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 these costs is to concentrate these activities on the fault-prone components of the system. An approach is proposed that identifies these components based on detailed UML models. The approach is validated using two industrial case studies.

Keywords: model-based development, fault estimation, UML 2.0, complexity measures, design measures

Submission Category: Regular Paper

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% [26]. 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 exploit the existing resources as efficiently as possible. Detailed

[⋆] This research was supported in part by the DFG within the priority program *SoftSpez* (SPP 1064) under project name *InTime* and by the German Ministry for Science and Education within the *Verisoft* project.

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.

The complexity of software code has been studied to a large extent. The most widely known metrics concerning complexity are *Halstead's Software Metric* [14] and *McCabe's Cyclomatic Complexity* [22] and many variations of these. It is often stated that complexity is related to and a good indicator for the fault-proneness of software [20, 25, 30].

Although the traditional complexity metrics are not easily applicable[1] to design models, there are already a number of approaches that propose design metrics, e.g. [8, 5, 35, 9]. Most of the metrics in [9] were found to be good estimators of fault-prone classes in [2] and are used and adapted in the following. However, they concentrate mainly on the structure of the designs. Since the system structure is not a sufficient metric 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. Sec. 2 defines complexity metrics for models built with a subset of UML 2.0. In Sec. 3 the case study of an automatic collision notification system and in Sec. 4 of an automotive network controller are used to validate the approach. Finally, related work and conclusions are discussed in the Sections 5 and 6.

2 Analyzing Fault-Proneness

This section describes the possibilities to identify fault-prone components based on models built with UML 2.0 [27]. We introduce a design complexity metrics suite for a subset of model elements of the UML 2.0 and explain how to identify the fault-prone components.

[1] Kannst Du erklaeren, inwiefern ?

The basis of our metrics suite form the suite from [9] for object-oriented code and the cyclomatic metric from [22]. In using a suite of metrics we follow [23, 13] 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.

We adjust metrics to parts of UML 2.0 based on the design approach taken in ROOM [31] or UML-RT [32], 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.

Structured Classes. Structured classes 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 classes to have an internal structure consisting of other classes that are bound by connectors. Furthermore ports are introduced 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.

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.

The metrics proposed in the following consider composite structure diagrams of single classes or components with their parts, interfaces, connectors, and possibly state machines. 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. A corresponding example is given in Fig. 1.

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 and the more dependencies are there, all of which may contribute to failure. 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 [15] and is also a substitute for the old *Coupling Between Objects (CBO)* that was criticized in [21] in that it does not represent the concept of coupling appropriately.[2] 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.

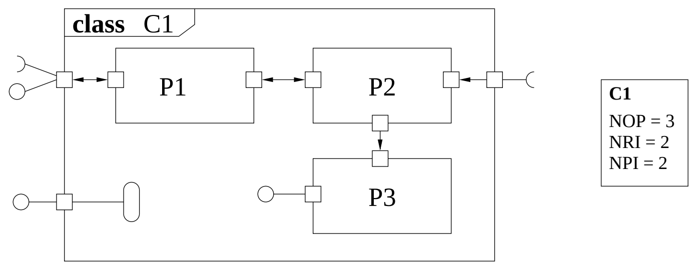

Fig. 1. An example structured class with three parts and the corresponding metrics.

[2] Kannst Du erklaeren warum diese metric besser ist ?

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.

We proceed with complexity metrics for behavioral models.

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 [22] 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 [22]. 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. An example for a possible code transformation of state machines can be found in [31].

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[3] 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 to add 1 for each transition because we have the implicit condition that the corresponding source state is active.[4] This results in the formula $CCS =$

[3] 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 [22].

