QAOOSE 2008 - Proceedings

Article

5 authors, including:

Yann-Gaël Guéhéneuc
Concordia University Montreal
260 PUBLICATIONS **5,163** CITATIONS

SEE PROFILE

Zoltán Porkoláb
Eötvös Loránd University
95 PUBLICATIONS **227** CITATIONS

SEE PROFILE

QAOOSE 2008 - Proceedings

12th ECOOP Workshop on

Quantitative Approaches in Object-Oriented Software Engineering

8th July 2008 - Paphos, Cyprus

Edited by:

Giovanni Falcone, Yann-Gaël Guéhéneuc, Christian Lange, Zoltán Porkoláb, Houari A. Sahraoui

QAOOSE 2008 - Proceedings

12th ECOOP Workshop on

Quantitative Approaches in Object-Oriented Software Engineering

8th July 2008 - Paphos, Cyprus

Edited by:

Giovanni Falcone, Yann-Gaël Guéhéneuc, Christian Lange, Zoltán Porkoláb, Houari A. Sahraoui

Outline:

QAOOSE 2008 is a direct continuation of eleven successful workshops, held during previous editions of ECOOP in Berlin (2008), Nantes (2006), Glasgow (2005), Oslo (2004), Darmstadt (2003), Malaga (2002), Budapest (2001), Cannes (2000), Lisbon (1999), Brussels (1998) and Aarhus (1995).

The QAOOSE series of workshops has attracted participants from both academia and industry that are involved/interested in the application of quantitative methods in object-oriented software engineering research and practice. Quantitative approaches in the object-oriented field is a broad and active research area that develops andor evaluates methods, practical guidelines, techniques, and tools to improve the quality of software products and the efficiency and effectiveness of software processes. The workshop is open to other technologies related to object-oriented such as component-based systems, web-based systems, and agent-based systems.

This workshop provides a forum to discuss the current state of the art and the practice in the field of quantitative approaches in the fields related to object-orientation. A blend of researchers and practitioners from industry and academia is expected to share recent advances in the fieldsuccess or failure stories, lessons learnedand seek to identify new fundamental problems arising in the field.

Organizers:

Giovanni Falcone, *University of Mannheim, Germany*

Giovanni Falcone is a PhD student at the Chair of Software Engineering, University of Mannheim, Germany. His PhD has the working title "Composition Metrics for Software Components" and will be completed in 2008. His research interests include High Performance Computing, Hardware/Software Codesign, Model Driven Architecture, Software Metrics, Component Based Development and Software Reuse.

Yann-Gaël Guéhéneuc, *University of Montreal, Canada*

Yann-Gaël Guéhéneuc is assistant professor at the Department of Informatics and Operations Research (software engineering group) of University of Montreal. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through reverse engineering and the identification of recurring patterns. He is also interested in empirical software engineering and in software laws and theories. He has published many papers in international conferences and leads the Ptidej team, which develops a tool suite to evaluate and to enhance the quality of object-oriented programs by promoting the use of patterns.

Christian Lange, *Eindhoven, University of Technology, The Netherlands*

Christian Lange is a postdoctoral researcher in the Software Engineering and Technology group at the Eindhoven University of Technology (The Netherlands). In 2007 he finished his PhD titled "Assessing and Improving the Quality of Modeling - A Series of Empirical Studies about the UML". His research interests include empirical software engineering, quantitative approaches, software quality, program comprehension, software architecture and software evolution. He is initiator of the EmpAnADa project for Empirical Analysis of Architecture and Design Quality at the TU Eindhoven. He is also the initiator of the MetricView tool. Christian Lange has published more than 20 papers in international journals, conferences,

and workshops such as: IEEE Software, ICSE, MoDELS/UML, ICPC, HICSS, or QAOOSE. He has served in the organizing committee of several international workshops, such as QAOOSE, Model Size Metrics (MSM, co-located with models, and the BENEVOL workshop for research on software evolution in Belgium and the Netherlands.

Houari A. Sahraoui, *University of Montreal, Canada*

Houari A. Sahraoui is associate professor at the department of computer science and operations research (software engineering group) of University of Montreal. Before joining the university, he held the position of lead researcher of the software engineering group at CRIM (research center on computer science, Montreal). He holds an Engineering Diploma from the National Institute of Computer Science (1990), Algiers, and a Ph.D. in Computer Science, Pierre & Marie Curie University LIP6, Paris, 1995. His research interests include the application of artificial intelligence techniques to software engineering, object-oriented metrics, software quality, software visualization, and software reverse- and re-engineering.

He has published around 100 papers in conferences, workshops, and journals and edited two books. He served as steering, program and organization committee member in several major conferences (ECOOP, ASE, METRICS, ICSM...) and as member of the editorial boards of two journals. He was the general chair of the IEEE Automated Software Engineering Conference in 2003.

Zoltán Porkoláb, *Eötvös Loránd University, Hungary*

Zoltán Porkoláb is associate professor at the department of Programming Languages and Compilers, at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, with almost 20 years of teaching experience both in the higher education and in the form of industrial trainings. He has finished his PhD in 2003 on the structured complexity of object-oriented programs. His research area is generative programming and software complexity, especially the connections between metrics and software paradigms. He has published more than 60 papers in workshops, conferences an journals. He has served in organizing committees, such ECOOP 2001, and programming committees, like GPCE 2007. He is the main organizer of WGT, an ETAPS satellite workshop, 2008.

Contents

1 Quantitative Comparison of MC/DC and DC Test Methods
Zalán Szügyi and Zoltán Porkoláb **1**

2 The AV-graph in SQL-Based Environment
Norbert Pataki, Melinda Simon, and Zoltán Porkoláb **11**

3 Quantitative analysis of testability antipatterns on open source Java applications
Muhammad Rabee Shaheen and Lydie du Bousquet **21**

4 Evaluating Quality-in-Use Using Bayesian Networks
M.A Moraga, M.F. Bertoa, M.C. Morcillo, C. Calero, and A. Vallecillo **31**

5 Metrics for Analyzing the Quality of Model Transformations
M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand **41**

6 A Basis for a Metric Suite for Software Components
Giovanni Falcone and Colin Atkinson **53**

Workshop Program

09:00 - 09:15	Welcome and QAOOSE introduction
09:15 - 10:30	Session 1: Graph-based Assessment
09:15 - 09:45	Quantitative Comparison of MC/DC and DC Test Methods *Zalán Szügyi and Zoltán Porkoláb*
09:45 - 10:15	The AV-graph in SQL-Based Environment *Norbert Pataki, Melinda Simon, and Zoltán Porkoláb*
10:15 - 10:30	Session Discussion
10:30 - 10:45	Session Break
10:30 - 11:45	Session 2: Quality Assessment
10:30 - 11:00	Quantitative analysis of testability antipatterns on open source Java applications *Muhammad Rabee Shaheen and Lydie du Bousquet*
11:00 - 11:30	Evaluating Quality-in-Use Using Bayesian Networks *M.A Moraga, M.F. Bertoa, M.C. Morcillo, C. Calero, and A. Vallecillo*
11:30 - 11:45	Session Discussion
11:45 - 12:15	Coffee Break
12:15 - 13:30	Session 3: Metrics
12:15 - 12:45	Metrics for Analyzing the Quality of Model Transformations *M.F. van Amstel, C.F.J. Lange, and M.G.J. van den Brand*
12:45 - 13:15	A Basis for a Metric Suite for Software Components *Giovanni Falcone and Colin Atkinson*
13:15 - 13:30	Session Discussion
13:30 - 15:30	Lunch Break
15:30 - 17:00	Session 4
15:30 - 17:00	Discussions and/or Work on a common problem/project/paper

Quantitative Comparison of MC/DC and DC Test Methods*

Zalán Szűgyi and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{lupin, gsd}@elte.hu

Abstract. Coverage refers to the extent to which a given verification activity has satisfied its objectives. There are several type of coverage analysis exists to check the code correctness. Usually the less strict analysis methods require fewer test cases to satisfy their requirements and the more strict ones require more. But it is not clear how much is the "more". In this paper we concern to the Decision Coverage and the more strict Modified Condition / Decision Coverage. We examined several projects used in the industry by several aspects: McCabe metric, nesting and maximal argument number in decisions. We discuss how these aspects are affected the difference of the necessary test cases for these testing methods.

1 Introduction

Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Appropriate coverage measures give the people doing, managing, and auditing verification activities a sense of the adequacy of the verification accomplished. [1]

The code coverage analysis contains three main steps [3], such as: finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage and determining a quantitative measure of code coverage, which is an indirect measure of quality. Optionally it contains a fourth step: identifying redundant test cases that do not increase coverage.

The code coverage analysis is a structural testing technique (white box testing), where it compares test program behavior against the apparent intention of the source code. Different types of analysis requires different set of test cases. We concern to Decision Coverage (DC), and Modified Condition / Decision Coverage (MC/DC) testing methods. The DC only requires that every lines of code in a subprogram must be executed and every decisions must be evaluated both to true and false. The MC/DC is more strict. It contains the requirements of DC and it demands to show that every condition in a decision independently affects

* Supported by the Hungarian Ministry of Education under Grant FKFP0018/2002

the outcome. It is clear there are more test cases are needed to satisfy the requirements of MC/DC than DC. But it is not so trivial how much can be spared when testing by DC instead of MC/DC. In this paper we answer that question by analyzing several projects used in the industry. These projects was written in Ada programming language and we analyzed them in several aspects: McCabe metrics, nesting, and maximal argument number in decisions. We examined how these aspects affected the difference of the necessary test cases.

In the second chapter we describe the most frequently used coverage metrics. In the third chapter we give a detailed description about how we analyzed the source codes of projects. Then we discuss the results of our analysis in the fourth chapter. And the summary and the conclusion comes in the fifth chapter.

2 Coverage Metrics

In this chapter we describe some commonly used coverage metrics.

2.1 Statement Coverage

To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. The main advantage of this method is that it can applied directly in object code and is not necessary to process source code. But this method is insensible to some control structure. Let us see the following example:

```
T* t = NULL;
if (condition)
    t = new T();
t->method();
```

In this example only one test case (where the `condition` is true) is enough to achieve 100% statement coverage because every statement is invoked once. In that case our program works fine, and we recognize it is faultless. But in the real usage, the condition can be false, and it causes in-deterministic behavior or segmentation fault.

