Model-Based Identification of Fault-Prone Components*

Stefan Wagner and Jan Jürjens

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany
http://www4.in.tum.de/~{wagnerst,juerjens}

Abstract. The validation and verification of software is typically a costly part of the development. A possibility to reduce costs is to concentrate these activities on the fault-prone components of the system. A classification approach is proposed that identifies these components based on detailed UML models. For this mainly existing code metrics are tailored to be applicable to models and are combined to a suite. Two industrial case studies confirm the ability of the approach to identify fault-prone components.

1 Introduction

The whole area of testing and quality assurance constitutes a significant part of the total development costs for software, often up to 50% [1]. Especially formal verification is frequently perceived as rather costly. Therefore there is a possibility for optimizing costs by concentrating on the fault-prone components and thereby exploiting the existing resources as efficiently as possible. Detailed design models offer the possibility to analyse the system early in the development life-cycle. One of the possibilities is to measure the complexity of the models to predict fault-proneness assuming that a high complexity leads to a high number of defects.

The complexity of software code has been studied to a large extent. It is often stated that complexity is related to and a good indicator for the fault-proneness of software [2–4]. There are two different approaches to the identification of fault-prone components. In the *estimative approach* models are used to predict the number of faults that are contained in each component. The *classification approach* categorizes components into fault-prone classes, often simply low-fault and high-fault. We use the latter approach in the following because it is more suitable for the model metrics.

Although the traditional complexity metrics are not directly applicable to design models because of different means of structuring and abstractions, there

* This research was supported in part by the *Deutsche Forschungsgemeinschaft (DFG)* within the project *InTime* and by the German Ministry for Science and Education within the *Verisoft* project.

are already a number of approaches that propose design metrics, e.g. [5–8]. Most of the metrics in [8] were found to be good estimators of fault-prone classes in [9] and are used in our approach as well. However, they concentrate mainly on the structure of the designs. Since the system structure is not sufficient as a source for the complexity of its components, which largely depends on their behavior, we will also propose a metric for behavioral models.

Contribution. This paper contains an adaption of complexity metrics to measure design complexity of UML 2.0 models. Based on these metrics an approach is proposed for deriving the fault-proneness of classes. Furthermore the metrics and the approach are validated by two industrial case studies.

Outline. In Sec. 2 complexity metrics for models built with a subset of UML 2.0 are defined and an approach for using the metrics to derive fault- and failure-prone components is explained. Two case studies are provided in Sec. 3. Finally, related work and conclusions are discussed in Sec. 4 and Sec. 5, respectively.

2 Analyzing Fault-Proneness

This section describes the possibilities to identify fault-prone components based on models built with UML 2.0 [10]. We introduce a design complexity metrics suite for a subset of model elements of the UML 2.0 and explain how to identify fault-prone components.

The basis of our metrics suite forms the suite from [8] for object-oriented code and the cyclomatic metric from [11]. In using a suite of metrics we follow [12, 13] stating that a single measure is usually inappropriate to measure complexity.

In [14] 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 has a lower value the more abstract the specification is. Models of software can be based on various different abstractions, such as functional or temporal abstractions [15]. Depending on the abstractions chosen for the model, various aspects may be omitted, which may have an effect on the metrics. 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.

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.

We adjust metrics to parts of UML 2.0 based on the design approach taken in AutoFocus [16], ROOM [17], or UML-RT [18], 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 and which have associated state machines that describe their 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. The particular usage should nevertheless be determined by the actual development process.

2.1 Measures of the Static Structure

We start introducing the new measures with the ones that analyze the static structure of models. These are important because the interrelations and dependencies among model elements contribute significantly to their complexity.

Structured Classes. The concept of structured classes introduces composite structures that represent a composition of run-time instances collaborating over communication links. This allows UML classes to have an internal structure consisting of other classes that are bound by connectors. Furthermore ports are used as a defined entry point to a class. A port can group various interfaces that are provided or required. A connection between two classes through ports can also be denoted by a connector. The parts of a class work together to achieve its behavior. A state machine can also be defined to describe behavior additional to the behavior provided by the parts.