[4] Was folgt daraus ? Wollen wir active states zaehlen ? (wurde vorher nicht erwaehnt)

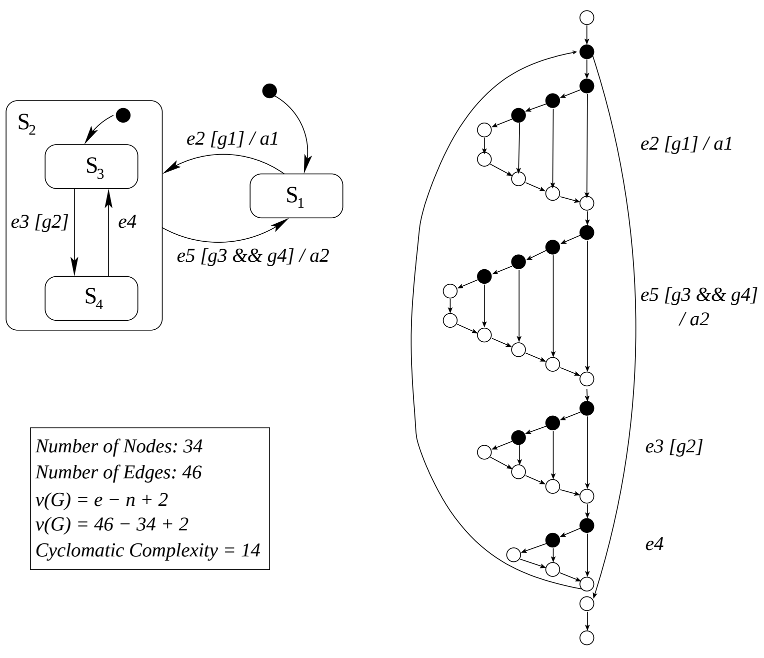

Fig. 2. (a) A simple state machine with one hierarchical state, event trigger, guard conditions, and actions. (b) Its corresponding control flow graph. The black vertices are predicate nodes. On the right the transitions for the respective part of the flowgraph are noted.

$|T| + |E| + |A_G| + 2$, 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.[5]

For this metric we have to consider two abstraction layers. First, we transform the state machine into its code representation[6] 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

[5] in welcher beziehung steht diese Formel zu den beiden Formeln oben ?

[6] 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.

complexity attributes of the code that implements it. The next abstraction to the control flow graph was established in [22].

In [16] 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.[7] This also holds for the CCS metric. Models of software can be based on various different abstractions, such as functional or temporal abstractions [28]. 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 [9] 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 [2]. However, we omit *Response For a Class (RFC)* and *Coupling Between Objects (CBO)* because they cannot be determined on the model level. Furthermore we consider the criticism from [21] in the following.[8] The remaining two metrics together with the new ones developed above form our metrics suite in Tab. 1. 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.

We consider whether our metrics are *structural complexity measure* by the definition in [23]. The definition says that for a set D of documents with a pre-order $\leq_D$ and the usual ordering

[7] inwiefern ?

[8] nochmal erklaeren was da drin stand und was daraus folgt

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 1. A summary of the metrics suite with its calculation

$\leq_{\mathbb{R}}$ 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. [9]

As mentioned before, complexity metrics are good predictors for the reliability of components [20, 25]. Furthermore the experiments in [2] show that most metrics from [9] 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 [20] 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

[9] in diesem Abschnitt habe ich viel geaendert, schau mal ob das passt

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 Case Study: Automatic Collision Notification

We validate our proposed fault-proneness analysis 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 UML, then we analyze the model and present the results.

Description. The case study we used to validate our results was done in cooperation with the car manufacturer BMW. 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 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 [11, 12] as a basis for the design model. 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 [11] 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.

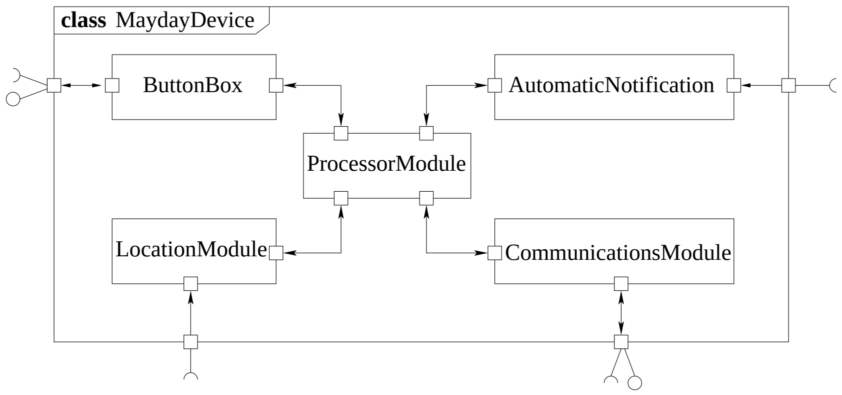

Fig. 3. The composite structure diagram of the mayday device.