2.2 Decision Coverage

This method requires that every statement must be invoked at last once and every decision must be evaluated as true and as false. In this case the error from the previous example turns out in testing time. This metric has the advantage of simplicity without the problems of statement coverage. A disadvantage is ignoring branches within boolean expressions which occur due to short-circuit operators. Let us see to following example:

```
if A or B then
```

Two test cases where (A = true, B = false and A = false, B = false) can satisfy the requirements of DC, but the effect of B is not tested. Thus these test cases cannot distinguish between the decision (A or B) and the decision A.

2.3 Modified Condition / Decision Coverage

The MC/DC criterion requires that every statement must be invoked at least once, every decision must be evaluated as true and as false, and each condition must be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. In our example three test cases (where A = true, B = true and A = true, B = false and A = false, B = true) provide MC/DC. The MC/DC is refined by Condition / Decision Coverage. You can read more information of these coverage methods in [1] and [3].

3 Analysis Method

In this chapter we describe our method to analyze the source codes written in Ada programming language. We used Antlr [4] parser generator with [5] grammar file to create the Abstract Syntax Tree (AST) of the source code. Our analysis is used on this AST.

3.1 Counting Test Cases for Decision Coverage

The Decision Coverage requires that every decision must be evaluated as true and as false at least once. So we need at least two test cases for every decision to satisfy these requirements. But one test case can test several decisions and more then two test cases are needed if a decision contains nested decisions. Let us see the following example:

```
if Condition_1 then
    if Condition_2 then
        true_statement_2;
    else
        false_statement_2;
    end if;
else
    false_statement_1
end if;
...
if Condition_3 then
    true_statement_3;
end if;
```

Three test cases are needed to cover DC where the Condition_1, Condition_2, Condition_3 are for example: (true, true, true), (true, false, false), (false, any, any). The Condition_1 must be evaluated as true twice because it has a nested decision in its true part. The Condition_3 can be tested the same time as Condition_1 because they are in the same level.

In summary we can say, $A + B$ test cases are needed to cover a decision. A is either the number of necessary test cases that are nested within a true consequence or 1 if there is no nested decision there. B means the same but in false consequence. A subprogram may contain more decisions in a same level. If all of these decisions are unique then we calculate the $max(A_i + B_i)$ where A_i, B_i belongs to the i^{th} decision ($i = 1..number\ of\ decisions$).

If there are identical decisions in same level we classify them. The identical decisions will be placed into the same class. Then we consider the $max(A_j + B_j)$ where A_j, B_j belongs to the j^{th} class. We calculate A_j and B_j in the following way: A_j `= max (` $A_{j_1}..A_{j_k}$ `),` B_j `= max (` $B_{j_1}..B_{j_k}$ `)` where `k` is the number of the decisions in class `j`. A_{j_l} and B_{j_l} are the number of necessary test cases for true and false consequences of the corresponding decision. ($l = 1..k$)

3.2 Counting Test Cases for Modified Condition / Decision Coverage

In this case we have two main steps. First we count how many test cases are needed to cover the decisions separately and then we check how these decisions affect each other. If a decision contains more than 15 arguments, then we calculate with argument number plus one test cases, which comes from [1].

Analyzing decisions separately

- If the decision contains only one argument or the negation of that argument we need exactly two test cases. Dealing with this case is same as we do in Decision Coverage.
- If the decision contains two arguments with logical operator `and, and then, or, or else,` or `xor` we need exactly three test cases:
  - TT, TF, FT for `and`
  - TT, TF, one of FT, FF for `and then`
  - FF, FT, TF for `or`
  - FF, FT, one of TF, TT for `or else`
  - three of TT, TF, FT, FF for `xor`

  where T means true and F means false.
- If the decision contains more arguments, then we use the following algorithm:
  1. Transform the AST that belongs to the decision to contain information about the precedence of logical operators. (The AST, which generated by [4] is a bit different.)
  2. Generate all the possible combinations of values for the arguments. (2^n combinations, where `n` is the number of arguments.) These are the potential test cases.
  3. Eliminate the masked test cases. For example let us consider `A and B`, where `B` is `false`. In this occasion the whole logical expression is `false` and independent of `A`. But `A` is not necessarily a logical variable. It can be another logical expression too and in this case the outcome value of `A` does not affect the whole logical expression. Therefore this test case is masked for `A` and it can be eliminated (for `A`). You can find a more detailed description and examples in [1] about this step.

4. For every logical operator in the decision: we collect the non-masked test cases which satisfy one of its requirements. So we get a set of test cases for every requirement of every logical operator. If one of these sets is empty the decision cannot be covered 100% by MC/DC. If this happens we try to achieve the highest possible coverage.
5. Calculate the minimal covering set of these sets. We do it in the following way: let us suppose we have n arguments in a decision. The maximum number of test cases is $m = 2n$ and we number them $0..m-1$. Of course almost all will be masked. Let us suppose all of the logical operators has two arguments (neither of them are `not`), so we have $s = 3*(n-1)$ sets. We calculate the minimal covering set by Integer Programming, where for every s_i set we have a disparity which is:

$$\sum_{k=0}^{m-1} \chi_{k \in S_i} X_k > 1$$

And our target function is:

$$min \sum_{k=0}^{m-1} X_k$$

With constraint: the value of every x_k can be only 0 or 1. When the result is calculated we get the minimal covering set. Every test case indexed with k is a member of the minimal covering set if x_k is 1.
To do that calculation we used Lemon graph library [5] with glpk linear programming kit [6].

`Analyzing decisions together`
Like in DC one test case can test several decisions when they are in same level, and one decision may require more test cases when it has nested decisions. But the way to calculate this is a bit more difficult because we have to deal with conditions in a decision. Here is an example about the problem of decisions in same level:

```
...
if a and b then
    ...
end if;
...
if c or d then
    ...
end if;
...
```

There are three test cases necessary to satisfy the requirements of both of these decisions. The test cases for the first decision are: TT, TF, FT, and for the

second decision are: FF, TF, FT. So three test cases can exercise both of the decisions simultaneously, because their conditions are independent. But let us see what happens if we change the `c` to `a` in the second decision:

```
...
if a and b then
    ...
end if;
...
if a or d then
    ...
end if;
...
```

Now three test cases are not enough because in the first decision, `a` has to be `true` twice and `false` once. And in the second decision it must be `true` once and `false` twice. So we need four test cases and two of them have to evaluate `a` as `true` and two others as `false`.

The method to calculate how many test cases are needed for decisions standing in same level:
Decision 1 has n variables: $a_1, ..., a_n$
Decision 2 has m variables: $b_1, ..., b_m$
The first s variables are the common variables where $s \leq min(n, m)$
Our algorithm works with k variables $c_1, ..., c_k$ where $k = n + m - s$
$c_i.true$ means the number of test cases where the variable c_i evaluated as $true$.
$c_i.false$ means the number of test cases where the variable c_i evaluated as $false$.
Let us consider:

$$c_i.true = \begin{cases} max(a_i.true, b_i.true) & \text{if } i = 1..s \\ a_i.true & \text{if } i = s+1..n \\ b_{i-n}.true & \text{if } i = n+1..n+m-s \end{cases}$$

$$c_i.false = \begin{cases} max(a_i.false, b_i.false) & \text{if } i = 1..s \\ a_i.false & \text{if } i = s+1..n \\ b_{i-n}.false & \text{if } i = n+1..n+m-s \end{cases}$$

If there are more than two decisions, we start the algorithm again with $c_1, ..., c_k$, and the variables of the next decision, and repeat it until all the decisions are processed.
Number of test cases:

$$max_{i=1..k}(c_i.true + c_i.false)$$

We deal with the nested decisions in the following way. Let us see an example:

```
...
if a or b then //first decision
    if c and d then  //second (nested) decision
        ...
    end if
end if
...
```

There are three test cases that are needed for both decisions: TF, FT, FF for the first and TT, TF, FT for the second. The variables are independent, so we can test them simultaneously. But in the third case the first decision is false, therefore the second decision cannot be executed. So we need an extra test case – where the first decision is true – to exercise the third requirements of the nested decision.

In general we calculate the maximum number of test cases that are needed to exercise the requirement of true and false consequences of decisions (m_{true}, m_{false} are the corresponding values). Then we get the set of test cases which cover the decision. We calculate how many of them are evaluated as `true` and how many are `false`. (The corresponding values are: d_{true}, d_{false}.) Then the number of necessary test cases are:

$$max(m_{true}, d_{true}) + max(m_{false}, d_{false})$$

We always consider the variables in nested decisions independent from the variables of outer decisions. Our future work is to refine this method to deal with the same variables.

4 Measurement and results

We analyzed six projects written in Ada programming language provided by a company. In every project about fifty per cent of the subprograms have no decisions. These are the initialiser, getter and setter subprograms. About twenty per cent of the subprograms have only one argument in their decisions. Every project has a similar ratio of subprograms having a different number of arguments in decisions. Thus we present our results of these projects all together. You can find information and more detailed statistics of these projects separately in [2] study. We used only those files which contain at least one subprogram definition, not only declarations. Let us see the details:

Number of files:	3448
Effective lines of code:	863631
Number of subprograms:	22842
Nr. of subprog. without decision:	12827
Nr. of subprog. with exactly 1 argument in their decisions:	7839
Nr. of subprog. with more arguments in their decisions:	2176

The table 1 shows the distribution of decisions by their argument numbers.

Number of arguments:	1	2	3	4	5	6	7	8	9
Number of decisions:	50302	3346	615	284	109	92	32	37	20

Number of arguments:	10	11	12	13	14	15	16	18	22	23	34
Number of decisions:	18	14	13	9	4	4	1	1	1	4	1

Table 1.

4.1 Differences and the McCabe metric

In the table 2 we can see how the McCabe metric values affect the difference between the necessary test cases for DC and MC/DC. We grouped the subprograms of the projects by their McCabe values, and we summarized the number of necessary test cases for DC and MC/DC. The `condition` column means the grouping conditions. E.g. 0..10 value in `Condition` column means those subprograms are in that group where the McCabe metric value is between 0 and 10. The `Nr.` column holds the number of subprograms in the group. The `DC` and `MC/DC` columns mean how many test cases are needed to cover all the subprograms in the group for DC and MC/DC. The `difference` column contains the difference of `DC` and `MC/DC` columns and the `Ratio` column means how many times more test cases are needed to cover MC/DC than DC.

