Component-based Development of Dependable Systems with UML

Jan Jürjens and Stefan Wagner

Software & Systems Engineering
Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany
http://www4.in.tum.de/~juerjens – http://www4.in.tum.de/~wagnerst

Abstract. Dependable systems have to be developed carefully to prevent loss of life and resources due to system failures. Some of their mechanisms (for example, providing fault-tolerance) can be complicated to design and use correctly in the system context and are thus error-prone. This chapter gives an overview of reliability-related analyses for the design of component-based software systems. This enables the identification of failure-prone components using complexity metrics and the operational profile, and the checking of reliability requirements using stereotypes. We report on the implementation of checks in a tool inside a framework for tool-supported development of reliable systems with UML and two case studies to validate the metrics and checks.

1 Introduction

There is an increasing desire to exploit the flexibility of software-based systems in the context of critical systems where predictability is essential. Examples include the use of embedded systems in various application domains, such as fly-by-wire in Avionics, drive-by-wire in Automotive and so on. Given the high reliability requirements in such systems (such as a maximum of 10^{-9} failures per hour in the avionics sector), a thorough design method is necessary. We define reliability as the probability of failure-free functioning of a software component for a specified period in a specified environment.

Reliability mechanisms cannot be "blindly" inserted into a critical system, but the overall system development must take these aspects into account. Furthermore, sometimes such mechanisms cannot be used off-the-shelf, but have to be designed specifically to satisfy given requirements. For example, the use of redundancy mechanisms to compensate for the failures that occur in any operational system may require complex protocols whose correctness can be non-obvious [Rus94]. This can be non-trivial, as spectacular examples for software failures in practice demonstrate (such as the explosive failure of the Ariane 5 rocket in 1997).

Any support to aid reliable systems development would thus be useful. In particular, it would be desirable to consider reliability aspects already in the design phase, before a system is actually implemented, since removing flaws

in the design phase saves cost and time. This is significant; for example, in avionics, verification costs represent 50% of the overall costs. Moreover a means to estimate reliability, or at least identify failure-prone components, early in the life-cycle of the software would be helpful to make verification more efficient. We believe that the design models are the best indicator in early phases for the future behavior of the system and thus should be used for reliability estimation. Following an idea advocated in [ABW03], we thus aim to incorporate quality attributes of models (such as measures derived from structural or behavioral attributes) into component-based models of software systems within the context of model-based software development.

As a design notation, we use the Unified Modeling Language (UML) [Obj03], the de facto industry-standard in object-oriented modeling. It offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

Problems in critical systems development often arise when the conceptual independence of software from the underlying physical layer turns out to be an unfaithful abstraction (for example in settings such as real-time or more generally safety-critical systems, see [Sel02]). Since UML allows the modeler to describe different views on a system, including the physical layer, it seems promising to try to use UML to address these problems by modeling the interdependencies between the system and its physical environment.

To support safe systems development, safety checklists have been proposed for example in [HJL96]. In the present chapter, based on an extending work presented in [Jür03b], we tailor UML in a similar approach to reliable systems by precisely defining some checks with stereotypes capturing reliability requirements and related physical properties. We also provide metrics that estimate the failure-proneness of a software system based on the complexity of its design models and its operational profile. The reliability requirements can then be compared with the results for failure-proneness.

In this way we encapsulate knowledge on prudent reliability engineering and thereby make it available to developers who may not be specialized in reliable systems. A prototypical framework for tool-support for this approach is also presented within this chapter.

Outline In Sect. 2.1 we explain the foundation for checking the constraints associated with the stereotypes suggested for reliable systems development which are presented in Sect. 2.2, together with examples of their use. A metrics suite for models is defined in Sect. 3.1 and these metrics are used to analyze the failure-proneness of components in Sect. 3.2. In Sect. 4, we briefly describe the tool assisting our approach. Two case studies describing an automatic collision notification system in Sect. 5 and an automotive network controller in Sect. 6 are finally used to validate our work.

2 Model-based Reliability Specification and Analysis

In safety-critical systems, an important concept also used here is that of a *safety level* (see e.g. [Ran00]). Since safety-critical systems generally need to provide a high degree of reliability, it makes sense to analyze these systems with respect to their maximum allowed failure rate. We thus define the concept of a *reliability level* analogous to the mentioned safety levels.

We exemplarily consider the following kinds of failure semantics in this paper (other kinds have to be omitted for space reasons).

- crash/performance failure semantics means that a component may crash or may deliver the requested data only after the specified time limit, but it is assumed to be partially correct.
- value failure semantics means that a component may deliver incorrect values.

Possible failures include:

message loss which may be due to hardware failures or software failures (for example, buffer overflows)

message delay which may in turn result into the reordering of messages if the delay is variable

message corruption when a message is modified in transit.

Forms of redundancy commonly employed against these failures include space redundancy (physical copies of a resource), time redundancy (rerunning functions) and information redundancy (error-detecting codes).

UML profile mechanisms We use the three main profile mechanisms (stereotypes, tagged values and constraints) to include reliability requirements in a UML specification, together with the constraints formalizing the requirements. To evaluate a model against the requirements, we refer to a precise semantics for the used fragment of UML extended with a notion of *failures* sketched in Sect. 2.1.

2.1 Evaluation of Reliability Requirements in UML Diagrams

We briefly give an idea how the constraints used in the UML extension presented in Sect. 2.2 can be checked in a precise and well-defined way. A precise semantics for a (restricted and simplified) fragment of UML supporting these ideas can be found in [Jür04]. It includes activity diagrams, statecharts, sequence diagrams, composite structure diagrams, deployment diagrams, and subsystems, each restricted and simplified to keep a mechanical analysis that is necessary for some of the more subtle behavioral reliability requirements feasible. The subsystems integrate the information between the different kinds of diagrams and between different parts of the system specification. For reliability analysis, the reliability-relevant information from the reliability-oriented stereotypes is then incorporated as explained below.

Outline of precise semantics In UML the objects or components communicate through messages received in their input queues and released to their output queues. Thus for each component C of a given system, the semantics defines a function $[\![C]\!]()$ which

- takes a multi-set[1] I of input messages and a component state S and
- outputs a set $[\![C]\!](I, S)$ of pairs (O, T) where O is a multi-set of output messages and T the new component state (it is a *set* of pairs because of the non-determinism that may arise)

together with an *initial state* S_0 of the component. The behavioral semantics $[\![D]\!]()$ of a state machine diagram D models the run-to-completion semantics of UML state machines. Similarly, one can define the semantics for UML 1.5 activity diagrams. Given a sequence diagram S, we define the behavior $[\![S.C]\!]()$ of each contained component C. Subsystems group together diagrams describing different parts of a system: a system component C given by a subsystem $\mathcal{S}$ may contain subcomponents $C_1, \ldots, C_n$. The behavioral interpretation $[\![\mathcal{S}]\!]()$ of $\mathcal{S}$ is defined by iterating the following steps:

(1) It takes a multi-set of input events.
(2) The events are distributed from the input multi-set and the link queues connecting the subcomponents and given as arguments to the functions defining the behavior of the intended recipients in $\mathcal{S}$.
(3) The output messages from these functions are distributed to the link queues of the links connecting the sender of a message to the receiver, or given as the output from $[\![\mathcal{S}]\!]()$ when the receiver is not part of $\mathcal{S}$.

When performing reliability analysis, after the last step, the failure model may corrupt the contents of the link queues in a certain way explained below. Note that this approach is similar to that taken in [Jür04], where a security analysis is performed in place of the reliability analysis.

As an example, the statechart in Fig. 1 is executed as follows: The fuel controller starts out in state WheelsOut. It awaits either the message fuel() or wheelsin(). In the first case, the argument of the message is multiplied with the constant d and the result returned. In the second case, if the argument is false, no change occurs. In case of true, the state is switched to WheelsIn. In that state, the same behavior occurs, except that the argument of fuel() is now multiplied with the constant c.

