In Proc. of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS), 2005.

# Analysis and Verification of the Interaction Model in Software Design

Guido Menkhaus Urs Frei, Jörg Wüthrich

## Abstract

*One essential aspect in software design and software quality insurance is the complexity of component interdependencies. More complex designs drive the cost of production and maintenance. This paper proposes a new methodology for the analysis and verification of the interaction model of the software design throughout the software development lifecycle. For the analysis, the interaction model is described in an interaction model description language. Analysis techniques are applied to identify critical parts of the software application and to anticipate potential scenarios of failure modes. Coupling, cohesion, and instability metrics are computed on different levels of design refinement. They guide the analysis during the risk assessment of failure modes. The interaction model allows for the verification of the model against the underlying implementation of the software application. We provide tool support for all activities.*

## 1. Introduction

Software has a substantial influence on the quality of a product and on the services an organization offers [23]. Organizations increasingly rely on computerized systems and these systems need to be designed, constructed, integrated and maintained such to support the quality that justifies this reliance [27]. At the same time, the complexity of computerized systems increases continuously and offers more opportunities for malfunctions and their consequences [32]. As the complexity of applications increases, it is essential to introduce means to control the complexity of the application and to define adequate methods and tools for analyzing and verifying such applications to ensure their quality.

In this paper, we describe a methodology for the analysis and verification of a software application and the interaction model of its design. Most IT organizations allocate only a limited amount of time to analysis, verification, and testing of the software design. Since time is limited, the application of the methodology is only tractable with tool support. We present a toolchain that allows for efficient execution of the analysis and verification of a software application.

The remaining of the paper is structured as follows: Section 2 presents a short overview of verification and testing activities. Our methodology for analysis and verification of software is presented in Section 3 and the steps of fault prevention, projection, prediction and identification are subsequently discussed. Section 4 presents results. Related work is presented in Section 5 and Section 6 concludes the paper with a brief talk about our future work.

## 2. Overview of Verification and Testing Activities

Verification and testing has become the preferred process by which software is shown to possess a specific quality. The verification process demonstrates that a program correctly conforms to its specification. Testing tries to find cases where a program does not satisfy its specification [17]. It is very common to divide testing and verification into low-level and high-level tests [6] that correspond to two broad categories: Acceptance and unit tests [3, 4]. Unit tests are specified by developers basing on technical priorities and insight into the implementation. They accompany and guide the development and ensure that the application runs as expected. Acceptance tests, being functional or non-functional, are driven by business priorities and incorporate business requirements. However, testing has its limits: The deviation between the operational and the testing environment and the difficulty to reproduce those operational environments limits the failure rates that can be verified empirically and thus the detection of the number of critical units of a system. This can lead to a misunderstanding of the systems reliability properties [35]. To reduce the risk of software failures, an early and thorough analysis identifying risk-critical components is necessary. Risk is the likelihood that a fault leads to an application failure. Risk-based analysis provides a methodology to identify what to test and to better prioritize test cases.

Acceptance and unit tests base on business requirements and technical insight. However, *the extent to which the program correctness can be established is not purely a function of the programs external specifications and behavior but it depends critically upon its internal structure* [12]. The

complexity of a structure increases when the variety (number of distinguishable parts) and the interactions (connections) of parts or aspects increase [18]. Software design is the process of conceiving a structure that allows for opportunities to implement a solution and to limit the possibilities with the aim of restricting the complexity of the solution. The design process is an evolutionary process, in which a comprehensive design evolves over time in an iterative manner. Thus, most software systems are subject to a varying degree of design changes during their development and are revised to a greater extent as they are extended and adapted to new requirements. The design changes from iteration to iteration often entail complex interactions between components to come to an implementation of a feature. Design violations are often due to a lack of anticipation of unexpected interactions between a systems components, subsystems and layers. As complexity increases, design violations are more likely to occur when more interactions make it harder to identify all possible behaviors [35].

Studies have found that reworking defective requirements, design, and code typically consumes 40 to 50 percent of the total cost of software development [19]. A defect created in the design has a high impact if it is only detected in a later design iteration of a project. Consequently, more time needs to be spent on early analysis and verification of the design of the software application since testing always starts late. Defects found during testing may require massive rework, and it is more cost effectively done in the early phases of the software development lifecycle.

## 3. Analysis and Verification Methodology

Both approaches, analysis and verification/testing activities, aim at producing more reliable software [13].