Condition	Nr.	DC	MC/DC	Difference	Ratio
0..10	21306	37522	40027	2505	1.07
11..20	923	6931	7541	610	1.09
21..30	294	3210	3431	222	1.07
31..40	141	1903	2114	211	1.11
41..	178	5429	5911	482	1.09

Table 2.

Since the McCabe deals with the number of decisions and not their structure or their argument numbers, we can say the difference between the necessary test cases for DC and MC/DC do not depend on the McCabe metric. We accept this result, because the McCabe value of a subprogram can be high even if there is only one argument in decisions. In that way the DC and MC/DC values are same. And on the other hand the McCabe value is low when there are few decisions in a subprogram even if they have many arguments. In that way there can be a big difference between the DC and MC/DC values.

4.2 Differences and the Nesting

In the table 3 we grouped the subprograms by the deepness of the nested structures. The `Depth` means the maximum depth of nesting in subprograms that belongs to the corresponding group.

Depth	Nr.	DC	MC/DC	Difference	Ratio
0..1	17294	28291	29713	1422	1.05
2..3	4049	15167	16399	1232	1.08
4..6	1288	9152	10211	1059	1.12
7..	211	2385	2701	316	1.13

Table 3.

An increase in the maximum nesting value causes the increase of the ratio very slightly, thus it does not affect the difference of necessary test cases significantly.

4.3 Differences and the Maximum Argument Numbers

Here you can see how the largest decision (which contains the most arguments) affects the difference in the number of necessary test cases for DC and MC/DC. We grouped the subprograms by the number of arguments of the largest decisions, and the `Max Arg` field means that value. The table 4 shows these values.

Max Arg	Nr.	DC	MC/DC	Difference	Ratio
0	12827	12827	12827	0	1.00
1	7839	28350	28350	0	1.00
2..3	1778	10851	12989	2138	1.19
4..5	229	1633	2354	721	1.44
6..10	121	875	1595	720	1.82
11..	48	459	909	450	1.98

Table 4.

In the first two cases there is no difference between DC and MC/DC by definition. As the number of arguments increases in decisions, the difference is increasing as well. Decisions with more than ten arguments in subprograms require almost twice as many test cases for MC/DC than DC.

4.4 Differences Overall

In the table 5 the differences can be seen for the whole projects separately and in the last row together. The `DC` and `MC/DC` columns mean how many test cases are needed to cover all the subprograms in the projects for DC and MC/DC. The `Diff` and the `Rat` columns contain the difference and the quotient of `DC` and `MC/DC` columns. `A` means the all subprograms of the projects. In `B`, we excluded those subprograms which have no decisions at all. And in `C`, we excluded those subprograms that either have no decisions or there are no decisions with more than one argument.

	A				B				C			
	DC	MC/DC	Diff	Rat	DC	MC/DC	Diff	Rat	DC	MC/DC	Diff	Rat
1.	4449	4682	233	1.05	3315	3548	233	1.07	1007	1240	233	1.23
2.	3694	4031	337	1.09	3108	3445	337	1.11	1151	1488	337	1.29
3.	12611	13292	681	1.05	9312	9993	681	1.07	2837	3518	681	1.24
4.	4678	4908	230	1.05	3682	3912	230	1.06	1216	1446	230	1.19
5.	14391	15685	1294	1.08	10922	12216	1294	1.12	3082	4376	1294	1.42
6.	15172	16426	1254	1.08	11829	13083	1254	1.11	4554	5779	1254	1.27
$\sum$.	54995	59024	4029	1.07	42168	46197	4029	1.09	13818	17847	4029	1.29

Table 5.

5 Conclusion and future work

We analyzed several projects written in Ada programming language and estimated the difference of the required test cases of Decision Coverage and the more strict Modified Condition / Decision Cverage. We found that the difference is about five to ten per cent because the decisions in most subprograms have only one argument and there are several subprograms which do not contain decisions at all. If we exclude these subprograms we get a difference that is four times larger. Most importantly, the maximum number of arguments in decisions affects the difference. For those subprograms where there are decisions with more than six arguments, almost twice as many MC/DC test cases are needed as DC. But these subprograms are only less than one per cent of the whole project.

Our future work is to refine our analyzer program to do a better estimation in some exceptional cases, and we plan to do this analysis for some open source and other industrial projects too.

References

1. Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, Leanna K. Rierson: A Practical Tutorial on Modified Condition/Decision Coverage
2. Z. Szűgyi: Test cases for DC and MC/DC `http://people.inf.elte.hu/lupin/dc_mcdc_study.pdf`
3. Steve Cornett: Code Coverage Analysis `http://www.bullseye.com/coverage.html`
4. `http://www.antlr.org/`
5. Oliver Kellogg: `http://www.antlr.org/grammar/ada`
6. `https://lemon.cs.elte.hu/site/`
7. `http://www.gnu.org/software/glpk/`

The AV-graph in SQL-Based Environment

Norbert Pataki, Melinda Simon, and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{patakino, melinda, gsd}@elte.hu

Abstract. Software metrics play an important role in software engineering because the cost of testing and maintenance depends on the complexity of the software. Multiparadigm metrics can be used in various environments and they are applicable when more different paradigms are used in the same time. AV-graph is a multiparadigm metric that arises from imperative approach.
Structured Query Language (SQL) is the most common language for managing databases. SQL is a declarative language which is standardized. Most complex software systems use database support for store and manipulate information, compute calculations. Usually SQL appears as an embedded language in a high-level host language (for example C++, Java or C#). SQL queries are often string objects with no other type information in the host language. Most metrics do handle string objects without any semantical examination, but in SQL supported programs complexity of query strings may differ.
In this paper we examine the AV-graph in SQL-based environment. The AV-graph metric is interpreted in the database realm and extended for SQL statements. We compare the behaviour of the metric to the expected functionality. We refine our multiparadigm metric for SQL supported software systems.

1 Introduction

Since "You cannot control what you cannot measure", metrics play an important role in modern software engineering. The cost of testing and bugfixing highly depends on the structural complexity of the code. In software design the most significant part of the cost is spent on the maintenance of the product. Therefore many software metrics have been developed to measure the complexity of programs.

Most modern programs are written by using more paradigms. Object-oriented programs have large procedural components in implementations of methods. AOP implementations highly rely on OOP principles. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs. Moreover multiparadigm programs [5] appear in C++, Java, on the .NET platform, and others.

Databases are essential auxiliary tools creating large programs. They relieve querying, deleting and modifying data. SQL is a widely-used language for managing databases. However, SQL is not an imperative language but a set-based

declarative one. Inasmuch as we use SQL as an embedded language in an imperative host language (for example C#, Java, C++, etc.) we have multiparadigm programs.

Metrics applied to different paradigms than the one were designed for, might report false results [13]. Therefore an adequate measure applied to multiparadigm programs should not be based on special features of only one programming paradigm. A multiparadigm metric has to be based on basic language elements and construction rules applied to different paradigms. A paradigm-independent software metric is applicable to programs using different paradigms or in a multiparadigm environment. Paradigm-independent metrics should be based on general programming language features which are paradigm- and language independent. Multiparadigm metrics can be used in many various environments for many different purposes [10, 8].

This paper is organized as follows. The following section describes the most important features of SQL. Section 3 presents a motivating example and argues for multiparadigm metrics. Section 4 defines our multiparadigm metric in an informal way. An SQL-related interpretation of AV-graph metric is given in section 5. We evaluate this approach in section 6. Finally, section 7 presents our conclusions and sets the course for future work.

2 Structured Query Language (SQL)

SQL is a widespread language managing *relational database systems*. SQL was created to shield the database programmer from understanding the specifics of how data is physically stored in each database management system and also to provide a universal foundation for updating, creating and extracting data from databases.

SQL statements are divided into groups:

- Data Definition Language (DDL): these statements aid to create schemas, tables etc. (for example `create table` or `drop view`).
- Data Manipulation Language (DML): data can be stored or modified with these statements (for example `insert`, `update`, `delete`).
- Query Language (QL): The most typical querying statement for retrieve information from a database. The `select` statement can be found in this group.
- Data Control Language (DCL): these statements handle permissions (for instance, `grant` or `revoke`).

These groups are the most important ones, but there are some other groups that rarely used (for example debugging statements).

In this paper we do not deal with statements of DCL and all statement of DDL, because they appear as embedded statements very rarely.

SQL provides *aggregate functions* to assist with the summarization of large volumes of data (for example, `count`, `avg`, `sum`). These functions are often used in queries.

Clauses are (in some cases optional) constituent components of statements and queries. The basic clauses are:

- The `select` clause is used to specify what fields will be included in the query result. This clause is always found in SQL query statements.
- The `from` clause specifies what tables data will come from. This exists in all SQL query statements.
- The `where` clause specifies what subset of the data will be used (always has non-aggregate conditions). This clause is almost always found in SQL query statements.
- The `join` clause is used to link more than one table together. This is often found in more complex queries that require retrieving data from more than one table. There are several formats (for example `inner join`, `right join`, etc.).
- The `group by` clause is used to specify about what fields data should be aggregated.
- The `having` clause is very similar to where clause except the statements within it are of an aggregate nature.
- The `order by` clause sorts the records in the result set.

More queries can be combined with the following operations: `union`, `intersect`, `except`.

SQL was designed for a specific purpose: to query data contained in a relational database. SQL is a declarative language. However, there are extensions to Standard SQL which add procedural programming language functionality, for example Oracle's PL/SQL, PostgreSQL's PL/pgSQL and Microsoft's T-SQL. These tools depend on the database server.

3 Motivating example

Nowadays SQL often appears as an *embedded language*, therefore we can write SQL statements in a high-level host language [11, 12, 1, 2].

Let us consider the following C# code fragment that can be found in [11]

```
SqlConnection connection = new SqlConnection(
  "server=localhost;database=Northwind;uid=sa;pwd=sa
);

SqlCommand command = connection.CreateCommand();

command.CommandText =
  "SELECT TOP 5 CustomerID, CompanyName, ContactName, Address " +
  "FROM Customers" +
  "ORDER BY CustomerID";

connection.Open();
```

```
SqlDataReader reader = command.ExecuteReader();

while(reader.Read())
{
  Console.Writeline("reader[\"CustormerID\"] = " +
    reader["CustomerID"];
  ...
}

reader.Close();

connection.Close();
```

The complexity of this code fragment includes the complexity of the SQL query. But the query appears as a string literal object in the previous code. Usually metrics measure string objects without any semantical examination - maybe has constant complexity or zero complexity at all. Sometimes strings are generated in runtime with string operations (concatenation, replace, etc.) Therefore hard to measure complexity of embedded SQL statements [2].