We start with three metrics, Number of Parts, Number of Required Interfaces, and Number of Provided Interfaces, which concern structural aspects of a system model. The metrics consider composite structure diagrams of single classes with their parts, interfaces, connectors, and possibly state machines. A corresponding example is given in Fig. 1.

Number of Parts (NOP). The number of parts of a structured class contributes obviously to its structural complexity. The more parts it has, the more coordination is necessary and the more dependencies there are, all of which may contribute to a fault. Therefore, we define *NOP* as the number of direct parts C_p of a class.

Number of Required Interfaces (NRI). This metric is (together with the NPI metric below) a substitute for the old *Coupling Between Objects (CBO)* that was criticized in [19] 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 a fault, for example if the interfaces are not correctly defined. Therefore we count the number of required interfaces I_r for this metric. Coupling metric as predictors of run-time failures were investigated in [20]. It shows that coupling metrics are suitable predictors of failures.

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.

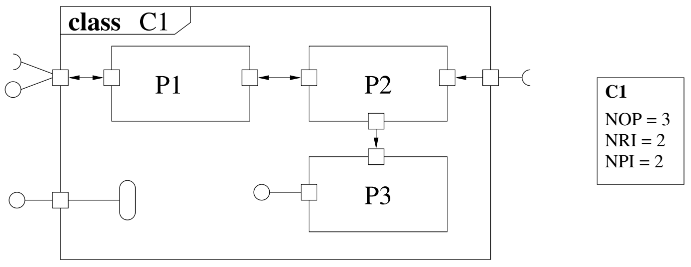

Fig. 1. An example structured class with three parts and the corresponding metrics.

Example. The example in Figure 1 shows the composite structure diagram of a class with three ports, two required and two provided interfaces. It has three parts which have in turn ports, interfaces and connectors. However, these connecting elements are not counted in the metrics for the class itself because they are counted by the metrics for the parts, and these can later be summed up to consider the complexity of a class including its parts.

2.2 Measure of Behavior

We proceed with a complexity metric for behavioral models because the behavior determines the complexity of a component to a large extent.

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 [11] 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.

To find the cyclomatic complexity of a state machine we build a control flow graph similar to the one for a program in [11]. 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 the code implementation. The code transformation that we use as a basis for the metrics can be found in [17]. However, different implementation strategies could be used [21].

Example. An example of a state machine and its control flow graph is depicted in Fig. 2. 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[1] 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 binary *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 handling hierarchy separately because we are not looking at understandability and we do not have to deal with hierarchy crossing transitions. Furthermore internal transitions are counted equally to normal transitions. Pseudo states are not counted themselves, but their triggers and guard conditions. Usage of the *InState* construct in guards is not considered.

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 to add 1 for each transition because we have the implicit condition that the corresponding source state is active. This results in the formula

$$CCS = |T| + |E| + |A_G| + 2. \tag{1}$$

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.

[1] 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 [11].