1. *Testing and Verification.* On the one hand the aim of testing is to identify faults. On the other hand, it verifies that the software performs what it is supposed to do. Verification and testing however, does not prove that the behavior of software is correct, but provides a certain level of confidence.

2. *Analysis.* Analysis provides characteristics of the overall confidence in the behavioral software properties. Analysis establishes also indicators that some parts of the software system are more likely to be prone to errors than others.

For quality insurance, Bowen and Stavridou proclaim using a combination of approaches and methods [5]. Analysis, testing and verification approaches can be related and mapped to activities in the software development lifecycle. Iterations of the software lifecycle can be represented by the V-model [7]. The left side of the inner V in Figure 1 describes the design process for the construction of the system, starting from the requirements definition, followed by the system design and becoming more detailed at every step until the implementation phase, where the code is created. The implementation phase lies between construction and design (left side of the inner V) and the integration and verification (right side of the inner V). The right side of the inner V covers the different testing and verification phases, which relate to specific design phases of the left side of the inner V. In parallel to the activities on the left and the right side of the V-model run analysis activities. The left side covers activities that are applied during the specific construction and design phases before implementation has started. Activities on the right side of the V-model perform analysis after a significant part of the implementation has already been established. Our risk-based analysis and verification methodology includes the following steps and concentrates on the horizontal aspect of software design throughout the software development lifecycle (see Figure 1):

1. *Fault Prevention.* From the perspective of software engineering, the software design has a key impact on product quality and complexity control [8, 34]. A sound and verifiable design prevents or minimizes the occurrence and presence of faults. Tracz states in [37] that the most difficult problems in testing and analyzing software all revolve around the question of how software architectures are specified. We believe that relating implementation to models at different levels of abstraction and to be able to reason about them is essential for software quality control.

2. *Fault Projecting.* Petroski argues in [30] that many spectacular bridge failures have occurred because of copying attributes of successful design and focusing on previous successes instead of additionally considering possible failure modes. Projecting failure modes in software development is used to establish a fault hypothesis and estimate the presence of faults.

3. *Fault Prediction.* Managing complexity is the single most important aspect in software development. Fault prediction tries to identify complex structures that are likely to become a source of faults. For complex software there is a necessity to provide tool support for objective and repeatable measurements at different levels of design refinements.

4. *Fault Identification.* Identify faults and remove them by repeated verification. An implementation that results in non-compliance to the design (a fault with respect to the design) will not mean that the software does not deliver the intended functionality. However, it is a major issue in complexity control, reusability and modularity.

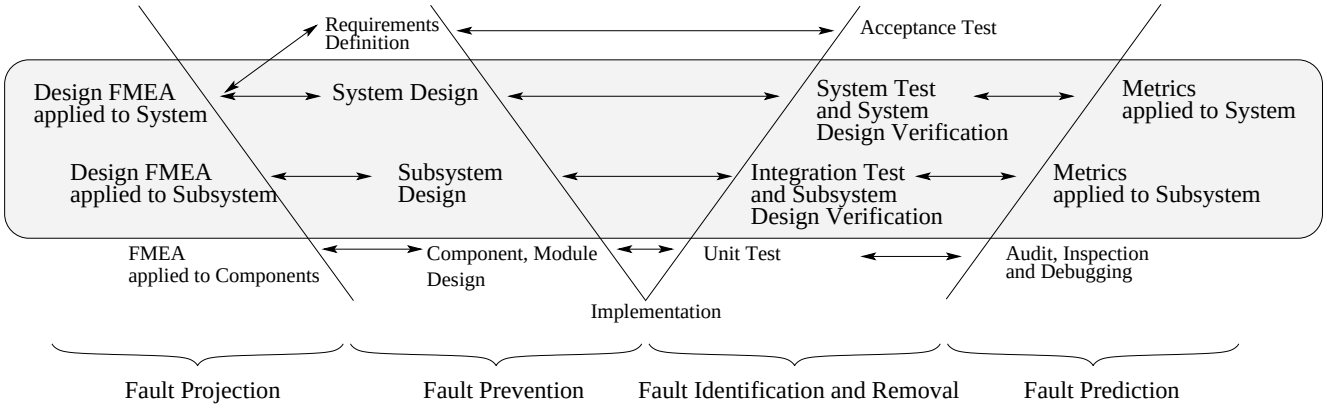


**Figure 1. Fault projection, prevention, identification, and prediction in the software development lifecycle.**

### 3.1. Fault Prevention by Model-based Design

All forms of engineering rely on models to understand complex, real-world systems. Software design produces models that provide abstractions for reasoning about the system by ignoring extraneous details while focusing on relevant ones. However, models and their resulting software applications diverge and software applications are likely to become complex and instable as they are developed, extended, and modified. The quality suffers as the complexity increases [22]. To control the growth of complexity in a system with a growing number of parts and connections, the complexity of each individual part must not increase with the growth of the system. Horizontal layering (abstraction), vertical layering (partitioning) and hierarchical division of systems are the means to combat the complexity of large systems [21].