Multiparadigm metrics can measure the complexity of the previous code snippet, but some strings should be measured as SQL queries and some of them should be measured as normal strings. We would like to measure embedded SQL queries and the host language code with the same metric because of consistency.

AV-graph is a multiparadigm metric that can be applied in this situation and it measures in a sophisticated way. But AV-graph metric should be refined to support SQL queries.

Not all host languages support database connections in a standard way. For example, the C++ standard library does not include standard database handler classes. Most SQL distributions offer tailor-made extensibles for C++. Therefore SQL queries' complexity cannot be measured via standard SQL connection classes. This is a problem because we cannot distinguish between strings according to their context of usages.

4 AV-graph metric

In this section we describe our multiparadigm metric in an informal way. The formal definition of the AV-graph can be found in [10].

The AV-graph complexity based on programs' three different characteristics. Complexity of dataflow, control structure, and datatypes are taken into account. We can transform the source code into a graph in a multiparadigm way because the previous characteristics cannot be binded to a specific paradigm and they appears in almost all kind of program.

The main concept behind the definition of AV-graph is that the complexity of a certain code element – either data or control – is heavily depends on its environment. The execution of a control node and the possible value of a data node

depends on the predicates dominating it. Thus understanding a node depends on its nesting depth.

There is an other possible way to get these results. We can map our AV-graph model with control and data nodes to the Howatt's model [6] without data nodes and data edges. Hence we replace data edges with special control nodes: "reader" and/or "writer". These control nodes only send and receive information. They will be inserted just before and after the real control nodes which read and/or write data. The nesting depth and complexity value we get with this model is the same as the AV-graph complexity.

This definition reflects our experience properly. For example, if we take a component out of the graph which does not contain a predicate node to form a procedure, (i.e. a basic block, or a part of it – this means a single node), then we increase the complexity of the whole program according to our definition. This is a direct consequence of the fact that in our measures so far we contracted the statement-sequences that are reasonable according to this view of complexity. If we create procedures from sequences, the program becomes slightly difficult to follow. Since we cannot read the program linearly, we have to "jump" from the procedures back and forth. The reason for this is that a sequence of statements can always be viewed as a single transformation. This could of course be refined by counting the different transformations as being of different weight, but this approach would transgress the competence of the model used. The model mirrors these considerations since if we form a procedure from a sub- graph containing no predicate nodes, then the complexity increases according to the complexity of the new procedure subgraph, (i.e. by 1).

On the other hand, if the procedure does contain predicate node(s), then by modularization we decrease the complexity of the whole program depending on the nesting level of the outlifted procedure. If we take a procedure out of the flowgraph, creating a new subgraph out of it, the measure of its complexity becomes independent of its nesting level. On the place of the call we may consider it as an elementary statement (as a basic block, or part of it).

As a matter of fact, we can decrease the complexity of a program in connection with data if we build abstract data types hiding the representation. In this case the references to data elements will be replaced by control nodes since data can only be handled through its operations. While computing the complexity of the whole program, we have to take into account not only the decreasing of the complexity, but also its increase by the added complexity determined by the implementation of the abstract data type. Nevertheless, this will only be an additive factor instead of the previous multiplicative factor.

That is the most important complexity-decreasing consequence of the object-oriented view of programming: the class hides its representation (both data structure and algorithm) from the predicates (decisions) supervising the use of the object of class. We can naturally apply our model to object-oriented programs. The central notion of the object-oriented paradigm is the class. Therefore we describe how we measure the complexity of a class first. We can see the class definition as a set of (local) data and a set of methods accessing them.

A data member of a class is marked with a single data node regardless of its internal complexity. If it represents a complex data type, its definition should be included in the program and its complexity is counted there. Up to the point, where we handle this data as an atomic entity, its effect to the complexity of the handler code does not differ from the effect of the most simple (built-in) types.

The complexity of a class is the sum of the complexity of the methods and the data members (attributes). As the control nodes (nodes belonging to the control structure of one of the member graphs) were unique, there is no path from one member graph to another one. However, there could be attributes (data nodes) which are used by more than one member graph. These attributes have data reference edges to different member graphs.

This model reflects the fact that a class is a coherent set of attributes (data) and the methods working on the attributes. Here the methods (member functions) are procedures represented by individual AV-graphs (the member graphs). Every member graph has its own start node and terminal node, as they are individually callable functions. What makes this set of procedures more than an ordinary library is the common set of attributes used by the member procedures. Here the attributes are not local to one procedure but local to the object, and can be accessed by several procedures.

Let us consider that the definition of the AV-graph permits the empty set of control nodes. In that case we get a classical data structure. The complexity of a classical data structure is the sum of the data nodes. The opposite situation is also possible. When a "class" contains disjunct methods – there is no common data shared between them –, we compute the complexity of the class as the sum of the complexities of the disjunct functions. We can identify this construct as an ordinary function library.

These examples also point to the fact that we use paradigm-independent notions, so we can apply our measure to procedural, object-oriented, aspect-oriented or even mixed-style programs. Now we consider what does this metric measure in the SQL realm.

5 Interpreting the AV-graph in SQL realm

AV-graph complexity includes three characteristics:

- complexity of datatypes
- complixity of dataflow
- complexity of control structure

We want to answer the following questions:

- What do the previous characteristics mean in SQL?
- How can we interpret AV-graph metric in this environment?

Complexity of datatypes can be considered as a complexity of tables in the database. It is not flabbergasting. When one writes a query, its complexity depends on the complexity of examined tables.

Direction of dataflow is taken into account in the model of AV-graph. This means we have to distinguish between the statements of DML and QL. Statements of DML write the data in the database. Statements of QL read information, they cannot modify it according to the standard.

Complexity of dataflow includes how many columns the query results in. The asterisk does not increase by the complexity – it stands for all columns. Therefore the complexity is based in how many columns enumerated in the query.

Usage of aggregate functions (for example, `count`) is taken into account at this point, because dataflow is nearly related to subprograms. We can think that `avg(Price)` is an ordinary function with one argument. This function call increases the complexity. Arguments in an aggregate function can be considered as input parameters, we do not modify them.

Special types (like `date`, `datetime`, `timestamp`, `text`, etc.) often demand special functions (like, `extract`, `substring`). These functions are similar to the aggregate functions, argumens are input parameters.

The keyword `distinct` is used to select only the different rows in a query. It can be considered as a special procedure – with zero argument. Its complexity is only the subprogram call with no parameter-passing complexity.

The most interesting question is how can we measure "complexity of control structure" of an SQL statement. SQL is declarative language, we cannot write loops. Of course, a query has a complexity based on its structure.

The easiest way to measure the structure-related complexity in an SQL statement is based on the clauses. Many clauses are not necessary in a query and complexity depends on the written clauses. Every optional clause increases the query's complexity.

In the where and having clause ordinary logical expression can be found – with boolean operators and adventitious function calls. Complexity of logical expressions means no problem.

The group by clause increases the complexity with the number of enumerated columns. The order by clause behaves similarly. The complexity of an order by clause can be increased by the specification of order.

Join clauses are similar to from clauses and inner selects, therefore join clauses modify the list of referred tables. They also change the executed condition. However, many modifiers can be used with join clauses and these modifiers (for example, inner, outer, left, right, etc.) also increase the join clause's complexity. The modifiers have the same constant complexity.

Combined queries' complexity is easy, because it is the sum of the complexity of the queries plus the operation. The operation has constant complexity. This complexity does not depend on the keyword, so the operation union, intersect, and except has the same complexity.

The "structural complexity" of an insert statement means how many values exist in the statement. Of course, this information is in connection with the complexity of the table, but the design of a table can decrease it because of the default values, etc.

The "structural complexity" of a delete statement can be measured as the complexity of its where clause.

The "structural complexity" of an update statement can be measured as a sum of its where clause's complexity and the number of the setted values in the statement.

We defined three important characteristics in connection with SQL statements. We extended the AV-graph metric to SQL-supported host languages.

6 Evaluation of the interpretation

In the previous section we have interpreted AV-graph metric in SQL queries that appear in high-level languages. Now we evalauate this interpretation, check it on easy examples and refine the metric.

Let us consider the following query:

```
select *
from Customers;
```

This is most simple query. When we measure the previous query we consider the following properties:

- this query reads the database
- the complexity of the table Customers
- all columns of table Customers this query results in, but we use the asterisk "metacolumn"

```
select
  CompanyName, ContactName, Address
from
  Customers
where
  CustomerID<10
order by
  CompanyName;
```

This query involves more features of SQL. In this case we consider the following properties:

- This query reads the database.
- The complexity of the table Customers
- Three columns of table Customers will this query result in.
- The records are ordered according to one column.
- We have a simple condition in the where clause.
- No function call can be found in this query.

```
delete from
  Customers
where
  CustomerID=10;
```

This is a delete statement. The following properties must be considered according to our metric:

- This statement modifies the database.
- The complexity of the table Customers
- We have a simple condition in the where clause.
- No function call can be found in this query.

The previous examples present what AV-graph metric is considered in SQL queries. It conforms to the experiences and to the original approach of AV-graph. However, some questions should be refined.

What is the complexity of a table? It can be the number of the columns, but of course, it should be weighted according to the type of columns.

How can we take into account the complexity of a dataflow in the case of an insert or a delete statement? The complexity of the modified table has to weighted with this information because that table is changed.

7 Conclusion and future work

Multiparadigm metrics can be used in various situations for various purposes. Many environments demand multiparadigm metrics for precise measurements. Our metric is a multiparadigm metric that comes from an imperative approach. It has been suggested what AV-graph means in SQL-based environment.

Measuring the complexity of SQL queries are important because complex software systems cannot being without database support. Nowadays usually SQL works as an embedded language. In this case the SQL statements appear as string literals and string objects that dynamically form queries. Hard to analyze code that use SQL as an embedded language.

In this paper we analyzed the behaviour of a multiparadigm metric called AV-graph in an SQL environment. We extended the AV-graph metric to SQL-supported host languages.

Many directions can be mentioned as future work. Measuring databases' complexity is made a motion [4]. In this paper we advocate measuring the *queries*. Therefore measuring databases' complexity would be to worth while.

Another extensions also can be mentioned. Procedural extensions and stored procedures often can be found when databases are involved in software development, because they can decrease the complexity. We do not measure the complexity of stored procedures and the usage of procedural extensions.

