derived from a fault domain model and expressed using a failure scenario model. The developed failure scenarios are utilized to derive so-called fault tree sets (FTS) that are prioritized based on severity from the user perspective. Severity analysis is provided for the top-level architecture and the most relevant fault categories are identified for the individual components. The method results in an impact analysis report that can be utilized for improving reliability of the software architecture with respect to user-perceived failures. The ideas are illustrated using an industrial case for analyzing reliability of the software architecture for a Digital TV.

The remainder of this paper is organized as follows. In section 2, we present the industrial case for analyzing reliability of a Digital TV Architecture. In section 3, we provide a general domain model for faults that has been derived from the reliability analysis domain. The fault domain will be used as an input for deriving failure scenarios. Section 4 explains the steps of SARA and applies this to the reliability analysis of the software architecture for Digital TV. Section 5 discusses the lessons learned. Section 6 provides the related work and finally section 7 concludes the paper.

2. Industrial Case: Digital TV Software (DTV) Architecture

At the Embedded Systems Institute (ESI) in The Netherlands, the TRADER (Television Related Architecture and Design to Enhance Reliability) project is carried out with Philips Consumer

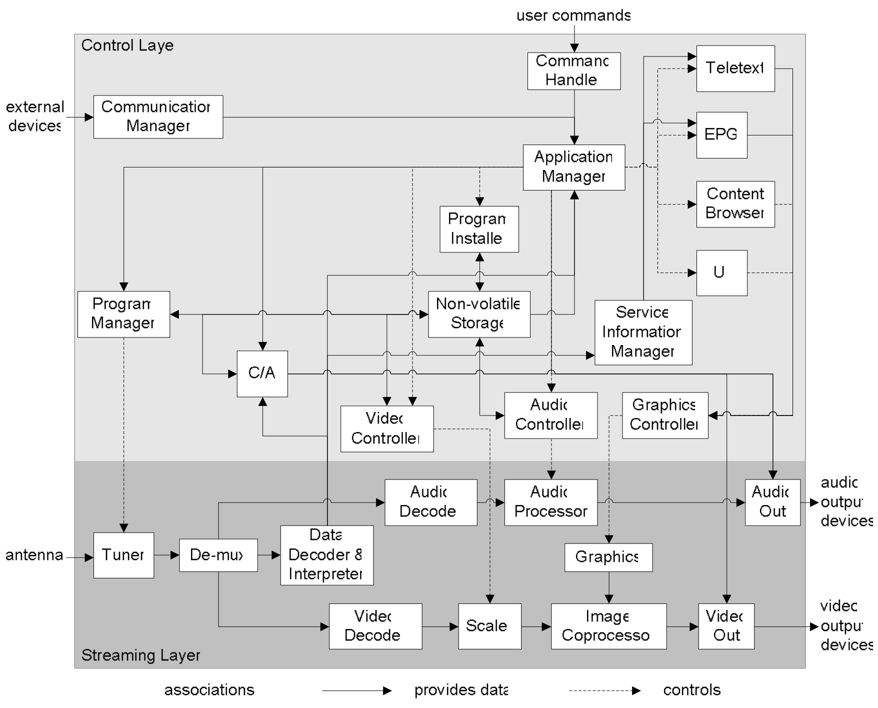


Figure 1. *(Simplified) top-level architecture of digital TV*

Electronics, Philips Semiconductors and several other academic and industrial partners [27]. The objective of the project is to develop methods and tools for ensuring reliability of consumer electronic products, while digital television (DTV) is chosen to be the industrial case to focus on. As in the case with other consumer electronic products one can observe two important trends in TV systems. Firstly, on the one hand the TV gets a broader number of features for each release. Secondly, the amount of software that is embedded in the TV is dramatically increasing leading to an exponential increase in software complexity. Historically the reliability of the TV was basically related to correctness and failures were basically detected and corrected at the run-time component level. However, it is now recognized that the reliability of the TV system can not be faced by using reliability techniques at the code level only, since potential failures very often result from early design decisions at the software architecture design. Therefore, one of the key aims in TRADER is the design of fault tolerant software architecture with respect to user-perspective.

A simplified conceptual architecture of DTV is depicted in Figure 1[1], which mainly comprises two layers. The bottom layer, namely the streaming layer, involves components taking part in streaming of audio/video information and embodies pipe-and-filter style [10]. The upper layer consists of applications, utilities and components that control streaming by interacting with corresponding components in the streaming layer. Besides of the components for decoding and processing audio, video and data some of the other important components are described below:

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

*Audio Controller (AC), c*ontrols audio features like volume level, bas and treble based on commands received from *Application Manager.*

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

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

[1] Due to space limitation and confidentiality, we can not make the DTV architecture and full set of failure scenarios available. Instead, we present a simplified architecture and representative set of scenarios in order to illustrate the application of the method.

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

EPG (Electronic Program Guide) presents and provides navigation of electronic program guide regarding a channel that is obtained from *Service Information Manager.*

Program Installer (PI) searches and registers programs to *Non-volatile Storage* together with channel information (i.e. frequency).