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.

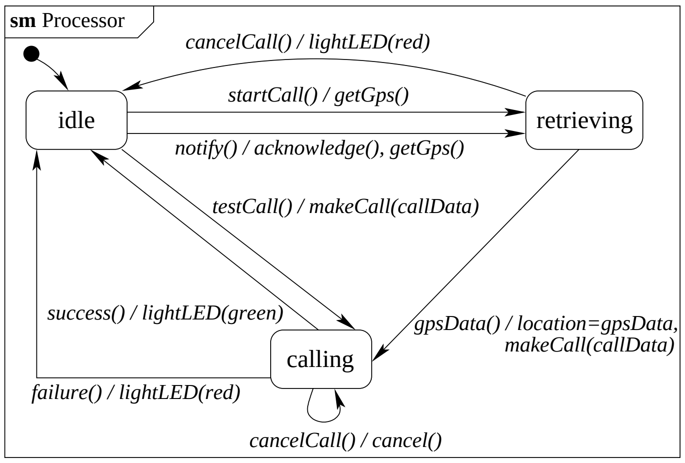
sm Processor
cancelCall() / lightLED(red)
idle
startCall() / getGps()
retrieving
notify() / acknowledge(), getGps()
testCall() / makeCall(callData)
success() / lightLED(green)
gpsData() / location=gpsData, makeCall(callData)
calling
failure() / lightLED(red)
cancelCall() / cancel()

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.

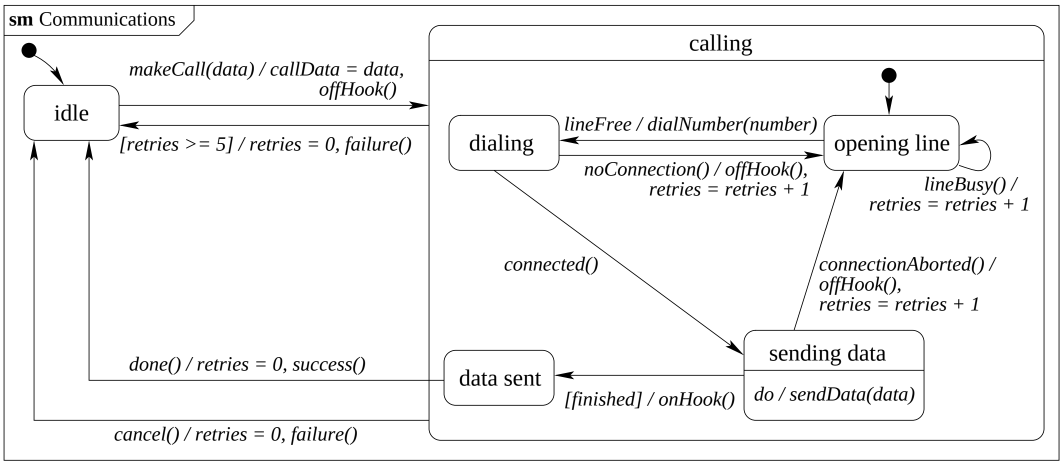
sm Communications
calling
idle
makeCall(data) / callData = data, offHook()
[retries >= 5] / retries = 0, failure()
dialing
lineFree / dialNumber(number)
opening line
noConnection() / offHook(), retries = retries + 1
lineBusy() / retries = retries + 1
connected()
connectionAborted() / offHook(), retries = retries + 1
sending data
do / sendData(data)
data sent
[finished] / onHook()
done() / retries = 0, success()
cancel() / retries = 0, failure()

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

Results. The five subcomponents of *MaydayDevice* are further analyzed in the following. At first we used 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. Therefore the metrics regarding these aspects are not helpful for this analysis.

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.