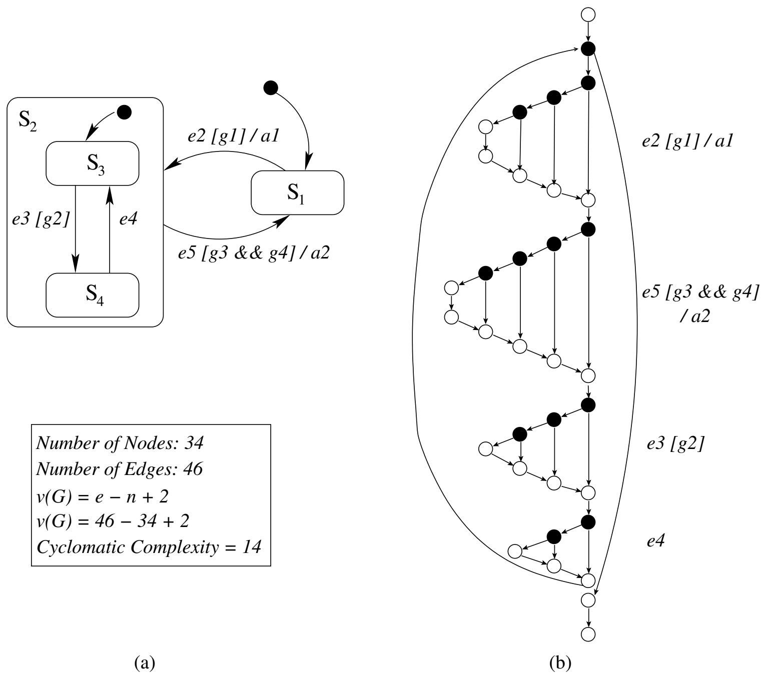

Fig. 2. (a) A simple state machine with one hierarchical state, event trigger, guard conditions, and actions. (b) Its corresponding control flow graph. The black vertices are predicate nodes. On the right the transitions for the respective part of the flowgraph are noted.

For this metric we have to consider two abstraction layers. First, we transform the state machine into its code representation[2] and second 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 because it 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 second abstraction to the control flow graph was established in [11] and is needed for the determination of paths through the program which reflect the complexity of the behavior.

2.3 Metrics Suite

In addition to the metrics which we defined above, we complete our metrics suite by adding two existing metrics from [8] that can be adjusted to be applicable to UML models. The metrics chosen are from the ones that were found to be good

[2] 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.

indicators of fault-prone classes in [9]. We omit *Response For a Class (RFC)* and *Coupling Between Objects (CBO)*[3] because they cannot be determined on the model level. The two adapted metrics are described in the following. The complete metrics suite can be found in Tab. 1.

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.

Table 1. A summary of the metrics suite with its calculation

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	$\|T\| + \|E\| + \|A_G\| + 2$

We analyze whether our metrics are *structural complexity measures* by the definition in [12]. 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 $m : (D, \leq_D) \longrightarrow (\mathbb{R}, \leq_\mathbb{R})$. This means that any structural complexity metric needs to be at least pre-ordered because this is necessary for comparing different documents. 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.

Fault Proneness. As mentioned before, complexity metrics are good predictors for the reliability of components [2, 3]. Furthermore the experiments in [9] show

[3] RFC counts all methods of a class and all methods recursively called by the methods. CBO counts all references of a class to methods or fields of other classes.

that most metrics from [8] 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 [2] 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.

The metrics suite is used to determine the most fault-prone classes in a system. Different metrics are important for different components. Therefore one cannot just take the sum over all metrics to find the most critical component. 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 coupling.

We suggest to use *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. These complexity levels show the high-fault and low-fault components.

Failure Proneness. The following constitutes an extension to the analysis of fault proneness towards failure proneness. 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 [22]. 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 complexity levels from above. Having assigned these complexity levels to the components, we know which components are highly fault-prone. The operational profile [23] 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*:

$$fp = L_C \times L_U \longrightarrow L_F, \text{where } L_F = L_U \cup \{low\} \tag{2}$$

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*. However, this is only one possibility how *fp* can look like.

$$fp(x, y) = \begin{cases} y & \text{if } x = high \\ low & \text{otherwise} \end{cases} \tag{3}$$

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. Parts of critical systems such as an exception handler still need thorough testing although its failure level might be low. However, this is not part of this work.

3 Case Studies

This section presents two industrial case studies that use the classification approach based on the metrics suite and contains a discussion of the results and observations. Both case studies do not analyze the DIT and NOC metrics because the models do not contain inheritance.

3.1 Automatic Collision Notification

The first case study we used to validate our proposed fault-proneness analysis is an automatic collision notification system as used in cars to provide automatic emergency calls. First, the system is described and designed using UML, then we analyze the model and present the results.

Description. The case study was done in cooperation with a car manufacturer. The problem to be solved is that many accidents of automobiles only involve a single vehicle. Therefore it is possible that no or only a delayed emergency call is made. The chances for successful help for the casualties are 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. In addition, manual notification using the location data from a GPS device can be made. We used the public specification from the *Enterprise* program [24, 25] as a basis for the design model. Details of the implementation technology are not available. In this case study, we concentrate on the built-in device of the car and ignore the obviously necessary infrastructure such as the call center.

Device Design. Following [24] we call the built-in device *MaydayDevice* and divide it into five components. The architecture is illustrated in Fig. 3 using a composite structure diagram of the device.