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

Service Information Manager (SIMR) registry of services and volatile information provided by channels including teletext and program guide.

UI (User Interface) Manager used by applications running on the system for generating and showing Graphical User Interface (GUI) elements. Controls *Graphics Controller* component for generation of graphical images.

3. Fault Domain

In order to understand reliability we need to understand failures and failure analysis. A *failure* is generally defined as an event that occurs when the delivered service of a system deviates from a correct service [5]. A correct service is delivered when the service implements the system function. A service failure can occur because it does not comply with the functional specification, or because the specification did not adequately describe the required system function. The cause of a failure is called a *fault*.

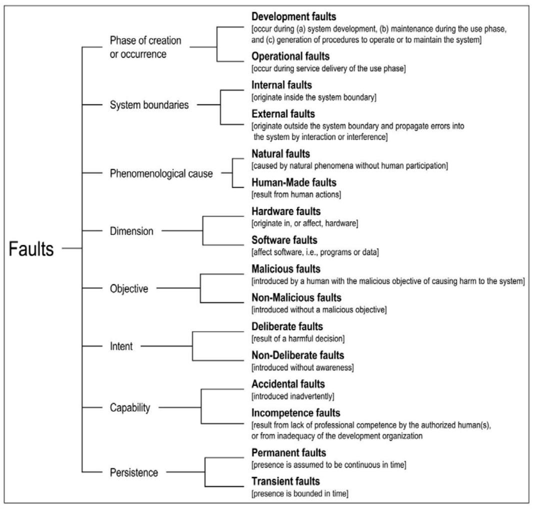

Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human-Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non-Malicious faults
[introduced without a malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non-Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Figure 2. *Domain model for faults [5]*

Aviezienis et al [5] have classified all the faults that may affect a system according to the eight viewpoints as defined in Figure 2 from which 31 likely combinations of faults are possible. The fault classes are grouped as (a) *development faults* that include all fault classes occurring during development; (b) *physical faults* that include fault classes that affect hardware and (3) *interaction faults* that include all external faults. Knowledge of all these fault classes is necessary to evaluate the architecture with respect to reliability.

The analysis of the faults can be used for [5]:

1. *fault prevention*, for preventing the occurrence or introduction of faults.
2. *fault tolerance*, for avoiding failures in the presence of faults
3. *fault removal*, for reducing the number and severity of the faults
4. *fault forecasting*, for estimating the current number and the likely occurrences of faults.

4. Architectural Reliability Analysis

The steps for SARA are presented in Figure 3. The rectangles represent the steps; the arrows represent the control flow. The steps are explained in the following subsections.

Error! Objects cannot be created from editing field codes.

Figure 3. *Software Architecture Reliability Analysis*

4.1 Describe Candidate Architecture

Similar to existing software architecture analysis methods SARA starts with defining potential alternative architectures. The candidate architecture includes the architectural components and their relationships. The method does not presume to provide a particular architectural view but in essence can be applied independently on the architectural view that has to be dealt with.

4.2 Define Relevant Fault Domain Model

As it was defined in section 3 the domain of faults is rather broad and one may easily assume that for a given reliability analysis project not all the potential faults are relevant or necessary. Therefore, a clear focus on the faults that need to be avoided or detected is required. This is defined by requirements engineers who define the key issues for reliability. The result of this is typically a feature diagram of a fault domain as presented in Figure 4.

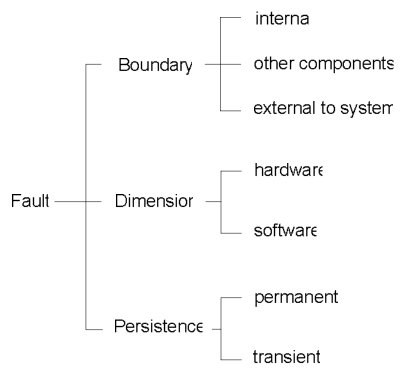


Figure 4. *Derived fault domain model for the fault tolerant architecture*

Note that the feature diagram of 4 is a refined subset of the feature diagram as presented in Figure 2, and provides the particular faults that are relevant for the corresponding project. The features represent alternative features. In particular it should be noted that the feature *Boundary* has now three alternative features since from the project perspective it is required to separate the faults that are internal to the component, originating from other components and external to the system.

4.3 Develop Failure Scenarios

An important step in scenario-based architecture analysis methods is the scenario elicitation process. In general, scenarios are developed based on expert knowledge on the corresponding system and should reflect the stakeholders' need and the required quality attributes. As in the case of scenarios for other quality attributes, analyzing software architecture with respect to reliability imposes specific constraints on the scenario elicitation. In this context an important observation is the fact that the set of faults that could occur is in theory nearly infinite. Even if we are dealing with a finite set of architectural components and relations, we could imagine many kinds of faults for each component and component relations. On the other hand, for an appropriate analysis it is still required that the set of scenarios that is elicited is sound and balanced from the stakeholders' perspectives. To provide a feasible scenario analysis for reliability we first reduce the set of possible faults by looking at the domain model for faults. The first reduction is already in the general intrinsic fault domain as provided in Figure 2. Each fault can only appear in one of the 31 fault categories. In section 4.2 we have seen that we could even further reduce this using the relevant fault domain that includes only the faults that are relevant for the project. In the given example, we could derive only 3 types of faults (*Boundary, Dimension, Persistence)* leading to 7 different faults.