Reliability analysis For a reliability analysis of a given UML subsystem specification $\mathcal{S}$, we need to model potential failure behavior. We model specific types of failures that can corrupt different parts of the system in a specified way, depending on the used redundancy model. For this we assume a function $\mathsf{Failures}_R(s)$ which takes a *redundancy model* R and a stereotype $s \in \{\text{«crash/performance»}, \text{«value»}\}$ and returns a set of expressions $\mathsf{Failures}_R(s) \subseteq$

[1] A *multi-set* (also called a *bag*) is a set whose elements may occur more than once.

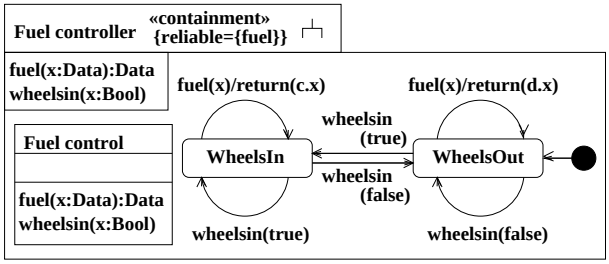

Fig. 1. Example Statechart

Risk	$\mathsf{Failures}_{none}()$
Crash/performance	$\{delay(t), loss(p)\}$
Value	$\{corruption(q)\}$

Table 1. Failure semantics

$\{delay(t) : t \in \mathbb{N} \wedge t > 0\} \cup \{loss(p) : p \in [0,1]\} \cup \{corruption(q) : q \in [0,1]\}$. Here R is a name representing a redundancy mechanism (such as duplication of components together with a voting mechanism), which is semantically defined through the $\mathsf{Failures}()$ sets. The natural number t represents the maximum delay to be expected in time units. p gives the probability that an expected data value is not delivered after the t time units specified in $delay(t)$. Given a value delivered within this time period, q denotes the probability that this value is corrupted. As an example for a failures function, Table 1 gives the one for the absence of any redundancy mechanism ($R = none$). Here, the time and probability parameters are still included as parameters; for a given system, these will be concrete numeric values.

The consistency of the failure model with the physical reality (and in particular the completeness in the sense that no possible failures are missing from the failure model) can for example be established by simulating the model and comparing the results with data obtained from experiments on the physical systems. Similarly, the probabilistic values and other numerical data can also be derived. Note that this consistency cannot, for principled reasons, not be *proved*, since mathematical proofs can only be constructed with respect to mathematical models of reality, not reality itself. The consistency of the running code with the execution semantics of the UML diagrams used here can be guaranteed in two ways: Firstly, one can use automated code generation, where the code generator would (for high assurance applications) ideally be formally proved to be correct with respect to the semantics. Where code generation is not useful, the manually implemented code can still be checked against the semantics by using model-based test-sequence generation (this is not considered here; [Jür03a] gives an introduction).

Then we model the actual behavior of a failure, given a redundancy model R, as a *failure function* that, at each iteration of the system execution, nondeterministically maps the contents of the link queues in $\mathcal{S}$ and a state S to the new contents of the link queues in $\mathcal{S}$ and a new state T as explained below.

For this, for any link l, we use a sequence $(lq_n^l)_{n\in\mathbb{N}}$ of multi-sets such that at each iteration of the system, for any n, lq_n^l contains the messages that will be delayed for further n time units. Here lq_0^l stands for the actual contents of the link queue l. At the beginning of the system execution, all these multi-sets are assumed to be empty. Also, for any execution trace h (that is, a particular sequence of system states and occurring failures describing a possible history of the system execution), we define a sequence $(p_n^h)_{n\in\mathbb{N}}$ of probabilities such that at the nth iteration of the system, the failure considered in the current execution trace happened with probability p_n^h. Thus the probability p^h that a trace h of length n will take place is the product of the values $p_1^h, \ldots, p_n^h$ (since in our presentation here, we assume failures to be mutually independent, to keep the exposition accessible). Then for an execution trace h, the failure function is defined as follows. It is non-deterministic in the sense that for each input, it may have a *set* of possible outputs.

- For any link l stereotyped s where $loss(p) \in \mathsf{Failures}_R(s)$ we
  - either define $lq_0^l := \emptyset$ and append p to the sequence $(p_n^h)_{n\in\mathbb{N}}$,
  - or append $1-p$ to the sequence $(p_n^h)_{n\in\mathbb{N}}$.
- For any link l stereotyped s where $corruption(q) \in \mathsf{Failures}_R(s)$ we
  - either define $lq_0^l := \{\Box\}$ and append q to the sequence $(p_n^h)_{n\in\mathbb{N}}$,
  - or append $1-q$ to the sequence $(p_n^h)_{n\in\mathbb{N}}$.
- For any link l stereotyped s where $delay(t) \in \mathsf{Failures}_R(s)$ and $lq_0^l \neq \emptyset$, we define $lq_n^l := lq_0^l$ for some $n \leq t$ and append $1/t$ to the sequence $(p_n^h)_{n\in\mathbb{N}}$.
- Then for each n, we (simultaneously) define $lq_n^l := lq_{n+1}^l$.

The failure types define which kind of failure may happen to a communication link with a given stereotype, as explained above. Note that for simplicity we assume that delay times are uniformly distributed. Also, corrupted messages (symbolized by $\Box$) are assumed to be recognized (using error-detecting codes). To evaluate the reliability of the system with respect to the given type of failure, we define the *execution of the subsystem $\mathcal{S}$ in presence of a redundancy model R* to be the function $[\![\mathcal{S}]\!]_R()$ defined from $[\![\mathcal{S}]\!]()$ by applying the failure function to the link queues as a fourth step in the definition of $[\![\mathcal{S}]\!]()$ as follows:

(4) The failure function is applied to the link queues as detailed above.

Containment A system ensures *containment* if there is no unreliable interference between components on different reliability levels (this is called *non-interference* in [DS99]). Intuitively, providing containment means that an output should in no way depend on inputs of a lower level. We assume that we are given an ordered set *Levels* of *reliability levels*. Then the *containment* constraint is that in the system, the value of any data element of level l may only be influenced by data of the same or a higher reliability level: Write $H(l)$ for the set of messages of level l or higher. Given a sequence $\boldsymbol{m}$ of messages, we write $\boldsymbol{m}\restriction_{H(l)}$ for the sequence of messages derived from those in $\boldsymbol{m}$ by deleting all events the message names of which are not in $H(l)$. For a set M of sequences of messages, we define $M\restriction_H \stackrel{\text{def}}{=} \{\boldsymbol{m}\restriction_H : \boldsymbol{m} \in M\}$.

Stereotype	Base Class	Tags	Constraints	Description
risk	link, node	failure		risks
crash/ performance	link, node			crash/performance failure semantics
value	link, node			value failure semantics
guarantee	link, node	goal		guarantees
redundancy	dependency, component	model		redundancy model
reliable links	subsystem		dependency reliability matched by links	enforces reliable communication links
reliable dependency	subsystem		«call», «send» respect data reliability	structural data reliability
critical	object	(*level*)		critical object
reliable behavior	subsystem		behavior fulfills reliability	reliable behavior
containment	subsystem		provides containment	containment
error handling	subsystem	error object		handles errors

Table 2. Stereotypes

Tag	Stereotype	Type	Multipl.	Description
failure	risk	$\mathcal{P}(\{delay(t), loss(p), corruption(q)\})$	*	specifies risks
goal	guarantee	$\mathcal{P}(\{immediate(t), eventual, correct\})$	*	specifies guarantees
model	redundancy	$\{none, majority, fastest\}$	*	redundancy model
error object	error handling	string	1	error object

Table 3. Tags

Definition 1. *Given a component C and a reliability level l, we say that C* provides containment *with respect to l if for any two sequences $\boldsymbol{i}, \boldsymbol{j}$ of input messages, $\boldsymbol{i}\downarrow_{H(l)} = \boldsymbol{j}\downarrow_{H(l)}$ implies $[\![C]\!]\boldsymbol{i}\downarrow_{H(l)} = [\![C]\!]\boldsymbol{j}\downarrow_{H(l)}$.*

2.2 Stereotypes for Reliability Analysis: The "Reliability Checklist"

In Table 2 we give some of the stereotypes, together with their tags and constraints, that we suggest to be used in the model-based development of reliable systems with UML, based on previous experience in the model-based development of reliable systems (for space restrictions, we can only give a representative selection). Thus, in a way, we define a UML-based "Reliability Checklist" (which one can verify mechanically on the design level). The constraints, which in the table are only named briefly, are formulated and explained in the remainder of the section. Table 3 gives the corresponding tags. The relations between the elements of the tables are explained below in detail.