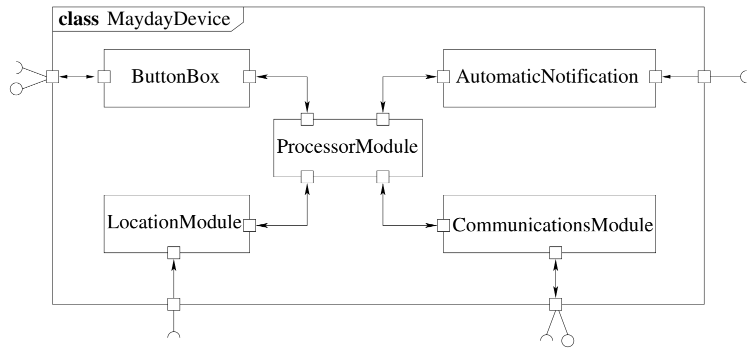

Fig. 3. The composite structure diagram of the mayday device.

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:

- *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 fails.
- *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.

Each of the components of the mayday device has an associated state machine to describe its behavior. We do not show all of the state machines because of space reasons but explain the two most interesting in more detail. This is, firstly, the state machine of the *ProcessorModule* called *Processor* in Fig. 4. It has three control states: *idle*, *retrieving*, and *calling*. The *idle* state is also the initial state. On request of an emergency call, either by *startCall* from the *ButtonBox* or *notify* from the *AutomaticNotification*, it changes to the *retrieving* state. This means that it waits for the GPS data. Having received this data, the state changes to *calling* because the *CommunicationsModule* is invoked to make the call. In case of success, it returns to the *idle* state and lights the green LED on the *ButtonBox*. Furthermore, the state machine can handle cancel requests and making a test call.

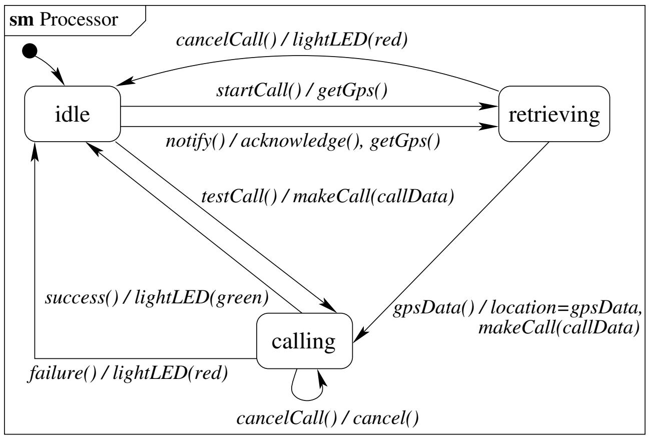
sm Processor
cancelCall() / lightLED(red)
startCall() / getGps()
idle
retrieving
notify() / acknowledge(), getGps()
testCall() / makeCall(callData)
success() / lightLED(green)
calling
gpsData() / location=gpsData, makeCall(callData)
failure() / lightLED(red)
cancelCall() / cancel()

Fig. 4. The state machine diagram of the *ProcessorModule*.

The second state machine is *Communications* in Fig. 5, the behavior specification of *CommunicationsModule*. One of the main complicating factors here is the handling of four automatic retries until a failure is reported. The state machine starts in an *idle* state and changes to the *calling* state after the invocation of *makeCall*. The *offHook* signal is sent to the mobile communications device. Inside the *calling* state, we start in the state *opening line*. If the line is free, the *dialing* state is reached by dialing the emergency number. After the *connected* signal is received, the state is changed to *sending data* and the emergency data is sent. After all data is sent, the *finished* flag is set which leads to the *data sent* state after the *onHook* signal was sent to the mobile. After the mobile sends the *done* signal, the state machine reports *success* and returns to the idle state. In case of problems, the state is changed to *opening line* and the retries counter is incremented. After four retries the guard [*retries* $>=$ 5] evaluates to *true* and the call fails. It is also always possible to cancel the call which leads to a *failure* signal as well.

Results. The components of *MaydayDevice* are further analyzed in the following. At first we use our metrics suite from Sec. 2 to gather data about the model. The results can be found in Tab. 2. It shows that we have no inheritance in the current abstraction level of our model and also that the considered classes have no parts apart from MaydayDevice itself. Therefore the metrics regarding these aspects are not helpful for this analysis.