Table 1.Template for defining a failure scenario

Failure scenario template	
failure id	a numerical value to identify the failures
component id	an acronym defining the component for which the failure scenario applies
failure mode	the manner how the component fails; based on the relevant fault domain model and the interactions of the component
failure cause	the cause of the failure mode defining both the description of the cause and their corresponding components
failure effect	the effect of the failure mode defining both the description of the cause and the target(s). The target could be the user or another component

We now define a *failure scenario* as a failure that is caused by a fault in the relevant fault domain, affecting an architectural component or the user. Hereby, fault scenarios are represented as instantiations of the fault model as provided in Table 1**Error! Reference source not found.**. The fault model itself is based on the concepts of the well-established Failure Modes and Effects Analysis (FMEA), which is a reliability analysis method for eliciting and evaluating potential risks. In FMEA, a *fault* is defined as the cause of a failure. A *failure mode* is defined as the manner in which the element fails. A *failure effect* is the (undesirable) consequence of a failure mode. A failure has an effect on a *target*. *Failure cause* can be the component itself (e.g. software fault) or external with respect to the system (e.g. reception of out-of-spec signals). Another possibility is that failure of a component can be a fault (failure cause) for another one, which leads to a failure of the target component. Target of failure can also be the user, which means that user perceives a deviation from the intended behavior of the system.

In SARA failure scenarios are thus derived using the candidate architecture and its components, relevant fault domain model, the stakeholders and the failure scenario template. For example, using the candidate architecture of the DTV in Figure 1 and the selected fault domain model in Figure 4, we can derive failure scenarios using the template in Table 1. A selected set of the results are the failure scenarios as defined in Table 2.

4.4 Define Fault Trees

After the failure scenarios are derived we define the *fault trees*. A fault tree is a model for representing the cause-effect relations of failures and faults. The root of a failure 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 tree are interconnected with logic gates characterizing the propagation behavior of faults. Every set of failure scenarios will lead to a set of fault trees. We term this as a Fault Tree Set (FTS). Fault Tree Set (FTS) is a set of fault trees forming a graph G (V, E) in which a vertex can be a part of more than one fault tree. G has the following properties.

1. $V = C \cup F \cup G$
2. C is the set of *faults*, which constitutes the leaf nodes of fault trees.
3. F is the set of *failures* each of which is associated with an architectural component.
4. $F_s \subseteq F$ is the set of *failures* targeting user (i.e. system failures), which constitutes the root nodes of fault trees.
5. G is the set of *gates*.
6. $\forall$ g $\in$ G,
    a. *outdegree(g)* = 1 $\wedge$
    b. *indegree(g)* $\geq$ 1
7. $G = G_{AND} \cup G_{OR}$ such that,
    a. G_{AND} is the set of AND *gates*
    b. G_{OR} is the set of OR *gates*
8. E is the set of directed edges (*u*, *v*) such that,
    a. $(u \in C \wedge v \in G) \vee$
    b. $(u \in C \wedge v \in F) \vee$
    c. $(u \in F \wedge v \in G) \vee$
    d. $(u \in F \wedge v \in F) \vee$
    e. $(u \in G \wedge v \in G)$

For the failure scenarios of Table 2 we can derive 5 fault trees as depicted in Figure 5.

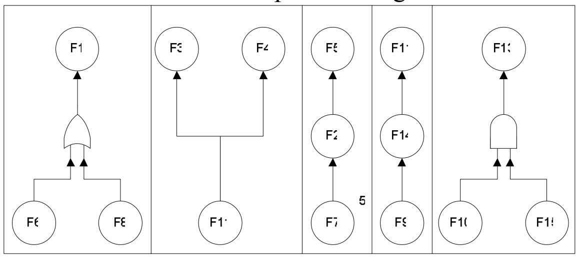


Figure 5

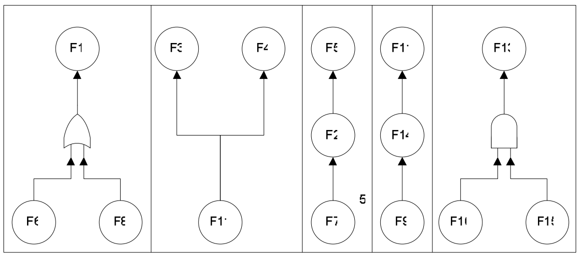


Figure 5 For example, on the left column the fault tree for F1 for *Application Manager* (AMR) has been given. By searching for the target AMR in other failure scenarios we can easily derive the causes of failure F1. It appears that failures F6 and F8 have a target AMR and as such cause F1. This can also be derived by inspecting the *failure cause* field of F1.