We have developed an interaction model description language (IMDL), inspired by approaches of xArch [11], concepts for component-based modeling [16] and the description language employed in Dependometer [25]. The principle abstractions in our IMDL are layers, subsystem, connectors, components, actions, and interactions. Layers, subsystems, and connectors describe the logical view of the software design. Components, actions and interactions present the implementation view of the software (see Figure 2). The IMDL combines both views consistently in a single model.

We consider a software system, which is divided into a set of layers, denoted by $L = \{l_1, \ldots, l_n\}$. Each layer $l_i$ contains a set of subsystems $U(l_i) = \{u_{l_i,1}, \ldots, u_{l_i,m}\} \in U$. A connector $c$ describes interactions between layers and subsystems forming a dependency graph. This dependency graph describes the interaction model of a software application. The set of connectors of a subsystem or layer is denoted as $C(\cdot)$. A connector is directed and has

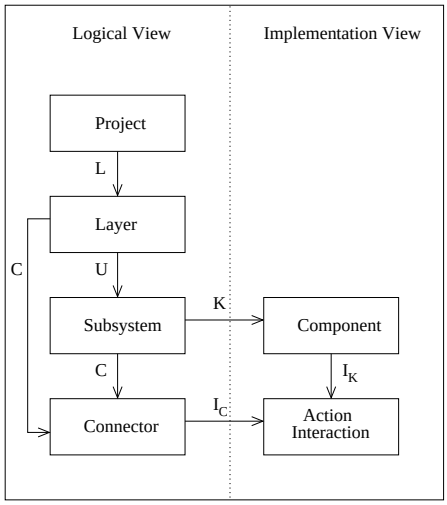


**Figure 2. Logical view of the software design and implementation view of the software application.**

a source $source(c) \in L \cup U$ and a target $target(c) \in L \cup U$. Each subsystem $u_j$ may consists of other subsystems and is implemented by a set of components $K(u_j) = \{k_{j,1}, \ldots k_{j,o}\} \in K$. A component $k$ has a vocabulary of actions $A(k)$ with $A = \bigcup_{k \in K} A(k)$. A connector $c$ is a subset of $A$ and defines a set of interactions $c = \{a_1, \cdots, a_p\}$, denoted by $I_C(c)$ and $I_K(k)$ denotes the set of interactions of a component $k$. An interaction $a_i$ has an initiator $init(a_i) \in K$ and a set of cooperators $coop(a_i) \in K$. If initiator and cooperator represent the same component, the interaction is called internal interaction; otherwise it is an external interaction. For the initiator, the interaction is an efferent interaction, for the cooperator, it is an afferent in-

teraction.

The IMDL describes the logical interaction model in the design of a software application and links it to the implementation. The model is subsequently used in the steps of fault projection, prediction, and identification.

### 3.2. Fault Projection applying FMEA

The Failure Mode and Effects Analysis (FMEA) is a risk-based analysis method that aims at identifying failure modes of a software application. It projects their effects, identifies their causes, designs detection and prevention mechanisms, and advises recommended actions to suppress these effects and to eliminate the causes of the failure modes [26, 29, 31]. Figure 3 shows the line of causal relationship between faults, errors, failures, and effects. The FMEA is developed along that line of cause and effect.

- *Fault.* The cause of a failure is a fault that ranges from specification and design defects to physical or human factors.
- *Error.* An error is a design flaw or a deviation from the desired or intended state of a system.
- *Failure.* A failure is defined as the manner in which a component, subsystem, or system could potentially fail to meet or deliver the intended function.
- *Effect.* The effect is the actual consequence of a system behavior in the presence of a failure.
- *Recommended Actions.* Actions that are not yet implemented but are recommended for implementation for identifying failure modes and reducing the probability of their occurrence.

A FMEA on the design level assessing risks can be performed early in the software development process. The objective is minimizing the impact of failure modes at a time when changes to the software system can be made most cost effectively [14]. This indicates the necessity to analyze the design of a software system thoroughly prior to construction and testing.