Note that some of the concepts introduced below are easier to apply at component rather than object level. We explain the stereotypes and tags given in Tables 2 and 3 and give examples (which for space restrictions have to be kept

simple). Note that the constraints considered here span a range in sophistication: Some of the constraints are relatively simple (comparable to type-checking in programming languages) and can be enforced at the level of abstract syntax (such as «reliable links») and can be used without the semantics sketched in Sect. 2.1. Others (such as «containment») refer to the semantics and can only be checked reliably using tool-support.

Overview We give an overview of the syntactic extensions together with an informal explanation of their meaning. «redundancy», with associated tag {model}, describes the redundancy model that should be implemented. «risk» describes the risks arising at the physical level using the associated tag {failure}. «guarantee» requires the goals described in the associated tag {goal} for communicated data. «reliable links» ensures that reliability requirements on the communication are met by the physical layer. «critical» labels critical objects using the associated tags {level} (for each reliability level *level*). «reliable dependency» ensures that communication dependencies respect reliability requirements on the communicated data. «reliable behavior» ensures that the system behaves reliably as required by «guarantee», in the presence of the specified failure model. «containment» ensures *containment* as defined in Definition 1. «error handling» with tag {error object} provides an object for handling errors.

We define the stereotypes and their constraints in detail.

Redundancy The stereotype «redundancy» of dependencies and components and its associated tag {model} can be used to describe the redundancy model that should be implemented for the communication along the dependency or the values computed by the component. Here we consider the redundancy models *none, majority, fastest* meaning that there is no redundancy, there is replication with majority vote, or replication where the fastest result is taken (but of course there are others, which can easily be incorporated in our approach).

Risk, crash/performance, value With the stereotype «risk» on links and nodes in deployment diagrams one can describe the risks arising at these links or nodes, using the associated tag {failure}, which may have any subset of $\{delay(t), loss(p), corruption(q)\}$ as its value. In the case of nodes, these concern the respective communication links connected with the node. Alternatively, one may use the stereotypes «crash/performance» or «value», which describe specific failure semantics (by giving the relevant subset of $\{delay(t), loss(p), corruption(q)\}$): For each redundancy model R, we have a function $\mathsf{Failures}_R(s)$ from a given stereotype $s \in \{$«crash/performance», «value»$\}$ to a set of strings $\mathsf{Failures}_R(s) \subseteq \{delay(t), loss(p), corruption(q)\}$.

If there are several such stereotypes relevant to a given link (possibly arising from a node connected to it), the union of the relevant failure sets is considered. This way we can evaluate UML specifications. We make use of this for the constraints of the remaining stereotypes. An example for a failures function was given above in Table 1.

Guarantee «call» or «send» dependencies in object or component diagrams stereotyped
«guarantee» are supposed to provide the goals described in the associated tag {goal} for the data that is sent along them as arguments or return values of operations or signals. The goals may be any subset of $\{immediate(t), eventual(p), correct(q)\}$. This stereotype is used in the constraints for the stereotypes «reliable links» and «reliable behavior».

Reliable links The stereotype «reliable links», which may label subsystems, is used to ensure that reliability requirements on the communication are met by the physical layer. We recall that in UML deployment diagrams, communication is specified on the logical level by communication dependencies between components, which is supported on the physical level by communication links between the nodes on which the components reside. More precisely then, the constraint enforces that for each dependency d with redundancy model R stereotyped «guarantee» between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype s such that

- if {goal} has $immediate(t)$ as one of its values then $delay(t') \in \mathsf{Failures}_R(s)$ entails $t' \leq t$,
- if {goal} includes $eventual(p)$ as one of its values then $loss(p') \in \mathsf{Failures}_R(s)$ entails $p' \leq 1 - p$, and
- if {goal} has $correct(q)$ as one of its values then $corruption(q') \in \mathsf{Failures}_R(s)$ entails $q' \leq 1 - q$.

Example In Fig. 2, given the redundancy model $R = none$, the constraint for the stereotype «reliable links» is fulfilled if and only if $T \leq t$, where t is the expected delay according to the $\mathsf{Failures}_{none}(crash/performance)$ scenario in Fig. 1.

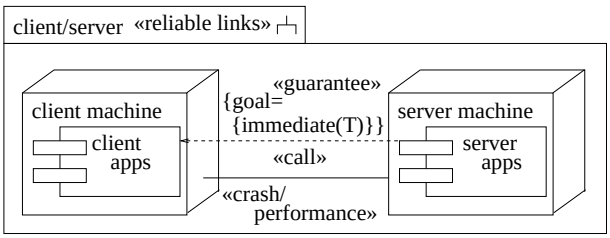

Fig. 2. Example *reliable links* usage

Reliable behavior The stereotype «reliable behavior» ensures that the specified system behavior in the presence of the failure model under consideration does provide the reliability goals stated in the tag {goal} associated with the stereotype «guarantee» as follows, by referring to the semantics sketched in Sect. 2.1.

immediate(t) In any trace h of the system, the value is delivered after at most t time steps in transmission from the sender to the receiver along the link l. Technically, the constraint is that after at most t steps the value is assigned to lq_0^l.

eventual(p) In any trace h of the system, the probability that delivered value is lost during transmission is at most $1 - p$. Technically, the sum of all p^h for such histories h is at most $1 - p$.

correct(q) In any trace h of the system, the probability that delivered value is corrupted during transmission is at most $1 - q$. Technically, the sum of all p^h for such histories h is at most $1 - q$.

3 Model-based Reliability Metrics

This section describes the possibilities and benefits of a reliability-related analysis of models based on complexity metrics. We first explain the motivation and assumed development process. Afterwards specific metrics for structured classes and state machines are proposed and combined with established object-oriented design metrics. These metrics are joined with the operational profile of the system to find the failure-prone components. This information is finally used in combination with the reliability requirements from Sec. 2.

The main idea is to identify failure-prone components early in the life cycle of the software by their complexity measures and operational profile, and use this information in checks regarding reliability requirements. The complexity information can help us to rethink design decisions and simplify the design in general. The further analysis can guide the test and review efforts to concentrate on the more critical and failure-prone components. Finally, annotated reliability requirements can be checked for consistency with this information.

3.1 Model Complexity Metrics

The complexity of software code has been studied to a large extent. The most widely known metrics concerning complexity are *Halstead's Software Metric* [Hal77] and *McCabe's Cyclomatic Complexity* [McC76] and many variations of these. In [KW90,MK96] it is shown that the reliability of a software is related to its complexity. It is generally accepted that complexity is a good indicator for the reliability of a component. This means that a component with a high complexity is more likely to contain faults. Depending on the operational profile of the component, this can mean that the reliability is low. For example, it is stated in [RHS98] that a combination of size and cyclomatic complexity delivers good results in reliability prediction.

Although the traditional complexity metrics are not easily applicable to design models, there are already a number of approaches that propose design metrics [CA88,BHS97,WWC99]. However, they concentrate mainly on the structure or do not support object-oriented designs. Nevertheless there are also various metrics for object-oriented design models. The most important is the metrics

suite proposed in [CK94] which concentrates on various aspects of classes. Most of these metrics were found to be good estimators of fault-prone classes in [BBM96] and will be used and extended in the following. In using a suite of metrics we follow [MGBB90,FP97] stating that a single measure is usually inappropriate to measure complexity.

Development Process The metric suite described below is generally applicable in all kinds of development processes. It does not need specific phases or sequences of phases to work. However, we need detailed design models of the software to which we apply the metrics. This is most rewarding in the early phases as the models then can serve various purposes. Otherwise, we assume no specific process (apart from being model-based) and therefore omit details on possible process models. The idea is mainly to incorporate reliability aspects during the development of the model.