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

In [12] there are detailed descriptions of operational tests with the developed system. The usage of the system in the tests was mainly to provoke an emergency call by pressing the the button on the button box.

The documentation in [12] 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.

4 Case Study: MOST NetworkMaster

We further validated our approach on the basis of the project results of an evaluation of model-based testing [29]. A network controller of an infotainment bus in the automotive domain, the MOST network master [24], was modeled with the case tool AutoFocus [17] 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. The AutoFocus notation is quite similar to UML 2.0 which allows straight-forward application of the metrics defined earlier (as long as the used functions written in an action language are ignored[10]). The composite structure diagram of the network master is shown in Figure 6.

[10] nicht klar was damit genau gemeint ist - weglassen oder genauer erklaeren

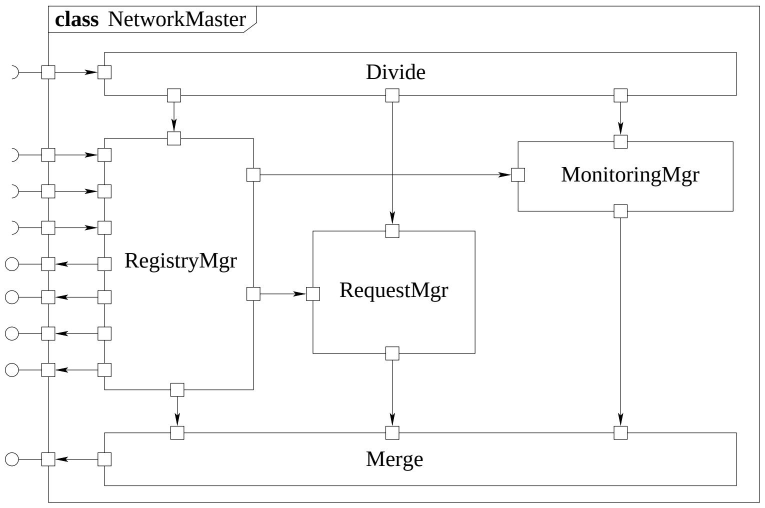

Fig. 6. The composite structure diagram of the MOST network master.

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.

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 3. 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 [29], several test suites were executed against an implementation of the network master. 24 faults have been identified by the test activities. 21 of these can be attributed to the *RegistryMgr*, 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.

5 Related work

There have been a few approaches that consider reliability metrics on the model level: In [35] an approach is proposed that includes a reliability model that is based on the static software architecture. A complexity metric that is in principle applicable to models as well as to code is discussed in [8], but it also only involves static structure. In [5] the cyclomatic complexity is suggested for most aspects of a design metric but not further elaborated.

Other approaches have been used for dependability analysis based on UML models, although these do not consider complexity metrics: [7] explains an approach to automatic dependability analysis using UML where automatic transformation 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. [6] presents a method 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. [18] explains a method for quantitative dependability analysis of systems modelled using UML statechart diagrams. 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. [3] explains an approach to dependability evaluation based on probabilistic Petri nets (which are similar to UML activity diagrams). Apart from reliability, other aspects of dependability have been considered using statecharts, including safety: [34] presents an approach to formal specify software for dependable systems which incorporates results of statecharts and Failure Mode and Effect Analysis in the development of formal specifications of fail-safe systems. It defines a general model of a safety-critical fail-safe system. Statecharts then support the construction of a formal specification by structuring informal functional requirements and formalizing safety requirements.

A number of other reliability assessment methods exist which do not apply to design models but to other software artefacts. Examples include the following: [33] explains a method which uses statistical testing as a verification technique for complex software. It uses a hierarchical specification produced in the STATEMATE environment for a black box analysis which is defined from behavior models deduced from the specification. [19] presents an approach for qualitative and quantitative reliability assessment which is based on the analysis and evaluation of software reliability by processing failure data collected on a software product during its development and operation. Failure prediction is improved by integrating software reliability modeling into an overall approach. The approach uses descriptive analyses, trend analyses, and reliability models to control testing activities, evaluate software reliability, and plan maintenance. [10] presents a method for the emulation of software faults in COTS components using software implemented fault injection. It aims to make fault locations ubiquitous, since every software module can be targeted no matter under whose control it is. [1] presents a fault injection tool designed to be adaptable to various target systems and different fault injection techniques. [4] presents a method for using test coverage to estimate the number of residual faults. It is applied firstly to justify the use of linear extrapolation to estimate residual faults and to show the importance of the amount of unreachable code in realistically estimating residual faults.