Another important question is related with implementation. How can we distinguish between the SQL-related and other string literals in a program? A special tool support is proposed in [2]. This tool support that is able to handle embedded queries in a wide range of host languages, such as COBOL, PL/SQL, Visual Basic 6, and Java.

References

1. Huib van den Brink, Rob van der Leek: Quality metrics for SQL queries embedded in host languages, in Proc. of Eleventh European Conference on Software Maintenance and Reengineering, 2007
2. Huib van den Brink, Rob van der Leek, Joost Visser: Quality Assessment for Embedded SQL, in proc. of Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2007, pp. 163–170.
3. Coral Calero, Houari Sahraoui, Mario Piattini: An Empirical Study with Metrics for Object-Relational Databases, In the proc. of the 7th European Conference on Software Quality (ECSQ02), 2002.
4. Coral Calero, Houari Sahraoui, Mario Piattini, Hakim Lounis: Estimating Object-Relational Database Understandability Using Structural Metrics, In Proc. of the 11th International Conference on Database and Expert Systems Applications (DEXA01), Munich, 2001.
5. James O. Coplien: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998.
6. J.W. Howatt and A.L. Baker: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, The Journal of Systems and Software, 10 (1989), 139–150.
7. Rick F. van der Lans: *Introduction to SQL: Mastering the Relational Database Language*, Fourth Edition/20th Anniversary Edition, Addison-Wesley (2006).
8. Norbert Pataki, Ádám Sipos, Zoltán Porkoláb: Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric, In Proc. of 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2006, Nantes, pp. 1–11
9. Mario Piattini, Coral Calero, Houari Sahraoui, Hakim Lounis: Object-relational database metrics, L'Objet, Vol. 17, No. 4, Edition Hermès Sciences, 2001, pp. 477–498.
10. Zoltán Porkoláb, Ádám Sillye: Towards a multiparadigm complexity measure, 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering QAOOSE Workshop, ECOOP 2005, Glasgow, pp. 134–142.
11. Jason Price: *Mastering C# Database Programming*, Sybex, 2003.
12. George Reese: *Database Programming with JDBC and Java*, 2nd Edition, O'Reilly, 2000.
13. G. Seront, M. Lopez, V. Paulus, N. Habra: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, QAOOSE Workshop, ECOOP 2005, Glasgow, pp. 109–117.
14. Jeffrey D. Ullman, Jennifer Widom: *A First Course in Database Systems*, Prentice Hall, 1997.

Quantitative analysis of testability antipatterns on open source Java applications

Muhammad Rabee Shaheen and Lydie du Bousquet

Laboratoire Informatique de Grenoble (LIG)
Universités de Grenoble (UJF)
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{muhammad-rabee.shaheen,lydie.du-bousquet}@imag.fr

Abstract. Testability is a software characteristic that aims at producing systems easy to test. Antipatterns is a factor that could affect negatively the testability of a software. Dependency is the main idea that is found behind different studies related to antipatterns. Testability antipatterns identify the existed weaknesses in a design pattern. In this paper we introduce a quantitative analysis to show some factors that lead to increase the antipatterns.

Keywords: test, testability, antipatterns, metrics

1 Introduction

Software testing is one of the most expensive phases in the life cycle of a software development. It is expensive in terms of time and money. It can represent 40% of the total development cost. Practitioners and researchers are looking for solutions to decrease the cost of testing. One idea is building systems easy to test, in order to reduce this cost. It is called "design for testing" or "design for testability" [8]. To do that one needs to identify the different types of weaknesses that make test process expensive. This idea relies on the observation that for a same problem, different solutions (with different designs) can be produced. Some of them are easier to test than others. "Design for testability" favors design solution(s) that would ease the test.

Several types of solution have been proposed for "Design for testability". Some of them are based on using testability metrics that have been proposed for OO design and code such as the Chidamber and Kemerer set [14, 15]. Some of these metrics are used as quality predictor. A large part of researches focuses on the validation of these metrics with respect to their predictive abilities [2, 10, 28, 9, 11, 17, 32, 23, 33, 12, 16]. Another solution is detecting and avoiding "testability antipatterns".

A testability antipattern is a design solution known to make test difficult (and/or known to increase the number of test to carry out) [3]. Two testability weaknesses have been described in [5], ***self usage*** and ***interactions***. Both of them

characterize dependency cycles in classes. Several studies have been done to introduce general testability metrics [13, 14, 8, 25, 18, 22], few studies have been carried out to identify the different weaknesses of a design patterns [3, 24].

This paper describes an empirical quantitative analysis of testability antipatterns. We studied 14 open-source Java applications to detect the self usage and interactions antipatterns occurrences. The objective was to observe how frequent and how complex are the cycles in those applications. We especially want to see if the cycles occurred with some code characteristic, where such cycles influence the difficulty of integration testing.

The rest of this paper is organized as following. Section 2 introduces the concept of testability. Section 3 describes the antipatterns. Section 4 presents our quantitative study. Section 5 concludes and draws some perspectives.

2 Testability

Originally, testability was defined for hardware components. In this context, testability is often characterized through observability and controllability. To test a component, one must be able to control its inputs and observe its outputs. When a component is embedded, its controllability and observability may be decreased. This partly depends on the architectural design.

For several decades, testability has become a preoccupation for software systems. Several software-oriented definitions have been proposed for testability. One reason for the diversity of testability definitions is that lots of elements can influence the work of testing.
Most testability definitions characterize testability as the complexity or the effort required for testing. For some of them, software testability is expressed in terms of controllability and observability [8, 30, 20, 21, 19, 29]. Other testability definitions are related implicitly or explicitly with some specific testing method [1, 26]. In [1], testability is the effort needed for testing. For Binder, testability is the relative ease and expense of revealing software faults [8]. Other definitions allow a quantitative evaluation of the testing effort. For IEEE, it is also considered as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met" [27]. In [6], the testability is the probability that a test of the program on an input drawn from a specified probability distribution of the input is rejected, given a specified oracle and given that the program is faulty. In other words, it is the probability to observe an error at the next execution if there is a fault in the program.

In [8], Binder identifies 6 primary factors influencing testability: ***representation***, ***implementation***, ***built-in test***, ***test suites***, ***testing environment/tool***, and ***process*** (see Fig. 1). He notices that "as practical matter, testability is as

much a process issue as it is a technical problem".

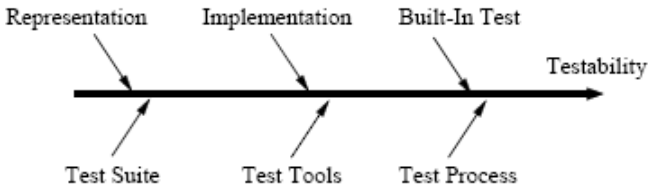

Fig. 1. Testability main facets

To detect testability weaknesses, several metrics have been proposed. Most of them focus on unit testing (either at a class or a method level). At integration level, the antipatterns were proposed to be metrics could be used at system or integration level. Difficulty of testing usually depends on dependencies among classes, which require an order in the integration. When a dependency cycle exists, it has to be *broken*. Detecting these anitpatterns could be done either at ***presentation*** facet or at ***implementation*** facet Fig. 1.

3 Antipatterns

Design patterns represent solutions to problems that arise when software is being developed in a particular context [3]. An antipattern "describes undesirable configurations in the class diagram". They could affect testability efforts intractable and make tests ineffective [7, 4].
Two testability antipatterns were proposed in [5]. Both captures the fact that there exists sine dependency cycle among the classes.

An object oriented system is a set of classes that communicate with each other. Some of these classes depend on others. In [24] two types of dependencies were defined. Physical dependency exists between two classes A and B if A cannot be ***compiled*** without B (e.g. inheritance). Logical dependency exists between A and B if a change happens to a B would require a change to A.
A dependency relationship could be either direct or indirect. It is indirect if the path between A and B includes other classes, otherwise it is direct. Figure 2(a) shows a dependency (interaction) between the class C and D, which could be interpreted either directly, or indirectly through the inheritance tree. Figure 2(b) represents the notion of class cycle, for example, one cycle exists between D and E, another one exists also between B and D through C.

The complexity of a dependency cycle increases when there are more than one path to reach from one class to another, which in turns increase the potential usages. That means more test cases or at least potential hidden errors. Also

more classes in a cycle will increase the difficulty of the test, one reason for that is the need for more instances to be instantiated.
These types of dependency (interaction through more than one path and class

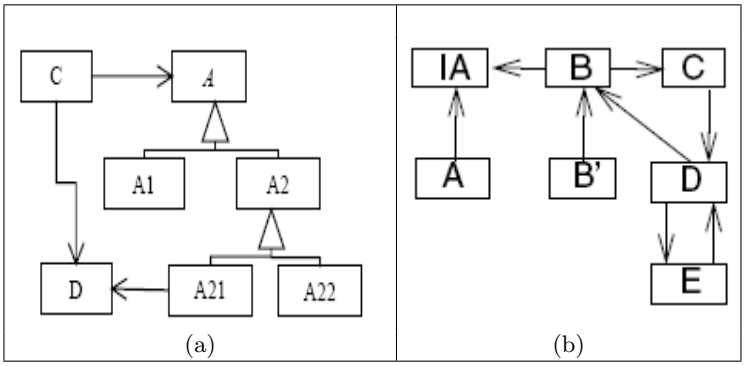

Fig. 2. Class interaction and Cycles

cycle) should be avoided either by refactoring or by using class interfaces [24, 3]. Several studies [6, 31] focused on detecting and assessing the dependency. Jungmayr in [24] introduced a set of metrics to measure the dependability and its influence on the testability.

4 Quantitative Analysis

4.1 Introduction

Our study aims at studying several open source applications in order to detect and analyze the testability antipattern occurrences, and especially the class cycles. We were motivated by several questions:

- Is it frequent to find cycles in software?
- How many classes could be found in a cycle?
- What is the size of smallest/ biggest cycle?
- Is there any elements that favor the occurrence of cycles? which ones?

To answer these questions we analyzed different applications presented in Table 1. This study is carried out on data collected from 14 open-source Java applications. They represent approximately 196 packages and more than 4100 classes and interfaces. They were chosen arbitrarily on the web mainly from sourceforge, Table 1 gives small description for these applications, and their sizes expressed by the number of packages and classes.

We have used Classycle's Analyser [1], an Eclipse plugin, to analyse the class dependencies in Java applications.