Table 2.*Failure Scenarios derived for DTV architecture*

FID	CID	Failure Mode	Failure Cause	Failure Effect
F1	AMR	Working mode is changed when it is not desired.	Reception of irrelevant signals <source > CH.F6 or CM.F8	Switching to an undesired mode <target> User
F2	AMR	Information can not be acquired from the connected device.	Can not acquire information <source> CM.F7	Can not provide information <target> CB.F5
F3	AC	Last volume level can not be restored.	Can not acquire last state information < source > NVS.F12	Adjustment of default volume level instead of last volume level <target> User
F4	CA	Audio/video data can not be authorized.	Can not acquire authorization information <source> NVS.F12	Provide unauthorized access to audio/video data <target> User
F5	CB	Information can not be presented due to lack of information	Can not acquire information <source> AMR.F2	Can not present content of the connected device <target> User
F6	CH	Signals are interpreted in a wrong way.	Software fault <source> internal	Provide irrelevant information <target> AMR.F1
F7	CM	Communication can not be sustained with the connected device.	Protocol mismatch <source> external	Can not provide information <target> AMR.F2
F8	CM	Signals are interpreted in a wrong way.	Software fault <source> internal	Provide irrelevant information
F9	DDI	Data can not be interpreted.	Reception of out-of-spec signals <source> external	Provide wrong/incomplete information <target> CA; AMR; SIMR.F14
F10	DDI	Scaling information can not be interpreted from meta-data.	Reception of out-of-spec signals <source> external	Can not provide data <target> S.F13
F11	EPG	Information can not be presented due to lack of information.	Can not acquire information <source> SIMR.F14	Can not present Electronic Program Guide <target> User
F12	NVS	Data can not be provided due to a corruption.	Hardware fault <source> internal	Can not provide data <target> PI; CA.F4; PMR; AC.F3; VC; AMR
F13	S	Video image can not be scaled appropriately.	Inaccurate scaling ratio information <source> DDI.F10 and VC.F15	Provide distorted video image <target> User
F14	SIMR	Information can not be provided due to lack of input data.	Can not acquire data <source> DDI.F9	Can not provide data <target> TXT; EPG.F11
F15	VC	Correct scaling ratio can not be calculated from the video signal.	Software fault <source> internal	Provide inaccurate information <target> S.F13

The next step is to decide the logical connection. For this, we utilize the domain knowledge and decide to apply an OR connection. This means that scenario F6 which denotes a wrong interpretation of signals by Command Handler (CH), and F8 denoting the wrong interpretation of signals by Command Manager (CM) can both independently lead to failure F1. We repeat the process until all the fault trees are derived as depicted in Figure 5**.**

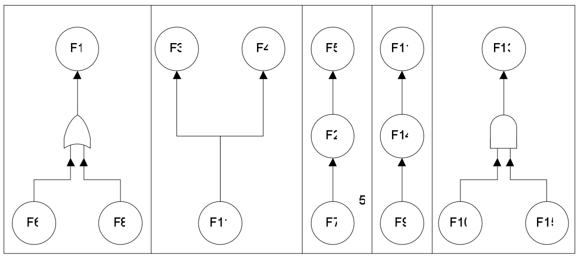


Figure 5. *Fault trees derived from failure scenarios of table 1*

One failure can lead to multiple failures as it is in the case of F11 (second column). Scenario F13 can only occur in case F10 and F15 occur together. Again, the selection of a logical connection is decided based on the domain knowledge, in this case the knowledge on the failures themselves.

4.5 Define Severities for Fault Trees

Once the fault tree set is defined we define the severity degrees. In conventional fault tree analysis, fault trees are processed in a bottom-up manner in order to calculate the probability that a failure would take place, based on the probabilities of fault occurrences which are assumed to be given [12]. The *severity* is defined as a concept related to faults denoting how severe a fault is (e.g. faulty component can be repaired or not). In our model, we take a user-centric approach and define *severity* based on the user-perception. System failures that we consider include are not restricted to complete crash-down of the system. For instance, a minor distortion in the brightness level of the video image and absence of any image are both failures. However, they would not upset the user in the same way. As a diversion from the usual approach, we process the fault trees in a top-down manner to assign severity values to intermediate failures based on severities of system failures. The more an intermediate failure takes place in a fault tree and contributes to the occurrence of system failures, the more would be its severity value. Also, it depends on severities of system failures that it is leading to.

Based on the FTS formalism, calculation of severity values of failures can be defined as follows:

$$\forall u \in F_S,\ s(u) = s_u \quad s.t. \quad 1 \le s_u \le 5$$
$$\forall u \in F \wedge u \notin F_S, \qquad (1)$$
$$s(u) = \sum_{\substack{\forall v \ \ s.t. \\ (u,v)\in E \,\wedge \\ (v \in F \,\vee\, v\in G_{OR})}} s(v) + \sum_{\substack{\forall v \ \ s.t. \\ (u,v)\in E \,\wedge \\ v\in G_{AND}}} s(v) \Big/ in\deg ree(v)$$

In the first part of the equation, a severity value is assigned for every system failure based on the level of severity from the user-perception. For this we first take into account the failure scenarios with target *user*. These are the failures F1, F3, F4, F5, F11 and F13. We apply a prioritization of the failure scenarios as defined in Table 3. The severity degrees range from 1 to 5 and are derived from the domain experts.