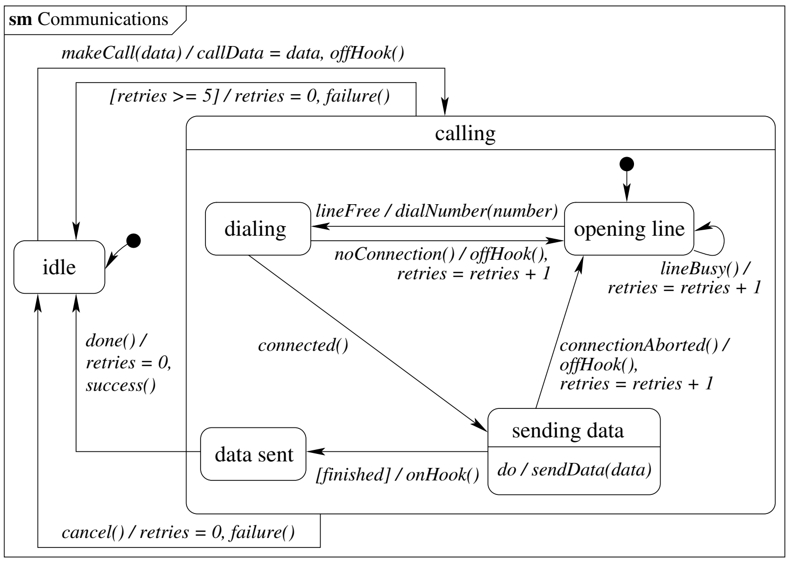

Fig. 5. The state machine diagram of the *CommunicationsModule*.

Table 2. The results of the metrics suite for the components of *MaydayDevice*.

Class	DIT	NOC	NOP	NRI	NPI	CCS
MaydayDevice	0	0	5	4	2	0
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

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 coupling and is therefore fault-prone. The same module has a high value for CCS but *Communications-Module* has a higher one and is also fault-prone.

In [25] there are detailed descriptions of acceptance and performance tests with the developed system. The system was tested by 14 volunteers. The usage of the system in the tests was mainly to provoke an emergency call by pressing the button on the button box.

The documentation in [25] 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 analysis identified the correct components. The types of the corresponding faults of the failures are not available.

3.2 MOST NetworkMaster

We further validated our approach on the basis of the project results of an evaluation of model-based testing [26]. A network controller of an infotainment bus in the automotive domain, the MOST® NetworkMaster [27], was modeled with the case tool AutoFOCUS and test cases were generated from that model and compared with traditional tests. An implementation in C running on a standard PC was tested. We use all found faults from all test suites in the following. The AutoFOCUS notation is quite similar to UML 2.0 which allows straight-forward application of the metrics defined earlier.

Device Design. The composite structure diagram of the network master is shown in Figure 6. It contains two components *Divide* and *Merge* that are only responsible for the correct distribution of messages. The *MonitoringMgr* checks the status of devices in the network but has no behavior in the model, i.e. was functionally abstracted. The *RegistryMgr* is the main component. All devices need to register with it on startup and it manages this register. Finally, the *RequestMgr* answers requests about the addresses of other devices.

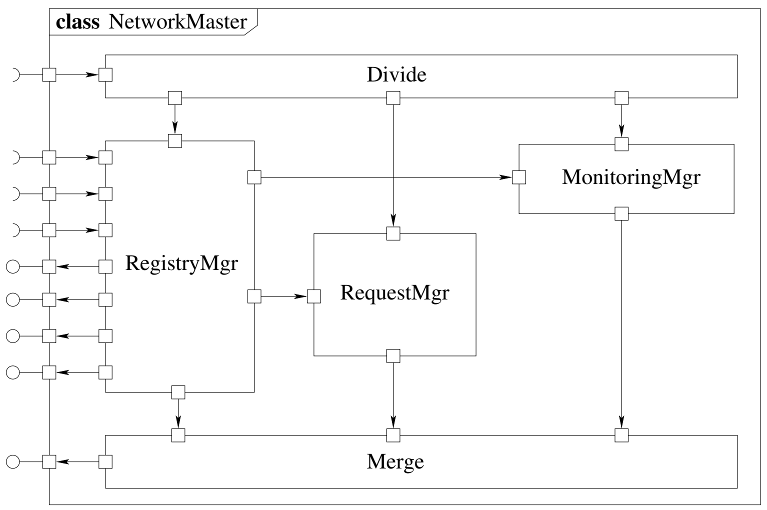

Fig. 6. The composite structure diagram of the MOST network master.