[1] http://classycle.sourceforge.net/

Application	Description	# packages	# classes
1-NanoXML[(1)]	small XML parser for Java	1	19
2-Chemical Evaluation Framework 1.001[(*)]	software to assist in hazard assessment	3	127
3-Java Groups-2.5.0[(*)]	group communication based on IP multicast	23	918
4-Jsxe[(*)]	Java simple XML editor	25	453
5-XMLMath[(*)]	XML-based expression evaluator	2	80
6-HTMLCleaner[(*)]	Transforms HTML a into a well-formed XML	1	27
7-KoLmafia-v12.3[(*)]	A interfacing tool with online adventure game	27	740
8-MegaMek-v0.32.2[(*)]	A networked Java clone of BattleTech	32	811
9-weirdx-1.0.32[(*)]	A pure Java X Window System server	3	108
10-Java Gui Builder0.6.5a[(*)]	Decouple GUI code from the rest of application	19	163
11-Bluepad v0.1[(*)]	Turns cellphone to a remote PC controller	5	11
12-Jaxe[(*)]	Java XML editor	7	178
13-JDom1-1[(4)]	Access to XML data	7	68
14-JFreeChart-1.0.9[(*)]	Create Charts	41	475
All	-	196	4178

(1) `http://nanoxml.cyberelf.be`; (4) `http://www.jdom.org/`;
(*) `http://www.sourceforge.net/` ;

Table 1. Data source

4.2 Data Analysis

The goal of this analysis was identifying the class cycles that appear in each application. Table 2 shows the frequency of different cycle sizes, expressed also by a percentage, the maximum cycle size, and the minimum cycle size. From this table one can observe that the number of cycles of size 2 and 3 represent 48.49 and 20.1 respectively of the total number of cycles of all sizes, while the cycles that have a size greater than 21 smaller than 30 represent 1.26% of the total number of cycles.

Then we tried to discover if there is any specific code structure that causes this observation. For that we studied the structure of the classes the make a part of the cycles. Table 3 shows the total number of classes related to all cycles in an application, the number of inner classes[2], and the percentage of inner classes in a cycle to the total classes in it. From this table one can notice that more than 50% of classes that belong to cycles are inner classes. Actually, the cycles of size n are less complex than cycles of size m, where $m > n$, but the total complexity caused by the cycles could be reduced if we restrict the number of the inner classes, and refactor the cycles of size 2 and 3 which represent about 70% of all classes.

Application/Cycle size	2	3	4	5-10	11-20	21-30	>30	#Max	#Min
CEF	1	4	1	3	0	1	1	38	2
Bluepad	0	0	0	1	0	0	0	10	10
HtmlCleaner	0	0	0	0	0	1	0	21	21
JavaGUIBuilder	6	4	1	1	0	0	0	6	2
Jaxe	6	2	0	0	0	0	1	132	2
Jdom	2	4	1	0	0	1	0	24	2
JfreeChart	20	5	1	2	1	0	1	36	2
Jgroup	91	36	24	33	4	0	1	53	2
JSXE	12	2	2	5	1	1	4	129	2
KoLmafia	13	4	4	3	0	1	1	598	2
MegaMek	38	18	3	11	3	0	4	101	2
Weirdx	2	1	0	0	0	0	1	58	2
NanoXML	2	0	0	0	0	0	0	2	2
XmlMath	1	0	1	0	0	0	0	4	2
Total	193	80	38	59	9	5	14	-	-
Percent	48.49	20.1	9.55	14.82	2.26	1.26	3.52	-	-

Table 2. The frequency of cycles of different sizes

[2] An inner class is a class that is defined in another class, it could be declared also inside the body of a method (either with name or without name).

Application	#Classes in all cycles	#Inner Classes	%Inner classes
CEF	105	91	86.67
Bluepad	10	0	0
HtmlCleaner	21	0	0
JavaGuiBuilder	34	22	64.71
Jaxe	150	84	56
JDom	44	13	29.55
JFreeChart	121	22	18.18
JGroup	702	465	66.24
JSXE	361	209	57.89
KoLmafia	703	396	56.33
MegaMek	571	321	56.22
NanoXML	2	1	50
Weirdx	65	10	15.38
XMLMath	6	1	16.67
TOTAL	2895	1635	56.48

Table 3. The percentage of inner classes

5 Conclusions and Perspectives

Testability is a software factor that could be used to detect the different weaknesses that could increase the difficulty of the test. Several metrics have been proposed to evaluate the testability of object oriented programs. Certain metrics were defined at representation facet other were defined at implementation facet.

Antipatterns are weaknesses that reduce the testability of a software. The antipatterns could be detected early in the life cycle of software development. The goal of our quantitative analysis was to find if there is any relationship between the occurrence of anitpatterns and the characteristics of the code.

This study based on 14 open-source Java applications. The results presented here show that about 50% of cycles are of size 2, and 20% of cycles are of size 3. On the other hand the analysis showed that 56.48% of classes that belong to cycles are inner classes. As a result about 70% of cycles could be avoided if one limits the use of cycles of size 2 and 3. And one can reduce the size of the cycles by avoiding the number of inner classes that present more than 55%.

For future work, we are looking for other structural elements that could increase the occurrence of the cycles i.e inheritance, complexity.

References

1. R. Bache and M. Mullerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, 5(2):86–92, 1990.

2. Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.
3. B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. Measuring and improving design patterns testability. In *9th IEEE International Software Metrics Symposium (METRICS 2003)*, pages 50–, Sydney, Australia, September 2003.
4. Benoit Baudry and Yves Le Traon. Measuring design testability of a uml class diagram. *Information & Software Technology*, 47(13):859–879, 2005.
5. Benoit Baudry, Yves Le Traon, and Gerson Sunyé. Testability analysis of a uml class diagram. In *8th IEEE International Software Metrics Symposium (METRICS 2002)*, pages 54–, Ottawa, Canada, June 2002.
6. Antonia Bertolino and Lorenzo Strigini. On the use of testability measures for dependability assessment. *IEEE Trans. Software Eng.*, 22(2):97–108, 1996.
7. J. M. Bieman and C. Izurieta. Testing consequences of grime buildup in object oriented design patterns. *First International Conference on Software Testing ICST 2008*, To be published in april 2008.
8. R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.
9. Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.
10. Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *ICSE*, pages 345–354, 1999.
11. Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001.
12. M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006.
13. S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, pages 197–211, 1991.
14. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
15. Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.
16. Fernando Brito e Abreu and Walcélio L. Melo. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*, pages 90–99, 1996.
17. Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.
18. Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001.
19. R. S. Freedman. Testability of software components. *IEEE Trans. Software Eng.*, 17(6):553–564, 1991.
20. A. Goel, S. C. Gupta, and S. K. Wasan. Controllability mechanism for object-oriented software testing. In *10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, pages 98–107, Chiang Mai, Thailand, December 2003. IEEE Computer Society.

21. A. Goel, S. C. Gupta, and S. K. Wasan. Probe mechanism for object-oriented software testing. In *6th International Conference Fundamental Approaches to Software Engineering (FASE 2003), Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003*, volume 2621 of *Lecture Notes in Computer Science*, pages 310–324. Springer, April 2003.
22. Suresh C. Gupta and Mukul K. Sinha. Improving software testability by observability and controllabliity measures. In *IFIP 13th World Computer Congress*, pages 147–154, September 1994.
23. Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.
24. Stefan Jungmayr. Identifying test-critical dependencies. In *18th International Conference on Software Maintenance (ICSM*, pages 404–413. IEEE Computer Society, 2002.
25. J. D. McGregor and S. Srinivas. A measure of testing effort. In *Second USENIX Conference on Object-Oriented Technologies (COOTS)*, 1996.
26. S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information & Software Technology*, 47(15):979–997, 2005.
27. Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. Technical report, IEEE, New York, USA, 1990.
28. Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An empirical study on object-oriented metrics. In *6th IEEE International Software Metrics Symposium (METRICS'99)*, pages 242–249, Boca Raton, FL, USA, November 1999. IEEE Computer Society.
29. J.M. Voas and K. Miller. Semantic Metrics for Software Testability. *J. Systems Software*, 20:207–216, 1993.
30. Y. Wang, G. King, I. Court, M. Ross, and G. Staples. On testable object-oriented programming. *SIGSOFT Softw. Eng. Notes*, 22(4):84–90, 1997.
31. Hong Yul Yang, Ewan D. Tempero, and Rebecca Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, pages 212–221. IEEE Computer Society, 2005.
32. Ping Yu, Tarja Systä, and Hausi A. Müller. Predicting fault-proneness using oo metrics: An industrial case study. In *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, Budapest, Hungary, March 2002. IEEE Computer Society.
33. Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Software Eng.*, 32(10):771–789, 2006.

Evaluating Quality-in-Use Using Bayesian Networks

M.A Moraga[1], M.F. Bertoa[2], M.C. Morcillo[3], C. Calero[1], A. Vallecillo[2]

[1]Alarcos Research Group – Institute of Information Technologies & Systems.
Dept. Information Technologies & Systems – Escuela Superior de Informática.
Universidad de Castilla-La Mancha, Spain
[2]Dept. Lenguajes y Ciencias de la Computación. Universidad de Málaga, Spain
[3]Dept. Estadística e Investigación Operativa. Universidad de Málaga, Spain

Abstract. This paper challenges the traditional approach for assessing the overall quality of a software product, which is based on the assumption that, in ISO/IEC 9126 terms, a good external quality ensures a good quality-in-use. Here we change the focus of the quality assessment, concentrating on the quality-in-use as the driving factor for designing a software product, or for selecting the product that better fits a user's needs. We propose a "backwards" analysis of the relationship between the external quality and the quality-in-use which tries to determine the external quality sub-characteristics that are really relevant to ensure the required level of quality in a given context of use, in order to avoid superfluous costs or irrelevant features – which may unnecessarily increase the final price of the product. In this paper we propose Bayesian Belief Networks to model such relationships, and propose a method to build them for different contexts of use.

Introduction

Assesing the quality of a software product is, in general, a complex and difficult task. One approach to simplify this evaluation process consists of breaking down the product quality into several individual factors or *characteristics*. One of the most widespread proposal that uses this approach is the one defined in ISO/IEC 9126 [1]. This standard proposes three levels at which the quality of a software product can be observed: *internal*, *external* and *in-use* (Fig. 1).