Table 3. *Prioritization of severity categories*

Severity	Description	Annoyance Description
1.	very low	user hardly perceives failure
2.	low	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; no perception possible

The severity values for failures F1, F3, F4, F5, F11 and F13 are depicted in Figure 6. These values are then used in order to determine the severity values of other failures as shown in the second part of the equation. If a failure directly leads to another one or if it is connected through an OR gate in which some other failures may exclusively cause the same failure also, we add the severity of resulting failure to the severity of failure under consideration. If there is a connection through an AND gate where a combination of failures results in another one, we share the severity value of the resulting failure among contributing failures. In other words, we add the severity of resulting failure divided by the number of contributing failures to severity of each contributing failure. For example, because F1 has the assigned severity value of 3, this is also assigned to the failures F6 or F8 that are connected through an OR logic to F1. In case of F13, F10 and F15 get the severity value of 4/2 because F13 has the severity value of 4 and it has an AND gate with 2 failures connected to it. The severity value of F11 is derived as the sum of the severity values of F3 and F4.

Error! Objects cannot be created from editing field codes.

(1)

Figure 6. *Fault trees with severity values*

4.6 Architecture-Level Analysis

After calculating severity values of failures as explained in the previous sections, we are able to perform architecture-level analysis in which we can pinpoint sensitive points of the architecture with respect to reliability. In the first step of the analysis, we take all components into consideration and analyze the percentage of failures (PF) that are associated with them. We basically calculate the following for each component *c*.

$$PF_c = \frac{\#of\ failures\, associated\ with\, c}{\#of\ failures} \times 100 \qquad (2)$$

The results for our case are shown in Figure 7. A first analysis already shows that the first and twentieth components appear to be associated with failures much more than other components. These two components are *Application Manager* and *Teletext*, respectively.

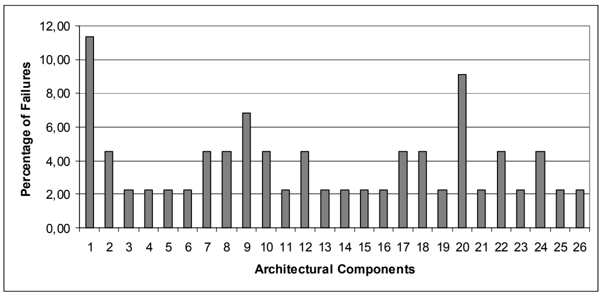


Figure 7. *Percentage of failure scenarios impacting components*

This analysis treats all failures equally. If we want to take the severity of failures into account, calculation of weighted percentages is essential. For this we define the Weighted Percentage of Failures (WPF) as given in equation (3).

Error! Objects cannot be created from editing field codes. (3)

For each component, 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.

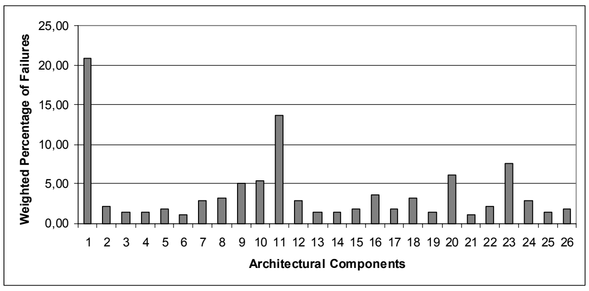


Figure 8. *Weighted percentage of failures impacting components.*

The result of the analysis is shown in Figure 8. Although weighted percentage presents different results compared to the previous one, the first component standing for *Application Manager* again appears to be very critical.

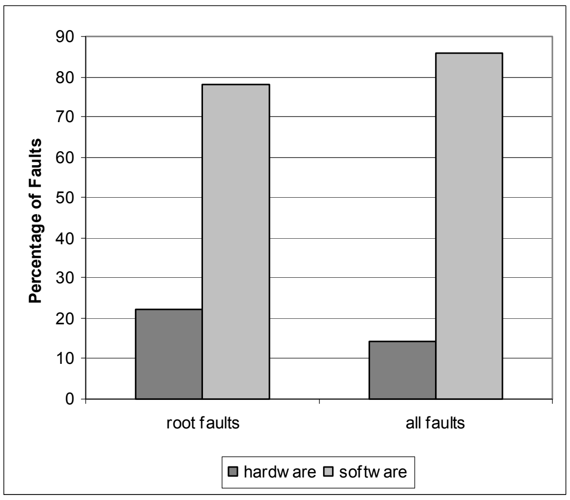
4.7 Component-Level Analysis

In this step, we analyze the categories of faults that impact a component. We focus on component *Application Manager*, which is distinguished during the sensitivity analysis and for this purpose the set of faults impacting it must be collected. At this point, two set of faults are distinguished; (1) *root faults* which comprise only leaf nodes of fault trees impacting the component, (2) *all faults* which comprise both leaf and intermediate nodes impacting the component. Formally, the set of root faults (RF) and the set of all faults (AF) impacting a component *c* can be defined as follows.