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 3.

Table 3. The results of the metrics suite for the NetworkMaster.

Class	DIT	NOC	NOP	NRI	NPI	CCS
NetworkMaster	0	0	5	4	5	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

Results. 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 [26], several test suites were executed against an implementation of the network master. Some of which were developed manually, other based on existing Message Sequence Charts, and the remaining ones were automatically derived from an AutoFocus model. There were 24 faults identified by the test activities of which 13 are programming faults, 9 requirements defects, and 2 model faults. Of these faults 21 can be attributed to the *RegistryMgr* and 3 to the *RequestMgr*. There were no faults revealed in the other components. Hence, the high fault-proneness of the *RegistryMgr* did indeed result in a high number of faults revealed during testing.

3.3 Discussion

The two case studies confirmed our approach for identifying fault-prone components using model metrics. In both cases the suite ranked the components as high-fault that had code implementations which actually contained the most faults. Both models were developed completely independent of the implementations. Hence, model faults that lead to implementation faults cannot have an influence. Unfortunately, inheritance was not used in the studies. Therefore the validity of these metrics remains to be shown. It holds for the whole approach that the the external validity of the results of the case studies is limited as the small sample size does not allow a thorough statistical analysis.

Correlation of Metrics. A main problem of software metrics is that different metrics are often not independent. We analyse our proposed metrics suite concerning the correlation of the different metrics based on the data from the case studies. The sample size is small therefore the validity is limited but may give first indications.

We cannot analyse DIT and NOC because they were not used in the case studies. Also it does not make sense to analyse NOP with only two non-null data points. Therefore we concentrate on NRI, NPI, and CCS. The correlation between NRI / CCS and NPI / CCS is low with a a correlation coefficient $r = -0.17$ and $r = -0,13$, respectively. Only the correlation between NRI and NPI is more interesting. The correlation coefficient is 0.55 but the Chi-test and

F-test only yielded probabilities of 0.35 and 0.17, respectively, for both data rows coming from the same population. Hence, we have a good indication that the metrics of our suite are not interdependent.

Correlation of Metrics and Faults. As we use the classification approach with our metrics, we cannot estimate numbers of faults and therefore a correlation between estimated and actual faults is not possible. Also a correlation analysis between the single metrics and the number of found faults is not helpful because only the combined suite can provide a complete picture of the complexity of the component. However, the statistical correlation between the metrics and the number of faults is not as low as expected. For NRI the coefficient is 0.35, for NPI 0.58, and for CCS 0.53 but chi- and f-tests showed a very low significance probably because of the small sample size.

Observations. By looking at the case studies it seems that the CCS metric has the most influence on the fault-proneness. However, there are components that do not have a state machine but their behavior is described by its parts and still might contain several faults. It also can be rather trivial to see that a specific component is fault-prone as in the case of the *RegistryMgr* of the *NetworkMaster*. This component has such a large state machine that it is obvious that it has to contain several faults. In larger models with a large number of components this might not be that obvious. Finally, there is no evident influence of the application type on the metrics visible from the case studies as both have components with a rather small number of interfaces and parts and a few components with quite large state machines.

4 Related work

There have been few approaches that consider reliability metrics on the model level: In [7] an approach is proposed that includes a reliability model that is based only on the static software architecture. A complexity metric that is in principle applicable to models as well as to code is discussed in [5], but it also only involves static structure as well. In [6] the cyclomatic complexity is suggested for most aspects of a design metric but not further elaborated.

In et al. describe in [28] an automatic metrics counter for UML. They classify their metrics into various categories including fault proneness. The metrics in this category are WMC, NOC, and DIT. The latter two are the same as in our approach. The calculation of WMC is given as the sum of the complexities of the methods but no further explanation is given how this complexity should be calculated from the model. State machines and structured classes are not analysed.