We base our metrics especially on some parts of UML 2.0 that are most relevant to embedded systems development. The parts that we will look at are classes, structured classes, components, and state machines.

We adjust new metrics and the ones from [CK94] to parts of UML 2.0 based on the design approach taken in ROOM [SGW94] or UML-RT [SR98], respectively. This means that we model the architecture of the software with structured classes (called actors in ROOM, capsules in UML-RT) that are connected by ports and connectors to describe the interfaces and which can have associated state machines that describe their behavior. The structured classes can have parts that may themselves be structured. Thus a hierarchical system decomposition is possible.

The metrics defined below for the different model elements can predict the fault-proneness of the components. To be able to make a reliability analysis we need information about the *failure*-proneness of components, i.e. the probability that the fault causes a failure. We will use a very simple form of an operational profile [Mus99] to determine the usage level of a component. Therefore the development process must support the creation of operational profiles early in the development.

Structured Classes and Components Structured classes and components are a new concept in UML 2.0 derived mainly from ROOM and UML-RT. It introduces composite structures that represent a composition of run-time instances collaborating over communications links. This allows UML components and classes to have an internal structure consisting of other components or classes that are bound by connectors. Furthermore ports are introduced as a defined entry point to a class or component. A port can group various provided and required interfaces. A connection between two classes or components through ports can also be denoted by a connector. The parts of a class or component work together to achieve its behavior. A state machine can also be defined to describe additional behavior.

The metrics defined in this section are applicable to components as well as classes. However, we will concentrate on structured classes following the usage of classes in ROOM. Therefore the set of documents under consideration in the following are composite structure diagrams of single classes or components with their parts, provided and required interfaces, connectors and their state machines if existing. An example for this is depicted in Fig. 3.

Number of Parts (NOP) The number of parts of a structured class or component contributes obviously to its structural complexity. The more parts it has, the more coordination is necessary because of the more dependencies. Therefore, we define *NOP* as the number of direct parts C_p of a class or component.

Number of Required Interfaces (NRI). This metric is (together with the NPI metric below) based on the fan-in and fan-out metrics from [HK81] and is also a substitute for the old *Coupling Between Objects (CBO)* that was criticized in [MH99] in that it does not represent the concept of coupling appropriately. It reduces ambiguity by giving a clear direction of the coupling. We use the required interfaces of a class to represent the usage of other classes. This is another increase of complexity which may as well lead to failure, for example if the interfaces are not correctly defined. Therefore we count the number of required interfaces I_r for this metric.

Number of Provided Interfaces (NPI) Very similar but not as important as NRI is the number of provided interfaces I_p. This is similarly a structural complexity measure that expresses the usage of a class by other entities in the system.

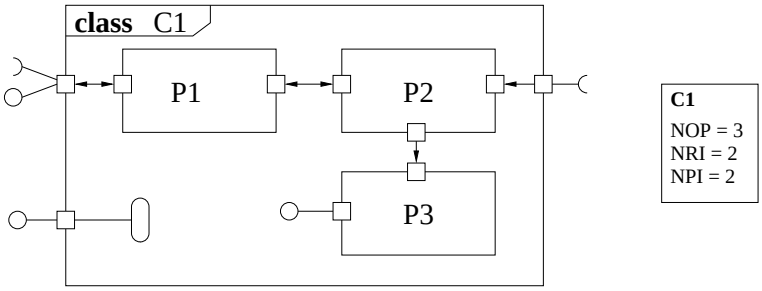

Fig. 3. An example structured class with three parts and the corresponding metrics.

State Machines. State machines are used to describe the behavior of classes of a system. They describe the actions and state changes based on a partitioning of the state space of the class. Therefore the associated state machine is also an indicator of the complexity of a class and hence its fault-proneness. State machines

consist of states and transitions where states can be hierarchical. Transitions carry event triggers, guard conditions, and actions.

We use cyclomatic complexity [McC76] to measure the complexity of behavioral models represented as state machines because it fits most naturally to these models as well as to code. This makes the lifting of the concepts from code to model straightforward. The basic concept is to transfer the metric from the realization of the state machine in code to the graphical representation.

To find the cyclomatic complexity of a state machine we build a control flow graph similar to the one for a program in [McC76]. This is a digraph that represents the flow of control in a piece of software. For source code, a vertex is added for each statement in the program and arcs if there is a change in control, e.g. an if- or while-statement. This can be adjusted to state machines by considering its code implementation. For a possible code transformation of state machines see [SGW94].

An example of a state machine and its control flow graph is depicted in Fig. 4. At first we need an entry point as the first vertex. The second vertex starts the loop over the automata because we need to loop until the final state is reached or infinitely if there is no final state. The next vertices represent transitions, atomic expressions[2] of guard conditions, and event triggers of transitions. These vertices have two outgoing arcs each because of the two possibilities of the control flow, i.e. an evaluation to *true* or *false*. Such a branching flow is always joined in an additional vertex. The last vertex goes back to the loop vertex from the start and the loop vertex has an additional arc to one vertex at the end that represents the end of the loop. This vertex finally has an arc to the last vertex, the exit point.

If we have such a graph we can calculate the cyclomatic complexity using the formula $v(G) = e - n + 2$, where v is the complexity, G the control graph, e the number of arcs, and n the number of vertices (nodes). There is also an alternative formula, $v(G) = p+1$, which can also be used, where p is the number of *predicate nodes*. Predicate nodes are vertices where the flow of control branches.

Hierarchical states in state machines are not incorporated in the metric. Therefore the state machine must be transformed into an equivalent state machine with simple states. This appears to be preferable to viewing substates as a kind of subroutines and keeping them out of the complexity calculation, because this would lose a considerable amount of information on the complexity. Furthermore internal transitions are counted equally to normal transitions. Pseudo states are not counted themselves, but their triggers and guard conditions.

Cyclomatic Complexity of State machine (CCS). Having explained the concepts based on the example flow graph above, the metric can be calculated directly from the state machine with a simplified complexity calculation. We count the atomic expressions and event triggers for each transition. Furthermore we need

[2] A guard condition can consist of several boolean expressions that are connected by conjunctions and disjunctions. An atomic expression is an expression only using other logical operators such as equivalence. For a more thorough definition see [McC76].

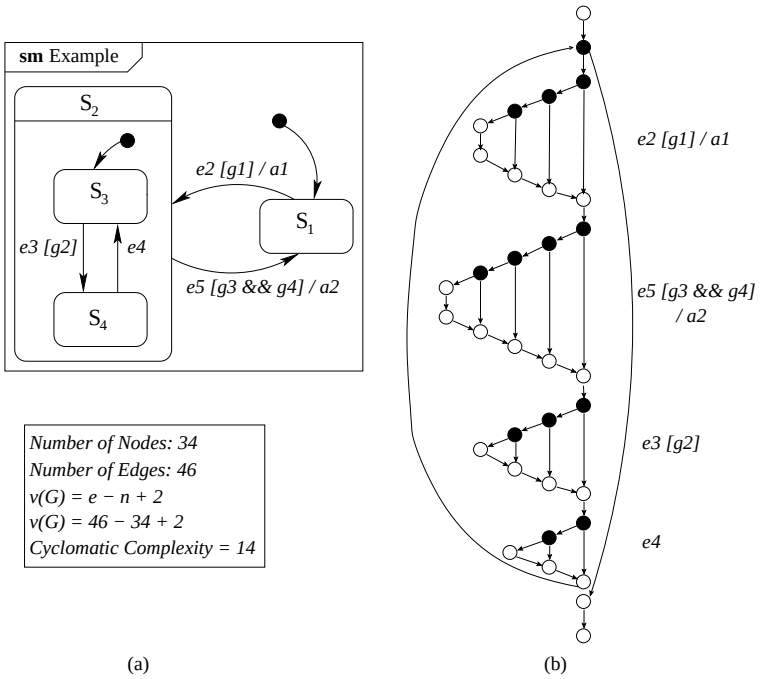

Fig. 4. (a) A simple state machine with one hierarchical state, event trigger, guard conditions, and actions. (b) Its corresponding control flow graph. The black vertices are predicate nodes. On the right the transitions for the respective part of the flowgraph are noted.

to add 1 for each transition because we have the implicit condition that the corresponding source state is active. This results in the formula $CCS = |T| + |E| + |A_G| + 2$, where T is the multi-set of transitions, E is the multi-set of event triggers, and A_G is the multi-set of atomic expressions in the guard conditions. This formula yields exactly the same results as the longer version above but has the advantage that it is easier to calculate.