When conducting an FMEA, the following three steps need to be performed:

1. Identification of failure modes, effects, and possible causes, associated with the layers, subsystems and components described in the interaction model in the IMDL.
2. Assessment of the risk of the failure modes.
3. Determination of risk reduction activities.

Failure modes are ranked according to the risk priority number (RPN). The RPN of a failure $f$ depends on (1) the severity $s(f)$ of the failure's effect, (2) the likelihood of detection $d(f)$ of the error leading to the failure and (3) the frequency of occurrence $o(f)$ of the error's cause. The RPN of a failure $f$ is computed as:

$$RPN(f) = s(f)o(f)(1 - t(d(f)))$$

where $t$ is a function that tends asymptotically to 1 with increasing quality and number of detection mechanisms, $s$ increases gradually from no effect to hazardous effects, and $o$ from persistent faults to unlikely faults. In a system in which components interact with other components, a components failure might result in a fault of a component, which depends on it (see Figure 3). From the point of view of the depending component, the risk associated with that failure is denoted as the inherited risk and the failure is an inherited failure. The RPN of a component $k$ having a set of failures $F$ and a set of failures inherited from components it depends on $F_{\succ}$, is defined as:

$$RPN(k) = \sum_{f \in F} RPN(f) + \sum_{f \in F_{\succ}} RPN(f)(1 - t(d(f)))$$

The term $1 - t(d(\cdot))$ in the last equation denotes the likelihood of detection of an inherited failure in the implementation of the depending component.

The results of the FMEA is a ranking of

1. components, prioritized according to the RPN of the components and
2. failure modes, which have the highest risk impact on the complete software system.

For each failure mode there is a list of its effects and causes assigned. The highest ranking components and failure modes are selected and recommended actions are determined, such as changes in the design that could eliminate or reduce the probability of the occurrence of the potential failures.

Ideally, a FMEA is applied in regular intervals in evolutionary software development. It verifies whether the recommended actions were successfully implemented to guarantee constant improvement. This makes the FMEA tedious, laborious and time-consuming to carry out [28]. Therefore it is necessary to focus the FMEA. After part of the implementation is present, metrics are used to identify the most complex and instable and thus most error-prone parts of a software system. Their results support the analysis team, which performs the FMEA, in their decision-making process and let them focus the analysis on the risk intrinsic parts of a software system.

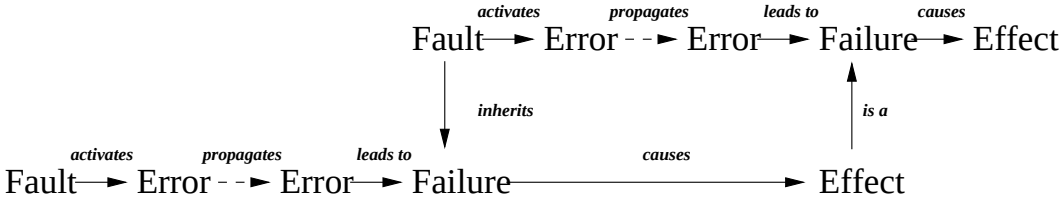


**Figure 3. Causal relationship between faults, errors, failures, and effects.**

### 3.3. Fault Prediction using Design Metrics

Hierarchically constructed software aims at producing systems that are either decomposable or near-decomposable, in which intra-layer, intra-subsystem and internal interactions are strong (high cohesion) and inter-layer, inter-subsystem and external interactions are negligible or weak (low coupling), respectively. As a rule-of-thumb, low complexity software should exhibit low coupling and high cohesion [9].

As outlined in Section 3.1 software design is performed on different levels of refinement. Metrics are computed on the basis of source code and are most often applied on the component level. They identify the most complex and therefore most error-prone and hard to test parts of a system. Employing our IMDL, we compute metrics at the level of components, subsystems and layers.

- *Component Level.* At the level of components we compute metrics measuring coupling and cohesion: Coupling between components and lack of cohesion of interactions [10].
- *Layer and Subsystem Level.* On the layer and subsystem level we compute a design instability metric [24].

**Component Level.** Coupling between components is a count of the number of other components to which a component is coupled: $cbo(k) = |\{a \in I_K(k) | coop(a) \neq k\}|$. Intensive coupling between components is opposed to modular design, prevents reuse and increases maintenance costs. The more independent a component, the easier it is to reuse it. Less independent components are more sensitive to changes in the design. Coupling metrics provide data indicating how complex the testing of parts of a design is likely to be. The higher the number of interactions, the more strict analysis and testing is required.