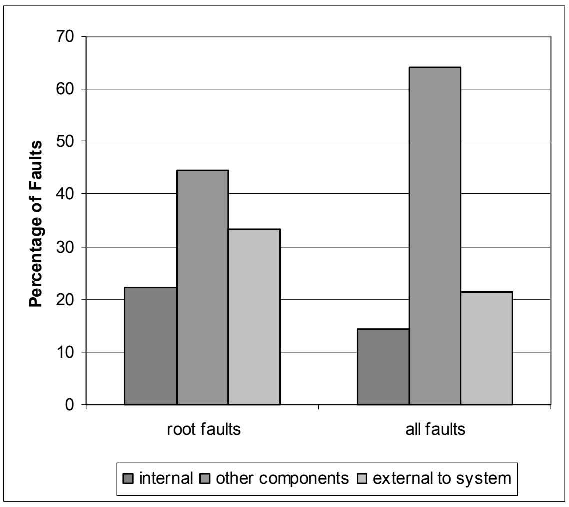
$$\begin{aligned} RF_c &= \forall u \in C \ s.t. \exists\, path(u,v) \wedge copm(v) = c \\ AF_c &= \forall u \in F \ s.t. \exists\, path(u,v) \wedge copm(v) = c \end{aligned} \quad (4)$$

The set of *root faults* are all vertices *u* ∈ C, where there exists a path from *u* to *v* in F and *v* is associated with the component under consideration. Similarly, the set of *all faults* is defined as all vertices *u* ∈ F, where there exists a path from u to v in F and v is associated with the component under consideration.

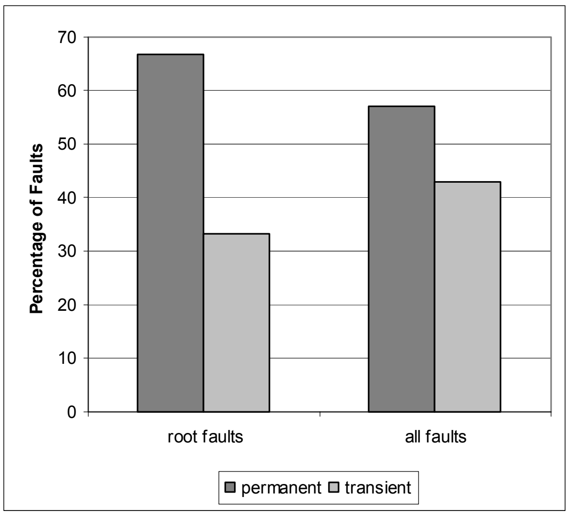
Following the derivation of the set of faults impacting the component, we group them. In accordance with the fault domain model (Figure 4), grouping is carried out based on three attributes; dimension, boundary and persistence. At the end, we end up with distribution of fault categories impacting the component as shown in Figure 9.



(a) Fault Categories Based on Dimension



(b) **Fault Categories Based on Boundary**



(c) **Fault Categories Based on Persistence**

Figure 9. *Distribution of Fault categories impacting Application Manager component*

In Figure 9a, we can see that the majority of faults having impact on the component are software faults rather than hardware faults. When not only root faults but also intermediate component failures are considered, dominance of software faults increases as expected. The chart depicted in Figure 9b shows that most of these faults are originated externally; either from other components or from sources that are external to the system. This notices us the necessity to pay attention on inputs received from external interfaces of the component. Consideration of intermediate component failures in this distribution inevitably increases the proportion of failures originated from other components. In the last chart shown in Figure 9c, we observe that %70 of the root faults is permanent. When all faults are taken into account, the percentage of permanent faults decreases but it still remains significant.

4.8 Define Impact Analysis Report

In section 3 we have described that the reliability analysis can be utilized for supporting *fault prevention*, *fault tolerance*, *fault removal*, or for *fault forecasting*. SARA defines a detailed description of the failure sets, the failure scenarios, the architectural level analysis and the component level analysis. These are described in the impact analysis report that summarizes the previous analysis results and provides hints for improvements. SARA analyzes the architecture and provides crucial information on the critical risks and as such provides support for enhancing the architecture. Using the impact analysis report the appropriate architecture design and reliability techniques can be used to improve the architecture. For example, by means of examining both specific faults and categories of faults impacting critical components, particular fault-tolerant techniques can be chosen among different alternatives [18]. Solutions based on introducing redundancy, for instance, would be more appropriate for tolerating permanent and internal faults (e.g. introducing a redundant storage in case of data corruptions).

5. 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 prediction for reliability is useful***

First of all, like other software architecture analysis methods have also confirmed we could state that early prediction of quality at the software architecture has a real and important benefit [11]. Although, we did neither assume or had access to the next release implementation of the Digital TV the reliability analysis with SARA still provided useful insight in the critical failures and components of the current architecture. For example, we were able to identify the critical components *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 components 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 in the architecture.