A white paper from Douglass [29] contains numerous proposals of model metrics for all types of UML models. Therefore this work has several metrics that are not comparable to ours. Moreover, detailed explanations of the metrics is not available for all of them. Our DIT metric is similar to the *Class Inheritance Depth (CID)*, and NOC is comparable to *Number of Children (NC)*. The *Class Coupling (CC)* aims at a similar target as the NRI and NPI metrics but does not consider the interfaces but the associations. Finally, there is a complexity metric for state machines called *Douglass Cyclomatic Complexity (DCC)* that is also based on the metric from McCabe but handles nesting and and-states differently. Also triggers and guards are ignored. The whole intention of DCC is different to our CCS metric. Douglass considers more the aspect of the complexity in terms of comprehensibility whereas we want to capture the inherent complexity of the behavior of the component. Douglass gives rough guidelines for values that indicate "good" models but does not relate the metrics to fault proneness.

Other approaches have been used for dependability analysis based on UML models, although these do not consider complexity metrics: In [30] an approach to automatic dependability analysis using UML is explained where automatic transformations are defined for the generation of models to capture systems dependability attributes such as reliability. The transformation concentrates on structural UML views and aims to capture only the information relevant for dependability. Critical parts can be selected to avoid explosion of the state space. A method is presented in [31] in which design tools based on UML are augmented with validation and analysis techniques that provide useful information in the early phases of system design. Automatic transformations are defined for the generation of models to capture system behavioral properties, dependability and performance. There is a method for quantitative dependability analysis of systems modeled using UML statechart diagrams in [32]. The UML models are transformed to stochastic reward nets, which allows performance-related measures using available tools, while dependability analysis requires explicit modeling of erroneous states and faulty behavior.

5 Conclusions

We propose an approach to determine fault-prone components of a software system in the design phase already by complexity analysis of the design models. We use the concept of the cyclomatic complexity of code, lift it to the model level and combine it with adjusted object-oriented metrics originally from [8] to a metrics suite for UML 2.0. The metrics from [8] and [11] have undergone several experimental validations, e.g. [2, 9, 4, 3]. Because we used these metrics as a basis for our metrics suite we believe that it is a good indicator for fault-proneness. This was confirmed in two industrial case studies.

The metrics can also be used in conjunction with static analyses of the model concerning reliability [33].

For future work, we plan further experimental work to validate the approach. Furthermore, as soon as more data is available a discriminant analysis similar to [34] will be used to get a more solid mathematical foundation for the classification.

Acknowledgments

We gratefully acknowledge the joint work with M. Baumgartner, C. Kühnel, A. Pretschner, W. Prenninger, B. Sostawa, T. Stauner, and R. Zölch on the MOST Network Master. Furthermore we are grateful to M. Broy and W. Prenninger for commenting on a draft version and A. Pataricza for many helpful suggestions.