6 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 [9] to a metrics suite for UML 2.0. The metrics from [9] and [22] have undergone several experimental validations, e.g. [20, 2, 30, 25]. 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.

For future work, we plan further experimental work to validate the approach and especially the metrics suite.

Acknowledgments

We gratefully acknowledge the joint work with Martin Baumgartner, Christian Kühnel, Alexander Pretschner, Wolfgang Prenninger, Bernd Sostawa, Thomas Stauner, and Rüdiger Zölch on the MOST NetworkMaster. Furthermore we are grateful to Manfred Broy and Wolfgang Prenninger for commenting on a draft version.

References

1. Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Goofi: Generic object-oriented fault injection tool. In *The International Conference on Dependable Systems and Networks (DSN'01)*, 2001.
2. 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.
3. Simona Bernardi, Andrea Bobbio, and Susanna Donatelli. Petri nets and dependability. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *LNCS*. Springer, 2004.
4. Peter G. Bishop. Estimating residual faults from code coverage. In *SAFECOMP 2002*, LNCS. Springer, 2002.
5. J.K. Blundell, M.L. Hines, and J. Stach. The measurement of software design quality. *Annals of Software Engineering*, 4:235–255, 1997.
6. 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.
7. Andrea Bondavalli, Ivan Mura, and Istvan Majzik. Automated dependability analysis of uml designs. In *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999.

8. D.N. Card and W.W. Agresti. Measuring Software Design Complexity. *The Journal of Systems and Software*, 8:185–197, 1988.
9. S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
10. Diamantino Costa, Tiago Rilho, M. Vieira, and Henrique Madeira. Esffi - a novel technique for the emulation of coftware faults in cots components. In *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, 2001.
11. Mayday: System Specifications. The ENTERPRISE Program, 1997. Available at http://enterprise.prog.org/completed/ftp/mayday-spe.pdf (September 2004).
12. Colorado Mayday Final Report. The ENTERPRISE Program, 1998. Available at http://enterprise.prog.org/completed/ftp/maydayreport.pdf (September 2004).
13. N.E. Fenton and S.L. Pfleeger. *Software Metrics. A Rigorous & Practical Approach.* International Thomson Publishing, 2nd edition, 1997.
14. M.H. Halstead. *Elements of Software Science.* Elsevier North-Holland, 1977.
15. S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, 7:510–518, 1981.
16. S. Henry and C. Selig. Predicting Source-Code Complexity at the Design Stage. *IEEE Software*, 7:36–44, 1990.
17. 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.
18. G. Huszerl, I. Majzik, A. Pataricza, K. Kosmidis, and M. Dal Cin. Quantitative analysis of uml statechart models of dependable systems. *The Computer Journal*, 45(3):260–277, 2002.
19. K. Kanoun, M. Kaaniche, and J.-P. Laprie. Qualitative and quantitative reliability assessment. *IEEE Software*, 14(2):77–87, 1997.
20. 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.
21. T. Mayer and T. Hall. A Critical Analysis of Current OO Design Metrics. *Software Quality Journal*, 8:97–110, 1999.
22. T.J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, 5:45–50, 1976.

23. 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.

24. MOST Cooperation. MOST Media Oriented System Transport—Multimedia and Control Networking Technology. MOST Specification Rev. 2.2, Version 2.2-00. November 2002.

25. 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.

26. G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

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

28. 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.

29. 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. Submitted to the 27th International Conference on Software Engineering (ICSE) 2005.

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

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

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

33. P. Thevenod-Fosse and H. Waeselynck. STATEMATE applied to statistical software testing. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 99 – 109, 1993.

34. Elena Troubitsyna. Integrating safety analysis into formal specification of dependable systems. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

35. 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.