For this metric we have to consider two abstraction layers. First, we transform the state machine into its code representation[3] and afterwards use the control flow graph of the code representation to measure structural complexity. The first "abstraction" is needed to establish the relationship to the corresponding code complexity. The code complexity is a good indicator of the fault-proneness of a program. The proposition is that the state machine reflects the major complexity attributes of the code that implements it. The next abstraction to the control flow graph was established in [McC76].

[3] Note that this is done only for measuring purposes; our approach also applies if the actual implementation is not automatically generated from the UML model but manually implemented.

In [HS90] the correlation of metrics of design specifications and code metrics was analyzed. One of the main results was that the code metrics such as the cyclomatic complexity are strongly dependent on the level of refinement of the specification, i.e. the metric as a lower value the more the specification is abstract. This also holds for the CCS metric. Models of software can be based on various different abstractions, such as functional or temporal abstractions [PP04]. Depending on the abstractions chosen for the model, various aspects may be omitted, which may have an effect on the metric. Therefore, it is prudent to consider a suite of metrics rather than a single metric when measuring design complexity to assess fault-proneness of system components.

In addition to the metrics which we defined above, we will now complete our metrics suite by adding two existing metrics from the literature.

Metrics Suite. Three of the metrics from [CK94] can be adjusted to be applicable to UML models. The metrics chosen are the ones that were found to be good indicators of fault-prone classes in [BBM96]. However, we omit *Response For a Class (RFC)* and *Coupling Between Objects (CBO)* because they cannot be determined on the model level. The remaining two metrics together with the new ones developed above form our metrics suite in Tab. 4. We now describe these two adapted metrics.

Depth of Inheritance Tree (DIT) This is the maximum depth of the inheritance graph T to a class c. This can be determined in any class diagram that includes inheritance.

Number of Children (NOC) This is the number of direct descendants C_d in the inheritance graph. This can again be counted in a class diagram.

Name	Abbr.	Calculation
Depth of Inheritance Tree	DIT	$max(depth(T, c))$
Number of Children	NOC	$\|C_d\|$
Number of Parts	NOP	$\|C_p\|$
Number of Required Interfaces	NRI	$\|I_r\|$
Number of Provided Interfaces	NPI	$\|I_p\|$
Cyclomatic Complexity of State machine	CCS	$\|T\| + \|E\| + \|A_G\| + 2$

Table 4. A summary of the metrics suite with its calculation

We consider whether our metrics are *structural complexity measure* by the definition in [MGBB90]. The definition says that for a set D of documents with a pre-order $\leq_D$ and the usual ordering $\leq_{\mathbb{R}}$ on the real numbers $\mathbb{R}$, a structural complexity measure is an order preserving function $m : (D, \leq_D) \longrightarrow (\mathbb{R}, \leq_{\mathbb{R}})$. Each metric from the suite fulfills this definition with respect to a suitable pre-order on the relevant set of documents. The document set D under consideration is depending on the metric: either a class diagram that shows inheritance and

possibly interfaces, a composite structure diagram showing parts and possibly interfaces, or a state machine diagram. All the metrics use specific model elements in these diagrams as a measure. Therefore there is a pre-order $\leq_D$ between the documents of each type based on the metrics: We define $d_1 \leq_D d_2$ for two diagrams d_1, d_2 in D if d_1 has fewer of the model elements specific to the metric under consideration than d_2. The mapping function m maps a diagram to its metric, which is the number of these elements. Hence m is order preserving and the metrics in the suite qualify as structural complexity measures.

As mentioned before, complexity metrics are good predictors for the reliability of components [KW90,MK96]. Furthermore the experiments in [BBM96] show that most metrics from [CK94] are good estimators of fault-proneness. We adopted DIT and NOC from these metrics unchanged, therefore this relationship still holds. The cyclomatic complexity is also a good indicator for reliability [KW90] and this concept is used for CCS to be able to keep this relationship. The remaining three metrics were modeled similarly to existing metrics. NOP resembles NOC, NRI and NPI are similar to CBO. NOC and CBO are estimators for fault-proneness, therefore it is expected that the new metrics behave accordingly.

This metrics suite can now be used to determine the most fault-prone classes and components in a system. However, different metrics are important for different components. Therefore one cannot just take the sum over all metrics to find the most critical component. Some component models may have an associated state machine, others not. This makes the sum meaningless. We propose to use the metrics so that we compute the metric values for each component and class and consider the ones that have the highest measures for each single metric. This way we can for example determine the components with complex behavior, or high fan-in and fan-out.

3.2 Failure Proneness

We pointed out already that the fault-proneness of a component does not directly imply low reliability because a high number of faults does not mean that there is a high number of failures [Wag04a,Wag04b]. However a direct reliability measurement is in general not possible on the model level. Nevertheless we can get close by analysing the failure-proneness of a component, i.e. the probability that a fault leads to a failure that occurs during software execution.

It is not possible to express the probability of failures with exact figures based on the design models. We propose therefore to use more coarse-grained *failure levels*, e.g. $L_F = \{high, medium, low\}$, where L_F is the set of failure levels. This allows an abstract assessment of the failure probability. It is still not reliability as generally defined but the best estimate that we can get in early phases.

To determine the failure level of a component we use the metrics suite from above to define *complexity levels* $L_C = \{high, low\}$. We assign each component such a complexity level by looking at the extreme values in the metrics results. Each component that exhibits a high value in at least one of the metrics is considered of having the complexity level *high*, all other components have the

level *low*. It depends on the actual distribution of values to determine what is to be considered a high value.

Having assigned these complexity levels to the components, we know which components are highly fault-prone. The operational profile [Mus99] is a description of the usage of the system, showing which functions are mostly used. We use this information to assign *usage levels* L_U to the components. This can be of various granularity. An example would be $L_U = \{high, medium, low\}$. When we know the usage of each component we can analyze the probability that the faults in the component lead to a failure.

The combination of complexity level and usage level leads us to the *failure level* L_F of the component. It expresses the probability that the component fails during software execution. We describe the mapping of the complexity level and usage level to the failure level with the function *fp*:

$$fp = L_C \times L_U \longrightarrow L_F, \text{where } L_F = L_U \cup \{low\}$$

What the function does is simply to map all components with a high complexity level to its usage level and all component with a low complexity level to *low*.

$$fp(x, y) = \begin{cases} y & \text{if } x = high \\ low & \text{otherwise} \end{cases}$$

This means that a component with high fault-proneness has a failure probability that depends on its usage and a component with low fault-proneness has generally a low failure probability.

Having these failure levels for each component we can use that information to guide the verification efforts in the project, e.g. assign the most amount of inspection and testing on the components with a high failure level.

3.3 Checking of Reliability Requirements

In this section, we sketch exemplarily how to use the information on fault-proneness of components obtained using the metrics in the previous section in combination with the checks of reliability requirements considered in Sect. 2. More specifically, we explain how to relate the levels derived from an UML model and the operational profile to reliability requirements formulated in the UML diagram using UML stereotypes.

Critical As defined in Sect. 2, the stereotype «critical» labels classes whose instances are critical in some way, as specified by the associated tags {level} for each reliability level *level* $\in$ *Levels*. The intention is now that for the failure level $f \in L_F$ defined in Sect. 3.2 for any reliability level $l \in$ *Levels*, and for any component C stereotyped with this level «l», if the levels l and f are contradictory, C should be more closely inspected for possible flaws (for example, using a formal verification, which in general would be too costly to apply to the whole system). Contradictory means here a failure level and a reliability level that are not compatible, e.g. a high failure level and a high reliability level.