Lack of cohesion of interactions measures the dissimilarity of actions in a component by attributes. A component $k$ has a vocabulary of attributes $M(k) = \{m_1, \cdots, m_q\}$ and interactions $I_K(k)$. Let $A(I_K(k), m_j)$ denote the set of interactions interacting with attribute $m_j$. Lack of cohesion of a component $k$ is defined as $locoa(k) = (\frac{1}{|M(k)|} \sum_{m \in M(k)} |A(I_K(k), m)| - |I_K(k)|)/(1 - |I_K(k)|)$. High values indicate lack of cohesion of a component which means increased complexity and high likelihood of occurrence of errors during the development process. Low values imply simplicity and high reusability.

**Layer and Subsystem Level.** Stable layers and subsystems are both independent and highly responsible. They are independent, if they do not depend upon the results of other subsystems. They are called responsible, if changes of this subsystem have a strong impact on other subsystems. The responsibility, independence and instability of a layer or subsystem can be computed by measuring the numbers of their connections and interactions.

Instability of a layer is considered as the ratio of the number of efferent to the number of efferent and afferent connectors. Efferent connectors of a layer $l$ account for the number of connections of its subsystem $C(l) = \bigcup_{u \in U(l)} C(u)$ to subsystems in different layers. The set of efferent connections of a layer is defined as $ICE(l) = \bigcup_{u \in U(l)} \{c \in C(u) | init(c) \in C(l) \wedge coop(c) \notin C(l)\}$ and the set of afferent connections as $ICA(l) = \bigcup_{u \in U(l)} \{c \in C(u) | coop(c) \in C(l) \wedge init(c) \notin C(l)\}$. Instability of a layer $l$ is here computed as: $instability(l) = |ICE(l)|/|ICE(l) \cup ICA(l)|$. Analogous to the instability of a layer we describe the instability of a subsystem. Instability of a subsystem considers efferent and afferent interactions of its components to components in different subsystems. The set of efferent interactions of a subsystem $u$ is defined as $ICE(u) = \bigcup_{k \in K(u)} \{a \in I_K(k) | source(a) \in K(u) \wedge target(a) \notin K(u)\}$ and the set of afferent interactions as $ICA(u) = \bigcup_{k \in K(u)} \{a \in I_K(k) | target(a) \in K(u) \wedge source(a) \notin K(u)\}$. Instability is an indicator of the resilience to change. A value of zero means maximal stability and a value of one means maximal instability. Instable subsystems are generally undesirable and are recommended for careful design, implementation and testing.

**Automatic Computation.** The computation of metrics helps identifying complex, instable and therefore most error-prone components. An advantage of the presented metrics is the fact that they are automatically computed. The results of the computation of the metrics allow the analysis team to direct their effort of analysis to the indicated

critical parts of the software application. The team does not require performing a detailed analysis and inspection of the complete system.

### 3.4. Design Verification and Fault Identification

According to the laws of software evolution [22], the functional scope of a software system must be continually maintained, adapted and improved over the system development and lifetime to maintain usefulness. If the system is not adapted to take into account changes, the quality will decline. Thus, continuing change, adaptation and increase in volume and functional scope increases the complexity, unless work is done to maintain or to reduce it.

To control the complexity of a software product there is need for support of automatic and repeatable verification of the logical interaction model of the software design against the underlying implementation, which might deviate from the intended design over time. Given the importance of the software design and the laws of software evolution, we verify the design and identify violations of the implementation with respect to the logical interaction model.

A design violation is defined as an implemented interaction between two components, the two components being in two different subsystems or layers and there is no connector specified between the two subsystems and layers in the interaction model. Formally, we consider all connectors between layers and subsystems $C = \bigcup_{u \in L \cup U} C(u)$ and their interactions $IC = \bigcup_{c \in C} I_C(c)$. We check the implementation of all components $k \in K = \bigcup_{u \in L \cup U} K(u)$ for interactions $IK = \bigcup_{k \in K} I_K(k)$ and verify that those interactions comply with the interaction model. A design violation of the interaction model is defined as: $DVIM(k) = \{a | a \in IK \wedge a \notin IC \wedge init(a) = k\} \neq \emptyset$. Checking the model for violations in the implementation, identifying and removing those violations ensures consistency of model and implementation and allows for complexity control.