- ***Quality model for deriving scenarios is necessary***

In the initial stages of the project we tried to collect failure scenarios in a bottom up approach by interviewing stakeholders. In our experience this had a limited range because on the one hand the set of failure scenarios for a given architecture are in theory to large and secondly even for the experts in the project it is hard to provide failure scenarios. This directly shows the need for defining quality models for which the scenarios need to be derived. In this context, we have benefited from the huge domain of reliability engineering, did a thorough analysis on failures and defined a fault domain model. This fault domain does not only provide systematic means for deriving failure scenarios but also defined the stopping criteria. Basically we looked at all the components of the architecture, checked the fault domain and expressed the failure scenarios using the failure scenario template that we have presented in section 4.3.

- ***Specific project requirements must be considered for a feasible scenario elicitation and prioritization***

From the industrial project perspective it was not sufficient to just define a fault 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 severity of the failures based on their severities for the user. In fact, from a broader sense, the focus on user-perceived failures could just be considered an example. SARA provides a framework for reliability analysis and the method could be easily adapted to consider other types of properties.

- ***Quantification of scenarios***

In the quantification of failure scenarios we have not considered the occurrence rate of each failure. A failure that rarely takes place can have very severe effects. On the other hand, a relatively less severe failure might occur too often that it would lead to more user-frustration. As introduced by FMEA for instance, frequency values can be integrated to the model and metrics. For this we will require the possible frequencies from the domain experts. While the general steps of the analysis method would remain the same, in the future, we would like to refine the individual steps by integrating such information to our model.

- ***Inherent dependence on domain knowledge***

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 we guide the scenario elicitation by using the relevant fault domain as described in section 4.2 and the use of failure scenario template in section 4.3. The initial assignment of severity values for the user-perceived failures is defined by the domain experts, but the calculation of the failures is defined by the method itself as defined in the equation (1) in section 4.5.

6. Related Work

Balsamo et al [6] 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 underlying performance model (syntactic, semantic, both) and the level of automation (low, medium, high). Our approach could be categorized as an early analysis approach that is based on fault tree sets, which could be categorized as trace analysis in [6]. The automation in our case is provided by the spreadsheets that define the failure scenarios and automatically calculate the severity values of the fault trees (after initial assignment of the user-perceived severities).

Several methods software architecture reliability evaluation methods that employ quantitative models are proposed. As listed in [15], most of these methods make use of variations of Markov models (DTMC, CTMC) in order to model the software architecture. They basically introduce Markov chains in which, nodes represent states of the running system or components of the software architecture taking part in the execution flow. It is assumed that a running system is available. Reliabilities of components/states together with probabilities of transitions in-between are observed from such a system and they are supplied to the reliability model. Comparison of these models can be found in [14]. In [24], reliabilities of components and transition probabilities between them are calculated by means of a training data and an optimization algorithm. In this technique, the training data has a substantial effect on results. There exists the risk of getting trapped at a local optimum point. A scenario-based reliability evaluation method is proposed in [29]. In this method, reliability of the system is predicted out of reliabilities of its individual components, which are assumed to be known. In [28], another scenario-based method is introduced. Actually, it is based on a quantitative model, so called Component Dependency Graph (CDG). This method assumes that an implementation of the software exists. A set of scenarios run several times and observations like average execution time of components are collected. These are used to determine the parameters of the model, thereafter.

In general, quantitative approaches aim to predict the reliability of a software architecture based on reliability of its components. Such approaches need knowledge about the execution profile and reliability of individual components. In this work, without presence of such information, we aimed to analyze the impact of faults on components of the architecture and investigate sensitive points based on a set of failure scenarios that is derived from the domain knowledge. We do not presume the availability of an implemented system so, the methodology can be applied in early stages of software life cycle and provide insight for the following development phases.

7. Conclusion

In the previous sections we have presented SARA, software architecture reliability analysis method that has been developed within an industrial project on reliability analysis of software architectures for embedded systems [27]. We have applied the method for the analysis of the software architecture of the Digital TV. SARA is developed after a study to both software architecture analysis methods and reliability engineering techniques. The overall scenario-based elicitation and prioritization is inspired from the work on software architecture analysis methods [11]. The definition of a fault model, the failure scenarios, fault tree sets and the severity calculations have been inspired from the reliability engineering domain [12]. The use of a fault model has provided a direct benefit for deriving scenarios in a meaningful and systematic approach. In fact in our opinion it is necessary to define the quality attribute models to provide a meaningful and feasible scenario-based analysis. SARA will be extensively used in the Trader project to guide the software architecture design. In our future work we will consider other quality attributes such as cost and reuse, which are required from a different perspective.

Acknowledgements

We thank all members of the TRADER project, for their input about the TV domain knowledge and reliability issues. Particularly, we would like to thank Rob Golsteijn from Philips Semiconductors, Paul L. Janson and Pierre van de Laar from Philips Research, Iulian Nitescu from Philips TASS for their contribution in deriving the abstract architecture of DTV and possible failures. This work has been carried out as a 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.