Containment We use the stereotype «containment» of subsystems defined in Sect. 2 to detect system parts with a high failure level which may influence data values that are supposed to be highly reliable These system parts can then be inspected more thoroughly for possible flaws.

4 Tool Support for Model-Based Reliability and Safety Analysis

To support our approach, we developed automated tools for the formal verification of UML models for the constraints associated with the stereotypes introduced in the Secs. 2 and 3. We describe a framework that incorporates several formal verification tools (including the model-checker Spin and the automated theorem prover e-SETHEO).

Functionality There are three consecutive stages in implementing full verification functionality for the formalized UML models. There exist verification tools for all stages. The framework is, however, designed to be extensible, so new analysis plug-ins can be added easily.

- *Static features.* Checkers for static features (for example, a type-checking like enforcement of safety levels in class and deployment diagrams) have been implemented directly.
- *Simple dynamic features.* Checks of UML models of a bounded size for simple dynamic properties (for example, that a deterministic Machine without interaction with the environment does not reach a certain critical state) can still be directly implemented.
- *Complex dynamic features.* Checks for complicated behavioral properties or of large, or highly non-deterministic or interactive UML models require the use of sophisticated tools (such as model-checkers). This is implemented by translating the required UML constructs into the model-checker input language (for example, a Temporal Logic formula).

To be able to apply sophisticated tools (such as model checkers) to compute a metric, one needs a front-end which automatically produces a semantic model and includes the relevant formalized safety requirements, when given a UML model. This avoids requiring the software developers themselves to perform this formalization, which usually needs a high level of specialized training in formal methods. UML supports this approach by offering predefined safety primitives (such as safety requirements or mechanisms) with a strictly defined semantics, which can be applied by a developer without an extensive training in safety-critical systems by simply including the relevant stereotypes in the UML model. These primitives are translated into the targeted formal language, protecting from potential errors in manual formalization of the safety properties. Since safety requirements are usually defined relative to failure model, to analyse whether the UML specification fulfills a safety requirement, the tool-support

has to automatically include the failure model arising from the physical view contained in the UML specification.

We briefly describe the functionality of the UML tool that meets the listed requirements. The developer creates a model and stores it in the UML 1.5 / XMI 1.2 file format.[4] The file is imported by the tool into the internal MDR repository. The tool accesses the model through the JMI interfaces generated by the MDR library. The checker parses the model and checks the constraints associated with the stereotype. The results are delivered as a text report for the developer describing found problems, and a modified UML model, where the stereotypes whose constraints are violated are highlighted. The tool can be executed as a console application, as a web-application, or a GUI application.

5 Case-Study: Automatic Collision Notification

In this part of the chapter, we validate our proposed safety and reliability analyzes in a case study of an automatic collision notification system as used in cars to provide automatic emergency calls. First, the system is described and designed using the UML extension that we made in the previous sections, then we analyze the model and present the results.

Description This case study that we used to validate our results was done in cooperation with the automotive manufacturer BMW. There is a similar project currently in development. The problem to be solved is that many accidents of automobiles involve only a single vehicle. Therefore it is possible that no or only a delayed emergency call is made. The chances for the casualties is significantly higher if an accurate call is made quickly. This has lead to the development of so called *Automatic Collision Notification (ACN)* systems, sometimes also called *mayday* systems. They automatically notify an emergency call response center when a crash occurs but also manual notification using the location data from a GPS device can be made. We used the public specification from the *Enterprise* program [EP97,EP98] as a basis for the design model because the work together with BMW is in an early stage. In this case study, we will concentrate on the built-in device of the car and ignore the obviously necessary infrastructure such as the call center.

Device Design Following [EP97] we will call the built-in device *MaydayDevice* and divide it into five components. The architecture is illustrated in Fig. 5 using a composite structure diagram of the device.

The device is a processing unit that is built into the vehicle and has the ability to communicate with an emergency call center using a mobile telephone connection and retrieving position data using a GPS device. The components that constitute the mayday device are:

[4] The framework will be updated to UML 2.0 as soon as the official DTDs will be available.

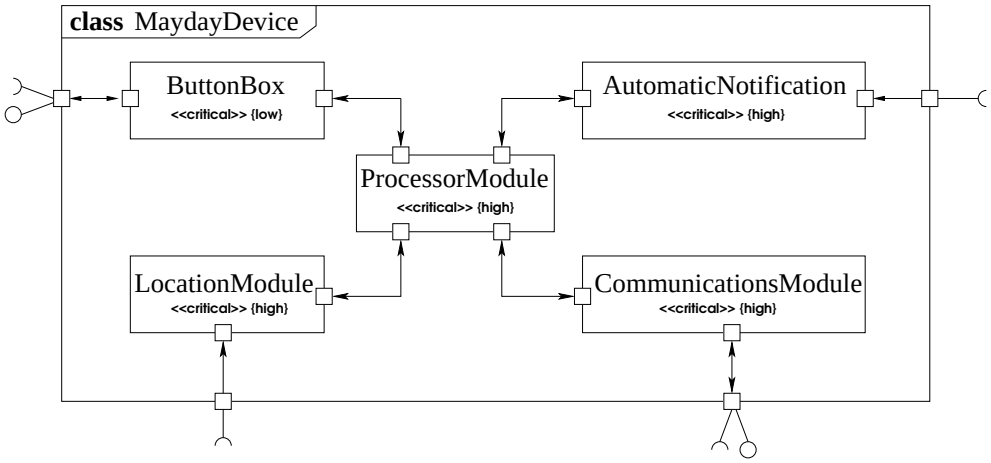

Fig. 5. The composite structure diagram of the mayday device.

- *ProcessorModule*: This is the central component of the device. It controls the other components, retrieves data from them and stores it if necessary.
- *AutomaticNotification*: This component is responsible for notifying a serious crash to the processor module. It gets notified itself if an airbag is activated.
- *LocationModule*: The processor module request the current position data from the location module, that gathers the data from a GPS device.
- *CommunicationsModule*: The communications module is called from the processor module to send the location information to an emergency call center. It uses a mobile communications device and is responsible for automatic retry if a connection did fail.
- *ButtonBox*: This is finally the user interface that can be used to manually initiate an emergency call. It also controls a display that provides feedback to the user.

These components are again shown in Fig. 6 in a class diagram showing the attributes and methods of each. It also shows that we have exactly one instance of each class in the system. Furthermore we used some tagged values based on Sect. 2 to describe safety requirements on some data values. The central *ProcessorModule* has the annotated requirements that the method *getGpsData* from the *LocationModule* delivers its data in realtime and correct, and that the data of the call is transfered correct by the method *makeCall* of *CommunicationsModule*. *LocationModule* and *CommunicationsModule* have the corresponding annotations, therefore are these requirements consistent, as defined in Sect. 2.2.

Each of the components of the mayday device has an associated state machine to describe its behavior. We do not show the state machines because of space reasons but they can be found in [WJ05].

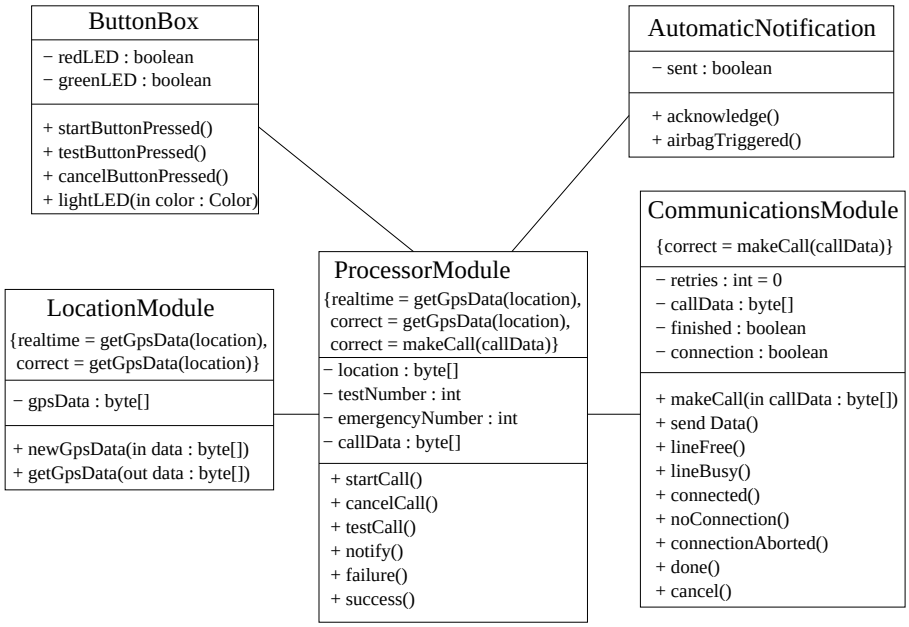

Fig. 6. The class diagram of the parts.