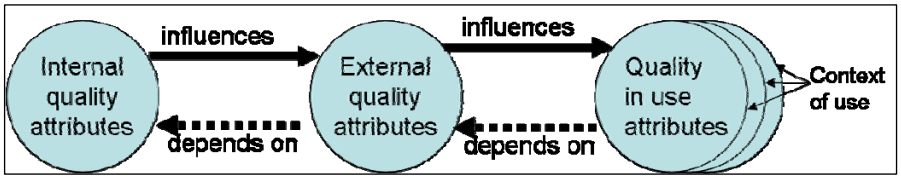

Fig. 1. Relations between qualities views, based on ISO/IEC 9126 [1].

Internal quality utilizes a "white box" view of the software product, mainly related to its static properties, and evaluated during the design and development phases. External quality uses a "black box" view, assessing the software product from its external attributes and features, e.g., using its dynamic properties during execution, which

can be evaluated running the code in a testing environment. Finally, quality in use (QiU) is the end-user viewpoint when the software product is used in its working environment, to carry out the specific tasks that the final user needs to perform. There is also a strong relationship between them, as Fig. 1 shows.

So far, the emphasis has been put on assessing the internal and external quality of software products (see, e.g., [2–8]), mainly because of these two views are more precisely defined in the literature, can be more easily identified (and are thus easier to evaluate), and also because of the "dependency" of the QiU on the external quality (EQ), as ISO/IEC 9126 states (Fig. 1).

This relationship between the external and in-use quality of the software product seems to be based on the assumption than having a product with a high (external) quality will guarantee a product with a high QiU. However, this is not necessarily true in most situations: we all know that a product with the best (external) quality does not necessarily guarantee that the product will fulfil the user's needs in its context of use. Especially when the overall quality (as perceived by the end-user) is composed of many conflicting factors; e.g., a Ferrari is not the best car to use to go to work if your job is social assistant in a deprived suburb in the outskirts of New York.

In this paper we completely change the focus of the quality assessment, concentrating on the quality in use as the driving factor to consider when designing a software product, or when selecting the product that better fits a user's needs.

There are several reasons that have moved to challenge the traditional approach to evaluating the quality of a software product. Firstly, not all the external quality characteristics of a software product have the same influence on its QiU. This, together with the false assumption about the direct dependency between the external quality and the QiU mentioned above, forces many times to over-specify some of the product aspects (which are non critical for the end user), for the sake of ensuring a certain level of QiU. This unnecessarily increases the costs and development efforts, without a direct effect on the advantages that the end-user perceives. Secondly, we have realized that users normally tend to perceive as equally important all the external quality characteristics of a software product when asked about them in the abstract, i.e., without any concrete context-of-use in mind. However, when the context-of-use of a software product is fixed, users are able to tell apart the product characteristics that really matter from their point view, from those that are desirable, but not critical.

Therefore, in this paper the focus is on the QiU, and we also analyze the relationship between the external quality and the QiU of a software product in the opposite direction as it has been traditionally done. Thus, we propose a "backwards" analysis (we start with a given level of QiU and we want to determine the minimum level of external quality that guarantees such a desired quality in use), as opposed to the traditional "forward" analysis (by which we try to determine the level of QiU of a software product, given a measured level of external quality). Actually, to obtain a (good) level of quality in use, the goal would be to be able to select the reduced set of *really relevant* external quality sub-characteristics that ensure the required level of quality, focusing just on them in order to avoid superfluous costs or irrelevant features which may unnecessarily increase the final price of the product.

Using Bayesian networks for representing the QiU-EQ relationships, in this paper we analyse the different possibilities to model such relationships, and propose an approach that can be used for different contexts of use.

2. The ISO/IEC 9126 Quality Model

ISO/IEC 9126 [1] defines one of the most widely used quality models, which breaks down the overall quality of a software product in three views (internal, external and in-use) and defines each one in terms of a set of characteristics. These characteristics are further refined into several sub-characteristics in order to conduct more precise assessment analysis. Measures are used to evaluate each sub-characteristic.

ISO/IEC 9126 defines 6 characteristics for the internal and external quality: *Functionality*, *Reliability*, *Usability*, *Efficiency*, *Maintainability* and *Portability*. Please note that they are the same because the standard defines a parallelism between them, assuming that there is a one-to-one dependency relationship (e.g., internal understandability influences external understandability and vice-versa). ISO/IEC 9126 does not define the same level of parallelism between external quality and QiU, though. In fact, ISO/IEC 9126 defines only four QiU characteristics, *Effectiveness*, *Productivity*, *Safety* and *Satisfaction*, without breaking them down into any sub-characteristics.

ISO is currently working on the preparation of the new SQUARE (Software QUality REquirements) family of standards, which will replace the ISO/IEC 9126 series. However, SQUARE is not mature yet. Actually, there is not even a consensus yet in the Working Committee about the different views of quality they want to distinguish. Therefore, we decided to base our proposal on ISO/IEC 9126, waiting for the SQUARE proposal to stabilize.

3. Bayesian Belief Networks

In order to determine the relationship between the QiU and the external quality, we need statistical methods and tools that can carry out backward analysis. In fact, commonly used linear regression (LR) or principal component analysis (PCA) are not useful here because they conduct forward analysis and need initial numerical information at data level, which is difficult to obtain in many cases. However, Bayesian Belief Networks (or, simply, Bayesian Networks, BNs) can be very useful here. A BN is a directed acyclic graph, whose nodes are the uncertain variables and the edges are the casual or influential links between variables. A Conditional Probability Table (CPT) is associated with each node to denote such causal influence. [9].

To define a BN we need to: (1) provide the set of random variables (nodes) and the set of relationships (causal influence) among those variables; (2) build a graph structure with them; and (3) define conditional probability tables associated with the nodes. These tables determine the weight (strength) of the links of the graph and are used to calculate the probability distribution of each node in the BN.

In this framework it is possible to:

- Represent the relationships between the sub-characteristics of the external quality and the QiU characteristics.
- Incorporate subjectivity in the evaluation criteria, and also uncertainty when regrouping several decision criteria into one single criterion (based on the experts' judgments).

Thus, using BNs we can model the different relationships between the characteristics and sub-characteristics of the external quality and the QiU, as well as the degree of dependence or influence between them. This will require defining the structure and conditional probability tables where the uncertainty relationships are reflected among the BN nodes (characteristics) we want to build. Once the network is defined, it is necessary to train it through a set of controlled experiments, so that it "learns".

Furthermore, the trained network can be used to make inferences about the values of the variables in the network. Bayesian propagation algorithms use probability theory to make such inferences using the information available (usually a set of observations or evidences). Such inferences can be *abductive*, if we want to determine the external quality sub-characteristics we must consider to guarantee a required level of QiU (the cause that better explains the evidence); or *predictive*, if we want to determine the probability of obtaining certain results in the future. Thus every variable of the network can be used either as a source of information or as an object of prediction, depending on the evidence available and on the goal of the diagnostic process.

Fig. 2. Characteristics and sub-characteristics of external quality (ISO/IEC 9126)

Fig. 3. QiU characteristics (ISO/IEC 9126)

4. Using BNs for evaluating Quality in Use

Our working hypothesis is that the EQ has influence on the QiU and this influence can be modelled and studied through a BN, in such a way that we can conduct backwards analysis on the required level of EQ to ensure a given level of QiU. In the first place, we need to model the relationships between the external quality characteristics and its sub-characteristics according to the ISO/IEC 9126 quality model. This is shown in Figure 2.

Secondly, we have to model the QiU and its four characteristics using a Bayesian Network (remember that ISO/IEC 9126 does not define sub-characteristics for QiU).

The corresponding BN is shown in Fig. 3. Finally, we need to link these two networks in order to define the relationships we are looking for. There are at least two possible approaches for defining this connection between the two BNs, taking into account their hierarchical structure and the construction rules for Bayesian networks, which are described in the following sections.

4.1. Using the influence of EQ characteristics on QiU characteristics

The relationship between EQ and QiU can be modelled by determining the characteristics of the former which affect the characteristics of the latter. To build the Bayesian network we have studied the relationships between the different characteristics, identifying the EQ characteristics that have a significant influence on the QiU characteristics. Table 1 shows these relationships using a matrix, where the "X" indicates a relationship between those characteristics. This table allows us to build the BN shown in Fig. 4.

Table 1. Relationships between characteristics of external quality and QiU.

		Quality in Use			
		Safety	Satisfaction	Productivity	Effectiveness
External Quality	Functionality	X	X	X	X
	Reliability	X	X		X
	Usability		X		X
	Efficiency		X	X	
	Portability		X		
	Maintainability	X			

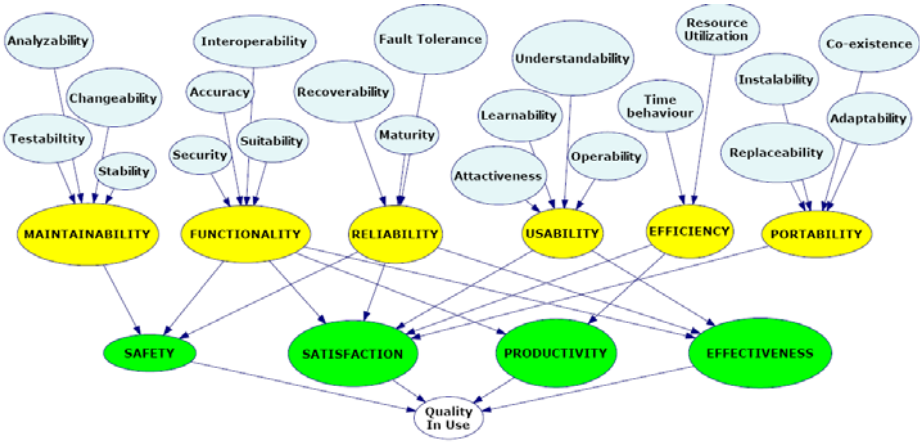

Fig. 4. BN showing the influence of EQ characteristics on QiU characteristics

Of course, the level of influence (indicated by the relationships in Table 1) may vary between different application domains. The relationships shown in Table 1 might then need to be tailored to other domains. In any case, the method indicated in this paper is still valid.

The next step in the construction of the BN is defining the probability tables. We have used the 3 categories defined by IEEE International Standard 1061 [10]: *Acceptable*, *Marginal* and *Unacceptable*. We assume that the *Marginal* value is used only

when we cannot clearly assess the feature as acceptable or unacceptable. Somehow, it represents the permissible error that we make when evaluating a feature, or when we are uncertain about it.