## 4. Results

Our analysis and verification methodology fosters continuous execution of activities for monitoring and controlling the design of a software system. We have developed a tool suite to support these activities: The fault avoidance (FA) tool allows for designing the interaction model and analyzing the software application. The fault identification (FI) tool computes metrics and verifies the interaction model. It uses the interaction model constructed and analyzed in the FA-tool.

Figure 4 presents a screenshot of the FA-tool Eclipse Plug-in. The FA-tool provides a perspective for performing a failure mode and effects analysis. Figure 4 shows the

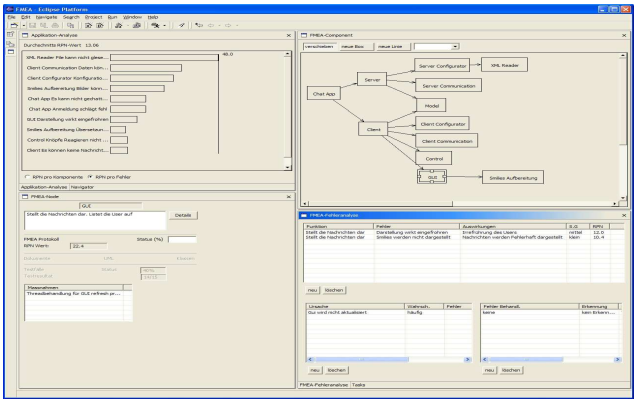

**Figure 4. FA-tool support for analyzing a software application.**

design of the interaction model of a chat-application and results of the analysis. The interaction model can be seen in the upper right corner. The results of the FMEA with respect to the risk assessment determining the RPN of each part of the application are shown in the upper left corner. The identified causes, failures, effects, and recommended actions of the subsystems are listed in panels at the bottom of the GUI.

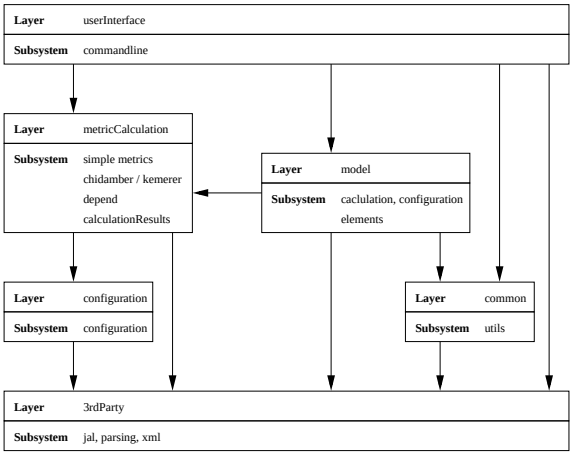


**Figure 5. Design of the interaction model of the FI-tool.**

The FI-tool is a command-line application that outputs a report, in which the results of the design metrics of Section 3.3 and the design violations of Section 3.4 are indicated. Currently, the FI-tool targets software applications written in Delphi. Figure 5 shows the design of the interaction model of the FI-tool representing its layers, subsystems, and layer connectors. Connectors at subsystem level and the subsystem's components are not shown. The *user-*

*Interface*-layer performs input and output tasks. The *model*-layer analysis the interaction model and initiates the computation of metrics at the layer and subsystem level. The *metricCalculation*-layer provides a set of metrics. The *configuration*-layer manages the configurations for the metric calculations and the *common* and the *3rdParty*-layers are proprietary and third party utilities.

The output of the FI-tool, with the IMDL of the FI-tool of Figure 5 as input, is a report indicating metric results and design violations. Table 1 shows results of the instability metric computed on the layer and subsystem level. They show that the *userInterface*-layer is the most unstable layer. This fact is supported by a high instability value of the *commandline*-subsystem. One of the most stable subsystems is the *calculation-results*-subsystem. However, the subsystems that are in the same layer (*metricCalculation*-layer) are highly unstable, resulting in an unstable layer. We have verified the interaction model of the FI-tool during its development and Listing 1 presents an extract of the results of an early version of the tool showing design violations. Line 2 shows that subsystem *3rdParty.parsing* interacts with subsystem *3rdParty.jal* although the design of the interaction model disallows this connection. The design violation from the *common* to the *configuration*-layer (Line 4) is caused by a connector from the subsystem *common.utils* to the subsystem *configuration.configuration* relating to an interaction between the components *DcaMetricManager* and *DcaMetricConfiguration*. More information about the concrete interaction that causes the violation can be found in the detailed report.