References

[1] G. Abowd. *Analyzing Development Qualities at the Architecture Level: The Software Architecture Analysis* Method. in: L. Bass, P. Clements, and R. Kazman (eds.). *Software Architecture in Practice*, Addison-Wesley 1998.

[2] E. Adams. Optimizing preventive service of software products, IBM Research Journal, 28(1), 2-14, 1984.

[3] M. Aksit (Ed.), *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer Academic Publishers, 2001.

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

[5] A. Avizienis, J.-C. Laprie, B. Randell, "Fundamental Concepts of Dependability", LAAS Report No 01145, LAAS-CNRS, France, April 2001.

[6] S. Balsamo, A. Di Marco, P. Inverardi & M. Simeoni. *Model-Based Performance Prediction in Software Development*, IEEE Transactions on Software Engineering, Vol. 30, No. 5, pp. 295-310, May 2004.

[7] V.R. Basili & B.T Perricone. Software Errors and Complexity: An Empirical Investigation', Communications of the ACM 27(1), pp. 42-52, 1984.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, second edition, Addison-Wesley 1998.

[9] P.O. Bengtsson and J. Bosch, ªArchitecture Level Prediction of Software Maintenance,º Proc. Third European Conf. Software Maintenance and Reeng., pp. 139-147, Mar. 1999.

[10] F. Bushman, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. Pattern-Oriented Software Architecture: A System of Patterns. West Sussex, England: John Wiley & Sons Inc., 1996.

[11] L.Dobrica & E.Niemela. A survey on software architecture analysis methods. IEEE Trans. on Software Engineering, Vol. 28, No. 7, pp.638-654, July 2002.

[12] J. B. Dugan. Handbook of Software Reliability

Engineering, M. R. Lyu, Editor, chapter 15, Software System Analysis Using Fault Trees, pages 615-659. McGraw-Hill, New York, NY, 1996.

[13] J.C. Duenas, W.L. de Oliveira, and J.A. de la Puente, A Software Architecture Evaluation Model,º Proc. Second Int'l ESPRIT ARES Workshop, pp. 148-157, Feb. 1998.

[14] K. Goseva–Popstojanova, A. P. Mathur, and K. S. Trived, “Comparison of Architecture-Based Software Reliability Models”, 12th International Symposium on Software Reliability Engineering, pp. 22-31, 2001.

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

[16] R.Kazman, G.Abowd, L.Bass & P.Clements. *Scenario-Based Analysis of Software Architecture.* IEEE Software, pp. 47-55, Nov. 1996.

[17] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The Architecture Tradeoff Analysis Method", Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), (Monterey, CA), pp. 68-78, August 1998.

[18] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun. Software Fault Tolerance, M. R. Lyu, Editor, chapter 3, Architectural Issues in Software Fault Tolerance, pages 47-80. John Wiley & Sons, Inc., New York, NY, 1995

[19] N. Lassing, D. Rijsenbrij, and H. van Vliet, On Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isn't Everything, Proc. Second Nordic Software Architecture Workshop (NOSA '99), pp. 1103-1581, 1999.

[20] C. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, ªAn Approach to Software Architecture Analysis for Evolution and Reusability, Proc. CASCON '97, Nov. 1997.

[21] G. Molter, ªIntegrating SAAM in Domain-Centric and Reuse-Based Development Processes, Proc. Second Nordic Workshop Software Architecture (NOSA '99), pp. 1103-1581, 1999.

[22] D. F. McAllister and M. A. Vouk. Handbook of Software Reliability Engineering, M. R. Lyu, Editor, chapter 14, Fault-Tolerant Software Reliability Engineering, pages 567-613. McGraw-Hill, New York, NY, 1996

[23] N.Medvidovic & R.N. Taylor. *A classification and comparison framework for Software Architecture Description Languages,* IEEE Trans. on Software Engineering, Vol. 26, No.1 pp. 70-93, 2000.

[24] R. Roshandel, N. Medvidovic, “Toward Architecture-based Reliability Estimation”, In Proceeding of the Twin Workshops on Architecting Dependable Systems, International Conference on Software Engineering (ICSE), 2004, Edinburgh, UK, May 2004, and The International Conference on Dependable Systems and Networks (DSN) 2004, Florence, Italy.

[25] Tanir, Oryal, “Modeling Complex Computer and Communication Systems”, McGraw-Hill, 1997.

[26] B. Tekinerdogan, “ASAAM: Aspectual Software Architecture Analysis Method”, WICSA 4th Working IEEE/IFIP Conference on Software Architecture, 2004.

[27] Trader project web site, http://www.esi.nl/site/projects/trader, accessed May 12, 2005.

[28] S. Yakoub, B. Cukic, H. Ammar, “Scenario-based reliability analysis of component based software”, In Proc. 10th Int`l Symp. on Software Reliability Engineering (ISSRE’99), pp. 22-31, 1999.

[29] A. Zarras, V. Issarny, “Assessing Software Reliability at the Architectural Level”, In Proceedings of the 4th International Software Architecture Workshop, June 2000, Limerick, Ireland.