Results The five subcomponents of *MaydayDevice* are further analyzed in the following. At first we used our metrics suite from Sec. 3 to gather data about the model. The results can be found in Tab. 5. It shows that we have no inheritance in the current abstraction level of our model and also that the considered classes have no parts. Therefore the metrics regarding these aspects are not helpful for this analysis.

Class	DIT	NOC	NOP	NRI	NPI	CCS
ProcessorModule	0	0	0	4	4	16
AutomaticNotification	0	0	0	2	1	4
LocationModule	0	0	0	1	2	4
CommunicationsModule	0	0	0	2	2	32
ButtonBox	0	0	0	2	2	8

Table 5. The results of the metrics suite for the components of *MaydayDevice*.

More interesting are the metrics for the provided and required interfaces and their associated state machines. The class with the highest values for NRI and NPI is *ProcessorModule*. This shows that it has a high fan-in and fan-out and is therefore fault-prone. The same module has a high value for CCS but *CommunicationsModule* has a higher one and is also fault-prone. Therefore we

assign the complexity level *high* to these two components, the other have the level *low*.

The documentation in [EP98] shows that the main failures that occurred were failures in connecting to the call center (even when cellular strength was good), no voice connect to the call center, inability to clear the system after usage, and failures of the cancel function. These main failures can be attributed to the component *ProcessorModule* that is responsible for controlling the other components and *CommunicationsModule* that is responsible for the wireless communication. Therefore our reliability analysis labeled the correct components with a high failure level.

6 Case Study: MOST Network Master

We further validated our approach on the basis of the project results of an evaluation of model-based testing [PPW+05]. A network controller of an infotainment bus in the automotive domain, the MOST® Network Master [MOS04], was modeled with the case tool AutoFocus [HSSS96] and test cases were generated from that model and compared with traditional tests. We use all found faults from all test suites in the following but as we have mainly fault information, we concentrate on fault-proneness rather than failure-proneness. AutoFocus is quite similar to UML 2.0 and therefore the conversion was straight-forward. The composite structure diagram of the Network Master is shown in Figure 7.

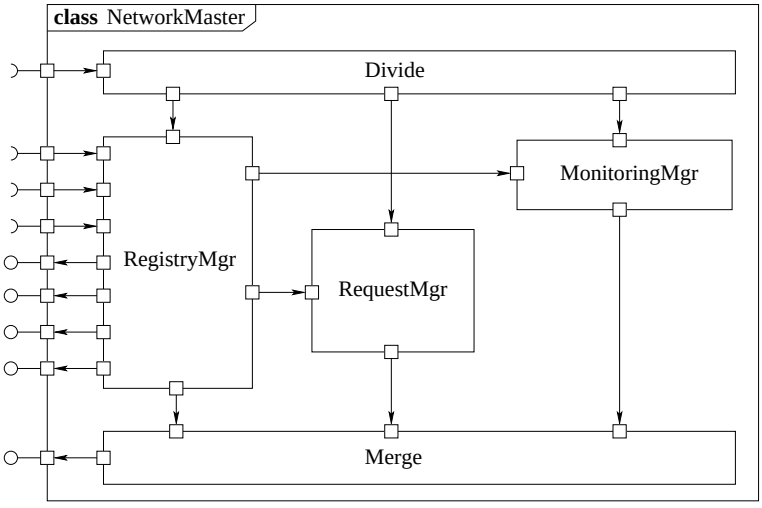

Fig. 7. The composite structure diagram of the MOST Network Master.

We omit further parts of the design, especially the associated state machines, because of space and confidentiality reasons. The corresponding metrics are summarized in Table 6.

Class	DIT	NOC	NOP	NRI	NPI	CCS
NetworkMaster	0	0	5	4	4	0
Divide	0	0	0	1	3	11
Merge	0	0	0	3	1	8
MonitoringMgr	0	0	0	2	1	0
RequestMgr	0	0	0	2	1	14
RegistryMgr	0	0	0	4	7	197

Table 6. The results of the metrics suite for the NetworkMaster.

The data from the table shows that the *RegistryMgr* has the highest complexity in most of the metrics. Therefore we classify it as being highly fault-prone. As described in [PPW+05] several test suites were executed against an implementation of the Network Master. 24 faults have been identified by the test activities. 21 of which can be attributed to the *RegistryMgr*, 3 to the *RequestMgr*. There were no revealed faults in the other components. Hence, the high fault-proneness of the *RegistryMgr* did indeed result in a high number of faults revealed during testing.

7 Related work

In the related area of real-time systems there has been a substantial amount of work regarding the usage of UML. For example, [SR98] describes constructs to facilitate the design of software architectures in this domain which are specified using UML. [JCF+02,JRFF03,JRFF04] contain several approaches to developing systems with various criticality requirements using UML. In particular, [HG02] discusses a pattern-based approach for using UML use cases for safety-critical systems. The focus is on the development of a testing strategy rather than model analysis. [PMP01] discusses methods and tools for the checking of UML statechart specifications of embedded controllers. The focus is on the use of statecharts and on efficient methods for automated checking and does not include the use of other UML diagrams or the inclusion of safety requirements using stereotypes. Also relevant is the work towards a formal semantics of UML (see the proceedings of related conferences, including the UML and FASE conferences).

[LM01] proposes the automated generation of fault trees based on the source code of software which may be combined with fault trees based on the electronic circuit design of the hardware, allowing the software and the hardware fault trees to be composed into a fault tree of the system. It presents a prototype of a fault tree generation tool that is capable to generate fault trees based on C++-code. [BDCL+01] presents an integrated tool environment where automatic transformations of UML models can capture dependability requirements.

The proposed metrics suite as a basis to find fault-prone components can also be found in [WJ05]. Lano et al. [LCA02] propose a method to analyze object-oriented models in terms of safety and security but not considering complexity directly. In [WWC99] an approach is proposed that includes a reliability model that is based on the software architecture. A complexity metric that is in principle applicable to models as well as to code is discussed in [CA88], but it also only involves static structure. Another approach related to safety-critical systems is proposed in [Jür03b]. It annotates UML models with safety-related information for further analysis. In [BHS97] the cyclomatic complexity is suggested for most aspects of a design metric. Safety checklists have been proposed for example in [HJL96]. [HH99] uses Z and Petri nets for modeling safety-critical systems.

8 Conclusions

In this chapter we propose means to incorporate reliability requirements into UML models. It is achieved using the UML profile mechanisms of stereotypes and tagged values. This makes these important requirements visible in the model, helps to encapsulate knowledge of reliability mechanisms, and simplifies its use by non-experts. Furthermore by formalizing the requirements, checks can be done.

Having annotated a model with the defined stereotypes and tagged values, one can check the consistency of the requirements through-out the model. This lends itself to tool support for automatic checking. We describe a framework in which several of such checks are implemented.

To provide a reasonable basis for the reliability analysis of a system, we also present a metrics suite for UML models based on the work of [CK94] to measure the structural complexity of the models. Specifically, we use the numbers of provided and required interfaces as a metric for fan-in and fan-out, and lift the cyclomatic complexity [McC76] to the machine level to measure the complexity of state machines. The suite is then used to find fault-prone components in a system.