| Layer or subsystem | IA | IE | Instability |
|---|---|---|---|
| 3rdParty | 13 | 0 | 0 |
| configuration | 4 | 3 | 0.43 |
| common | 3 | 5 | 0.63 |
| metricCalculation | 4 | 6 | 0.6 |
| userInterface | 0 | 5 | 1 |
| model | 2 | 7 | 0.78 |
| xml | 1 | 0 | 0 |
| parsing | 19 | 1 | 0.05 |
| calculationResult | 14 | 0 | 0 |
| configuration | 7 | 4 | 0.36 |
| jal | 14 | 0 | 0 |
| elements | 9 | 1 | 0.1 |
| utils | 9 | 15 | 0.63 |
| commandline | 0 | 14 | 1 |
| configuration | 1 | 6 | 0.86 |
| calculation | 2 | 17 | 0.89 |
| chidamber-kemerer | 0 | 7 | 1 |
| depend | 0 | 8 | 1 |
| simplemetrics | 0 | 3 | 1 |

**Table 1. Instability metric results at layer and subsystem level.**

## 5. Related Work

Following the steps in the V-model of software development, research activities in model-based verification techniques can broadly be divided into three categories. Techniques that allow for constructing models from requirements and specifications, models that target primarily the interaction model of a software design and models for the composition of components.

- *Modeling of Requirements.* The Abstract State Machine Language (AsmL) is a modeling language for describing scenarios and use cases as sequences of events [2]. AsmL-models can be used for semi-automatic parameter generation, action call sequence generation and conformance testing. The AsmL test environment allows binding a model to an implementation, and using the model as a test oracle. However, formal specification languages have been applied to numerous problems but had only limited success in a small number of specific domains.

- *Modeling of Design.* The Unified Modeling Language (UML) is the standard for visually describing the structure and behavior of software systems. Goseva et al. present a risk analysis on the architectural level using UML models [15]. For verifying the conformance of a logical view of the design against the implementation Dependometer [25] and the Software Tomograph [36] use their proprietary IMDL.

- *Modeling of Component Composition.* Component-based engineering is widely used in all engineering disciplines. A framework for modeling of the behavior, interaction and execution model is proposed in [16]. However, a coherent tool support is missing and only specific aspects can be formally verified, such as timing constraints [1].

For software analysis, semantic metrics assess the quality of software and are meant to be computed from requirements or design specifications [33]. They are based on the vague notion of ideas and concepts and cannot be automatically computed. Worse, in most projects requirements and specifications are incomplete, change often and most completed systems have implemented only a small fraction of the originally-proposed features and functions specified in the requirements.

Most analysis techniques are informal and use the insight of the system architect. The architecture tradeoff analysis method is used to base architectural design decisions on rational goal-based attributes [20]. Quality attributes such as

**Listing 1. Example of design violations of the FI-tool.**

```
1       ...
2       3rdParty.parsing                  -> 3rdParty.jal
3       ...
4       common                            -> configuration
5       common.utils                      -> configuration.configuration
6       common.utils.DcaMetricManager -> configuration.configuration.DcaMetricConfiguration
7       ...
```

modifiability, safety, and security are measured using inspections. Scenarios guide the analysis in the identification of risks, non-risks, sensitivity and tradeoff points in the architecture.

Most of the techniques target a specific aspect of software testing and are not integrated into a concept covering a complete horizontal layer of the development lifecycle from fault projection and prevention to fault prediction, identification and removal.

## 6. Conclusion

We presented a methodology for analyzing and verifying the interaction model of software design. Tool support for performing the analysis and verification activities has been developed to support automatic and repeatable measurements of interaction dependencies between software components, subsystems, and layers. The steps of our methodology start from fault prevention through logical interaction model design, fault projection applying risk priority number based analysis and evaluation, fault prediction by quantifying interactions between components, subsystems, and layers, and fault identification and removal by verifying the logical interaction model against the implementation of the software system.

In our future work we will investigate metrics, computed on the interaction model, which point out design weaknesses and potential design flaws. For this purpose it is less necessary to compute precise metrics. The benefit comes from being able to reveal and identify sensitivity and tradeoff points in the design.

## References


[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. In *Proc. of TACAS 2002*, volume 2280 of *LNCS Springer*, pages 460 – 464, 2002.

[2] M. Barnett, W. Grieskamp, Y. Gurevich, W. Schulte, N. Tillmann, and M. Veanes. Scenario-oriented Modeling in AsmL and its Instrumentation for Testing. In *Proc. of SCESM'03*, 2003.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] K. Beck. *Test Driven Development*. Addison-Wesley Professional, 2002.

[5] J. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *Software Engineering Journal*, 1992.