References

1. Myers, G.: The Art of Software Testing. John Wiley & Sons (1979)
2. Khoshgoftaar, T., Woodcock, T.: Predicting Software Development Errors Using Software Complexity Metrics. IEEE Journal on Selected Areas in Communications **8** (1990) 253–261
3. Munson, J., Khoshgoftaar, T.: Software Metrics for Reliability Assessment. In Lyu, M.R., ed.: Handbook of Software Reliability Engineering. IEEE Computer Society Press and McGraw-Hill (1996)
4. Rosenberg, L., Hammer, T., Shaw, J.: Software Metrics and Reliability. In: Proc. 9th International Symposium on Software Reliability Engineering (ISSRE'98), IEEE (1998)
5. Card, D., Agresti, W.: Measuring Software Design Complexity. The Journal of Systems and Software **8** (1988) 185–197
6. Blundell, J., Hines, M., Stach, J.: The Measurement of Software Design Quality. Annals of Software Engineering **4** (1997) 235–255
7. Wang, W.L., Wu, Y., Chen, M.H.: An Architecture-Based Software Reliability Model. In: Proc. Pacific Rim International Symposium on Dependable Computing (PRDC'99). (1999) 143–150
8. Chidamber, S., Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Trans. Software Eng. **20** (1994) 476–493
9. Basili, V., Briand, L., Melo, W.: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Trans. Software Eng. **22** (1996) 751–761
10. Object Management Group: UML 2.0 Superstructure Final Adopted specification (2003) OMG Document ptc/03-08-02.
11. McCabe, T.: A Complexity Measure. IEEE Trans. Software Eng. **5** (1976) 45–50
12. Melton, A., Gustafson, D., Bieman, J., Baker, A.: A Mathematical Perspective for Software Measures Research. IEE/BCS Software Engineering Journal **5** (1990) 246–254
13. Fenton, N., Pfleeger, S.: Software Metrics. A Rigorous & Practical Approach. 2nd edn. International Thomson Publishing (1997)
14. Henry, S., Selig, C.: Predicting Source-Code Complexity at the Design Stage. IEEE Software **7** (1990) 36–44
15. Prenninger, W., Pretschner, A.: Abstractions for Model-Based Testing. In Pezze, M., ed.: Proc. Test and Analysis of Component-based Systems (TACoS'04). (2004)
16. Huber, F., Schätz, B., Schmidt, A., Spies, K.: AutoFocus: A tool for distributed systems specification. In Jonsson, B., Parrow, J., eds.: Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96. Volume 1135 of LNCS., Uppsala, Sweden, Springer (1996) 467–470
17. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons (1994)

18. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Available at http://www-106.ibm.com/developerworks/rational/library/ (1998)
19. Mayer, T., Hall, T.: A Critical Analysis of Current OO Design Metrics. Software Quality Journal **8** (1999) 97–110
20. Binkley, A., Schach, S.: Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In: Proc. 20th International Conference on Software Engineering (ICSE'98), IEEE Computer Society (1998) 452–455
21. Pintér, G., Majzik, I.: Program Code Generation based on UML Statechart Models. In: Proc. 10th PhD Mini-Symposium, Budapest University of Technology and Economics, Department of Measurement and Information Systems (2003)
22. Wagner, S.: Efficiency Analysis of Defect-Detection Techniques. Technical Report TUMI-0413, Institut für Informatik, Technische Universität München (2004)
23. Musa, J.: Software Reliability Engineering. McGraw-Hill (1999)
24. The ENTERPRISE Program: Mayday: System Specifications (1997) Available at http://enterprise.prog.org/completed/ftp/mayday-spe.pdf (January 2005).
25. The ENTERPRISE Program: Colorado Mayday Final Report (1998) Available at http://enterprise.prog.org/completed/ftp/maydayreport.pdf (January 2005).
26. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One Evaluation of Model-Based Testing and its Automation. In: Proc. 27th International Conference on Software Engineering (ICSE'05). (2005) To appear.
27. MOST Cooperation: MOST Media Oriented System Transport—Multimedia and Control Networking Technology. MOST Specification Rev. 2.3. (2004)
28. In, P., Kim, S., Barry, M.: UML-based Object-Oriented Metrics for Architecture Complexity Analysis. In: Proc. Ground System Architectures Workshop (GSAW'03), The Aerospace Corporation (2003)
29. Douglass, B.: Computing Model Complexity. Available at http://www.ilogix.com/whitepapers/whitepapers.cfm (January 2005) (2004)
30. Bondavalli, A., Mura, I., Majzik, I.: Automated Dependability Analysis of UML Designs. In: Proc. Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (1999)
31. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability Analysis in the Early Phases of UML Based System Design. Journal of Computer Systems Science and Engineering **16** (2001) 265–275
32. Huszerl, G., Majzik, I., Pataricza, A., Kosmidis, K., Cin, M.D.: Quantitative Analysis of UML Statechart Models of Dependable Systems. The Computer Journal **45** (2002) 260–277
33. Jürjens, J., Wagner, S.: Component-based Development of Dependable Systems with UML. In Atkinson, C., Bunse, C., Gross, H.G., Peper, C., eds.: Component-Based Software Development for Embedded Sytems. An Overview on Current Research Trends. Springer (2005) To appear.
34. Munson, J., Khoshgoftaar, T.: The Detection of Fault-Prone Programs. IEEE Trans. Software Eng. **18** (1992) 423–433