Fault-proneness, i.e. the probability of containing faults, is not a good measure for reliability because it does not take into account the probability of the faults of leading to a failure. Therefore the operational profile [MIO87,Mus99] is used to estimate the usage of a component and the combination of fault-proneness and usage yields the failure-proneness of the component. This information can finally be used to check consistency with earlier defined reliability requirements in the model and to improve the efficiency of verification efforts.

We finish our chapter with two case studies. One of these describes an *Automatic Collision Notification* system for automobiles that sends automatically an emergency call in case of a crash. It shows that the metrics suite, especially in combination with an operational profile, indeed can identify failure-prone components. Furthermore, the case study illustrates the interplay of the metrics suite with the annotation with stereotypes. The second case study investigates only

the capabilities of the metrics suite. It confirms that the suite helps to identify fault-prone components.

Acknowledgments

We gratefully acknowledge the joint work with Martin Baumgartner, Christian Kühnel, Alexander Pretschner, Wolfgang Prenninger, Bernd Sostawa, and Rüdiger Zölch on the MOST Network Master. Furthermore we are grateful to Manfred Broy and Wolfgang Prenninger for commenting on a draft version. This work was partly sponsored by the DFG within the project *InTime* and the German Ministry for Science and Education within the *Verisoft* project.

References

[ABW03] C. Atkinson, C. Bunse, and J. Wüst. Driving component-based software development through quality modelling. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality*, volume 2693 of *LNCS*, pages 207 – 224. Springer, 2003.

[BBM96] V.R. Basili, L.C. Briand, and W.L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[BDCL+01] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *Journal of Computer Systems Science and Engineering*, 16:265–275, 2001.

[BHS97] J.K. Blundell, M.L. Hines, and J. Stach. The Measurement of Software Design Quality. *Annals of Software Engineering*, 4:235–255, 1997.

[CA88] D.N. Card and W.W. Agresti. Measuring Software Design Complexity. *The Journal of Systems and Software*, 8:185–197, 1988.

[CK94] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[DS99] B. Dutertre and V. Stavridou. A model of noninterference for integrating mixed-criticality software components. In *DCCA*, San Jose, CA, January 1999.

[EP97] Mayday: System Specifications. The ENTERPRISE Program, 1997. Available at http://enterprise.prog.org/completed/ftp/mayday-spe.pdf (October 2004).

[EP98] Colorado Mayday Final Report. The ENTERPRISE Program, 1998. Available at http://enterprise.prog.org/completed/ftp/maydayreport.pdf (October 2004).

[FP97] N.E. Fenton and S.L. Pfleeger. *Software Metrics. A Rigorous & Practical Approach*. International Thomson Publishing, 2nd edition, 1997.

[Hal77] M.H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[HG02] K. Hansen and I. Gullesen. Utilizing UML and patterns for safety critical systems. In Jürjens et al. [JCF+02], pages 147–154.

[HH99] M. Heiner and M. Heisel. Modeling safety-critical systems with Z and Petri Nets. In M. Felici, K. Kanoun, and A. Pasquini, editors, *18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'99)*, volume 1698, pages 361–374, 1999.

[HJL96] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Software Eng. and Methodology*, 5(3):231–261, July 1996.

[HK81] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Trans. Software Engineering*, 7:510–518, 1981.

[HS90] S. Henry and C. Selig. Predicting Source-Code Complexity at the Design Stage. *IEEE Software*, 7:36–44, 1990.

[HSSS96] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus: A tool for distributed systems specification. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96*, volume 1135 of *LNCS*, pages 467–470, Uppsala, Sweden, Sept. 9–13 1996. Springer.

[JCF+02] J. Jürjens, V. Cengarle, E.B. Fernandez, B. Rumpe, and R. Sandner, editors. *Critical Systems Development with UML*, number TUM-I0208 in TU München Technical Report, 2002. UML'02 satellite workshop proceedings.

[JRFF03] J. Jürjens, B. Rumpe, R. France, and E.B. Fernandez, editors. *Critical Systems Development with UML*, number TUM-I0317 in TU München Technical Report, 2003. UML'03 satellite workshop proceedings.

[JRFF04] J. Jürjens, B. Rumpe, R. France, and E.B. Fernandez, editors. *Third International Workshop on Critical Systems Development with UML*, TU München Technical Report, 2004. UML'04 satellite workshop proceedings.

[Jür03a] J. Jürjens. Critical systems development with UML and model-based testing. In *The 22st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, Sept. 23-26 2003. Full-day tutorial.

[Jür03b] J. Jürjens. Developing safety-critical systems with UML. In P. Stevens, editor, *UML 2003 – The Unified Modeling Language*, volume 2863 of *LNCS*, pages 360–372, San Francisco, CA, October 20–24, 2003. Springer.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[KW90] T.M. Khoshgoftaar and T.G. Woodcock. Predicting Software Development Errors Using Software Complexity Metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[LCA02] K. Lano, D. Clark, and K. Androutsopoulos. Safety and Security Analysis of Object-Oriented Models. In *SAFECOMP 2002*, volume 2434 of *LNCS*, pages 82–93. Springer, 2002.

[LM01] P. Liggesmeyer and O. Maeckel. Quantifying the reliability of embedded systems by automated analysis. In *2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pages 89–96. IEEE Computer Society, 2001.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Trans. Software Engineering*, 5:45–50, 1976.

[MGBB90] A. Melton, D. Gustafson, J. Bieman, and A. Baker. A Mathematical Perspective for Software Measures Research. *IEE/BCS Software Engineering Journal*, 5:246–254, 1990.

[MH99] T. Mayer and T. Hall. A Critical Analysis of Current OO Design Metrics. *Software Quality Journal*, 8:97–110, 1999.

[MIO87] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[MK96] J.C. Munson and T.M. Khoshgoftaar. Software Metrics for Reliability Assessment. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 12. IEEE Computer Society Press and McGraw-Hill, 1996.

[MOS04] MOST Cooperation. MOST Media Oriented System Transport—Multimedia and Control Networking Technology. MOST Specification Rev. 2.3. August 2004.

[Mus99] J.D. Musa. *Software Reliability Engineering.* McGraw-Hill, 1999.

[Obj03] Object Management Group. UML 2.0 Superstructure Final Adopted specification, August 2003. OMG Document ptc/03-08-02.

[PMP01] Z. Pap, I. Majzik, and A. Pataricza. Checking general safety criteria on UML statecharts. In U. Voges, editor, *SAFECOMP 2001*, volume 2187 of *LNCS*, pages 46–55. Springer, 2001.

[PP04] W. Prenninger and A. Pretschner. Abstractions for Model-Based Testing. In M. Pezze, editor, *Proc. Test and Analysis of Component-based Systems (TACoS'04)*, 2004.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. An Evaluation of Model-Based Testing and its Automation. In *Proc. 27th International Conference on Software Engineering (ICSE)*, 2005. To appear.

[Ran00] F. Randimbivololona. Orientations in verification engineering of avionics software. In R. Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, LNCS, pages 131–137. Springer, 2000.

[RHS98] L. Rosenberg, T. Hammer, and J. Shaw. Software Metrics and Reliability. In *Proc. 9th International Symposium on Software Reliability Engineering (ISSRE'98)*. IEEE, 1998.

[Rus94] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[Sel02] B. Selic. Physical programming: Beyond mere logic. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software Second International Conference (EMSOFT 2002)*, volume 2491 of *LNCS*, pages 399–406, 2002.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Available at http://www-106.ibm.com/developerworks/rational/library/, 1998.

[Wag04a] S. Wagner. Efficiency Analysis of Defect-Detection Techniques. Technical Report TUMI-0413, Institut für Informatik, Technische Universität München, 2004.

[Wag04b] S. Wagner. Reliability Efficiency of Defect-Detection Techniques: A Field Study. In *Suppl. Proc. 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, 2004.

[WJ05] S. Wagner and J. Jürjens. Model-Based Identification of Fault-Prone Components. In *Proc. The Fifth European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 435–452. Springer, 2005.

[WWC99] W.-L. Wang, Y. Wu, and M.-H. Chen. An Architecture-Based Software Reliability Model. In *Proc. Pacific Rim International Symposium on Dependable Computing (PRDC'99)*, pages 143–150, 1999.