[6] B. Broekman and E. Notenbool. *Testing Embedded Software*. Addison Wesley, 2003.

[7] A. Burnard. Verifying and Validating Automatically Generated Code. In *Proc. of International Automotive Conference (IAC)*, Stuttgart, Germany, 2004.

[8] W. Chapman, A. Bahill, and A. Wymore. *Engineering, Modelling and Design*. CRC Press, 1992.

[9] D. Chen and M. Törngren. A Metrics System for Quantifying Operational Coupling in Embedded Computer Control Systems. In *Proc. of EMSOFT'04*, pages 184 – 192, Pisa, Italy, 2004.

[10] S. Chidamber and C. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315, 1993.

[11] E. Dashof, A. v.d. Hoek, and R. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Proc. of WICSA 2001*, Amsterdam, Netherlands, 2001.

[12] E. Dijkstra. *Structured Programming*, chapter Notes on Structured Programming, pages 1 – 82. Academic Press, London, UK, 1972.

[13] F. Doerenberg. *Analysis and Synthesis of Dependable Computing and Communication Systems*, chapter Dependability Impairments: Faults, Errors and Failures. www.nonstopsystems.com. to be published, 2004.

[14] P. L. Goddard. Software FMEA Techniques. In *IEEE Proc. Annual Reliability and Maintainability Symposium*, 2000.

[15] K. Goseva-Popstojanova, A. Hassan, A. G. W. Abdelmoez, D. Nassar, H. Ammar, and A. Mili. Architectural-Level Risk Analysis Using UML. *IEEE Transaction on Software Engineering*, 29(10):946 – 960, October 2003.

[16] G. Gössler and J. Sifakis. Composition for Component-based modeling. In *Proc. of FMCO'02*, Leiden, Netherlands, 2002.

[17] B. Hailpern and P. Santhanam. Software debugging, testing and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[18] F. Heylighen. *The Evolution of Complexity*, chapter The Growth of Structural and Functional Complexity during Evolution. Kluwer Academic Publishers, 1996.

[19] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[20] R. Kazman, M. Klein, and P. Clements. Evaluating Software Architectures for Real-Time Systems. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 USA, Jul 1999.

[21] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, 23(1), 2002.
[22] M. Lehmann. Software evolution - cause and effects. In *Proc. of the 9th International Stevens Awards at IEEE ICSM*, 2003.
[23] Lufthansa. Failure of check-in system disrupts services. Deutsche Lufthansa AG Corporate Communications , September 2004.
[24] R. Martin. OO Design Quality Metrics (An Analysis of Dependencies). *ROAD*, 1995.
[25] D. Menges and I. Otter. Dependometer. Sourceforge, 2003.
[26] G. Menkhaus and B. Andrich. Metric Suite for Directing the Failure Mode Analysis of Embedded Software Systems. In *Proc. of ICEIS*, 2005.
[27] G. Menkhaus and U. Frei. Legacy System Integration using a Grammar-based Transformation System. *CIT - Journal of Computing and Information Technology*, 12(2):95 – 102, 2004.
[28] H. Parkinson, G. Thomson, and S. Iwnicki. The development of an FMEA methodology for rolling stock remanufacture and software quality. *ImechE Seminar Publication*, 20:55 – 66, 1998.
[29] H. Pentti and H. Atte. Failure mode and effects analysis of software-based automation systems. Technical Report STUK-YTO-TR 190, STUK, Helsinki, Aug. 2002.
[30] H. Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge University Press, 1994.
[31] SAE. Surface vehicle recommended practice. Technical Report SAE-J1739, Society of Automotive Engineers, Warrendale, USA, 2002.
[32] L. Sha. Using Simplicity to Control Complexity. *IEEE Software*, pages 20 – 28, July/August 2001.
[33] C. Stein, L. Etzkorn, and D. Utley. Computing Software Metrics from Design Documents. In *ACMSE*, 2004.
[34] N. Suh. *Axiomatic Design: Advances and Applications*. Oxford University Press, 2001.
[35] H. Thane. Safe and Reliable Computer Control Systems Concepts and Methods. Technical Report ISRN KTH/MMK/R–96/13-SE, Mechatronics Laboratory, Department of Machine Design, Royal Institute of Technology, KTH, Stockholm, Sweden, 1996.
[36] S. Tomography. www.software-tomography.com, 2003.
[37] W. Tracz. Testing and Analysis of Software Architectures. In *Proc. of ACM ISSTA96*, S. Diego, USA, 1996.