The tables of the upper level nodes have a single entry with the evaluation of the external quality sub-characteristics. The problem we face here is that in order to define indicators for them we need to consider the opinion of experts, who tend to request an *Acceptable* value for most of them (otherwise they fear that the product will not be accepted by users). This is the problem we mentioned at the beginning about considering a product *in the abstract*, i.e., without a given context-of-use in mind.

The probability tables of the nodes in the next level, corresponding to the EQ characteristics, have been defined taking into account the degree of influence (i.e., weight) that each one has on the corresponding QiU characteristic. *Unacceptable* values should be weighed differently, because if a sub-characteristic does not reach the required level of quality then its characteristic will be hardly *Acceptable*. (*Unacceptable* sub-characteristics are not compensated by *Acceptable* ones.)

In summary, we have identified two factors that influence the definition of these probability tables: (1) the weight we give to each sub-characteristic; and (2) the greater weight that needs to be given to the unacceptable values. With these assumptions, we have built the 6 probability tables for external quality characteristics. Nodes in the third level represent the QiU characteristics, and are built similarly, using the relationships defined in Table 1.

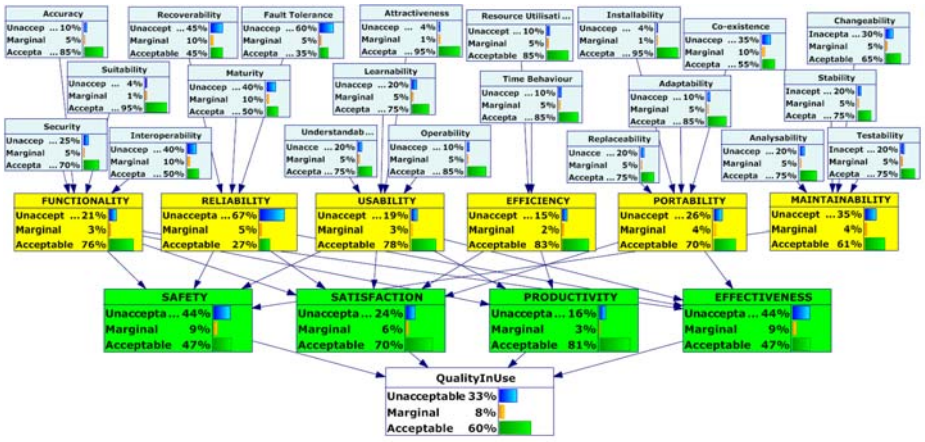

Fig. 5. Quality in Use Bayesian Network, Bar Chart format

The last step is to build the probability table for QiU (the single node in the 4th level), which depends on its four characteristics. In this step, we have studied if all four characteristics have a similar weight (influence) and, as in previous tables, we have assigned a greater weight to unacceptable values. Fig. 5 shows the proposed Bayesian network in a bar chart form, which depicts the values for all nodes.

To illustrate this approach, let us suppose that we want to have an *Acceptable* (100%) level of QiU. Using the BN it is possible to go backwards and get the level for each characteristic of the QiU and the external quality that ensure that value. In this example, we get very high values (> 85%) for *Satisfaction* and *Productivity*. Similar-

ity, we can study the nodes of the first and second tiers and we can see that the sub-characteristics with higher values (>90%) are: *Suitability*, *Attractiveness* and *Instalability*. This can be a clear sign that we should pay close attention to the processes and factors that have influence on these sub-characteristics.

4.2 Using the influence of EQ sub-characteristics on QiU characteristics

Due to the way in which the BN shown in Fig. 5 is built, all sub-characteristics of an external quality characteristic have the same influence on all related QiU characteristics. For example, *Functionality* is related to all four QiU characteristics (see Table 1). Thus, all its sub-characteristics (*Suitability*, *Accuracy*, *Interoperability* and *Security*) have influence on them. But, is this actually true? Is *Suitability* as "important" for *Safety* as for *Satisfaction*? The answer is, of course, No. The problem is that Table 1 was built abstracting away the individual influence of each EQ sub-characteristic on the QiU characteristics. Therefore, using this Bayesian network, we are forcing that whenever a external quality characteristic is linked with a QiU characteristic, then **all** its sub-characteristics have to have some influence on that QiU characteristic. Thus, it is not possible to define different probability tables for one EQ characteristic depending on the QiU characteristic.

Table 2. Relationships between EQ sub-characteristics and QiU characteristics.

			Quality in Use			
	Characteristic	Subcharacteristic	SAFETY	SATISFACTION	PRODUCTIVITY	EFFECTIVENESS
External Quality	Functionality	Suitability		X	X	X
		Accuracy	X	X	X	X
		Interoperability			X	
		Security	X			
	Reliability	Maturity	X	X		X
		Recoverability	X	X		
		Fault Tolerance	X	X		
	Usability	Understandability		X		X
		learnability		X		X
		Operability		X		X
		Attractiveness		X		
	Efficiency	Time behaviour		X	X	
		Resource Utilization			X	
	Portability	Adaptability		X		
		Instalability		X		
		Coexistence		X		
		Replaceability		X		
	Maintainability	Analyzability	X			
		Changeability	X			
		Stability	X			
		Testability	X			

Table 2 shows such refined relationships between EQ sub-characteristics and QiU characteristics. Thus we can see that *Resource Utilization* only affects *Productivity*; while *Time Behaviour* influences on *Productivity* and *Satisfaction.*

Based on this new table, a second BN was built, which reflects the direct relations between EQ sub-characteristics and QiU characteristics (Fig. 6). In this case we have removed the nodes corresponding to the EQ characteristics, considering that EQ sub-characteristics will directly influence the QiU characteristics.

Although this approach is more precise, it produces a very high number of entries on the nodes that represent the QiU characteristics. Therefore, the definition of the

probability tables is very laborious and cumbersome. For instance, tables with 11 entries such as *Satisfaction* will have 3^{11} (i.e., more than 175,000) columns.

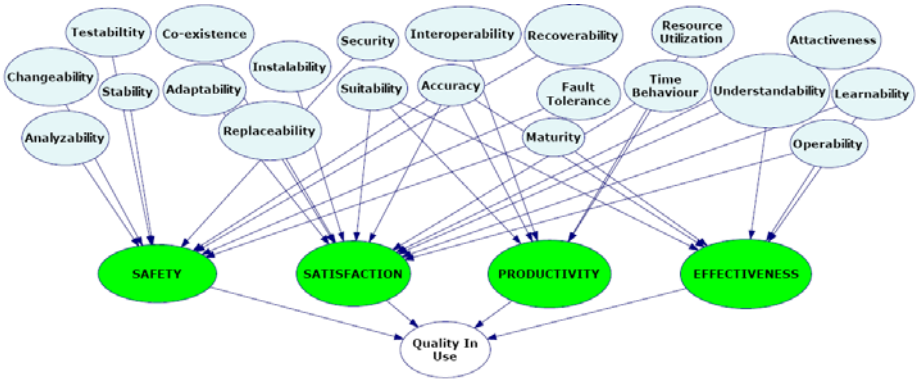

Fig. 6. BN with the influence of EQ sub-characteristics on QiU characteristics.

A common practice to simplify the relationships in BN is based on the introduction of *synthetic* nodes. In this case we decided to introduce a synthetic node among sub-characteristics of each EQ characteristic and each QiU characteristic. The entries of each of these intermediate nodes will be only those EQ sub-characteristics that affect the corresponding QiU characteristics (see Fig. 7). This is not required if there is just one influencing sub-characteristic, as it happens in the case of *Maturity* (*Effectiveness*) and *Time Behaviour* (*Satisfaction*). The most complex case occurs in the *Usability* node, for which four synthetic nodes are needed, one for each QiU characteristic: FUNCT_SAFE, FUNCT_SAT, FUNCT_PRO and FUNCT_EFF.

An interesting case also happens in the *Accuracy* sub-characteristic. It has influence on the four QiU characteristics, and by means of the intermediate nodes we are able to represent its independent influence of each one. Finally, *Security* is only related to *Safety* and then will not be affected by the rest of QiU characteristics as it happened in the previous network.

The obtained BN (Fig. 7) drastically reduces the number of entries in the probability tables. In addition, it avoids the undesirable influence of EQ sub-characteristics on the QiU characteristics when they do not have a direct relationship. Then, in this network a finer granularity is achieved while avoiding inconsistent relations.

As a disadvantage, the concept of external quality characteristic is lost in this new approach. However, please note that this drawback can be easily overcome if we make use of some of the properties and operations of BNs. In fact, we could easily create four individual BNs, one for each QiU characteristic, and then combine them to form the complete Bayesian Network. These four BNs are shown in Fig. 8.

By using this model it is possible to study each QiU characteristic independently, being able to observe more easily the influence of EQ sub-characteristics on each QiU characteristic. This approach not only facilitates the definition of the probability tables, but also the use of the network because of the smaller number of nodes. Once these BNs are built, they need to be combined into the global one to study the overall QiU in that context of use, using the union operation defined for BNs [9].

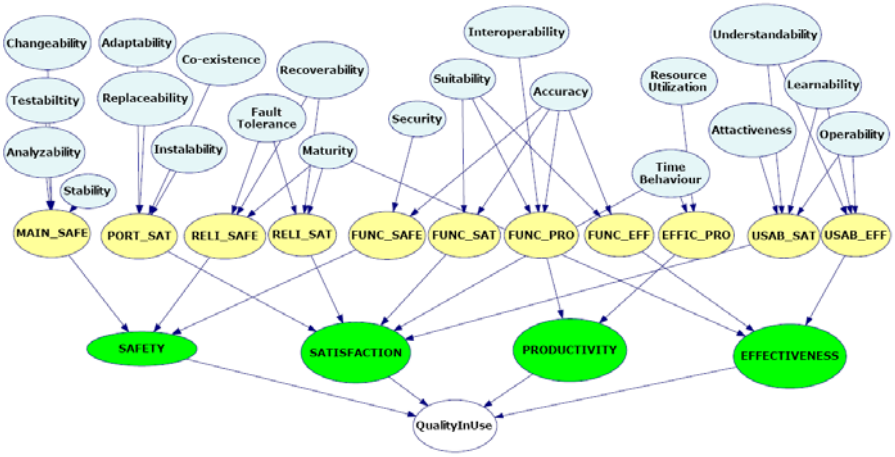

Fig. 7. The BN of Fig 6, with synthetic nodes.

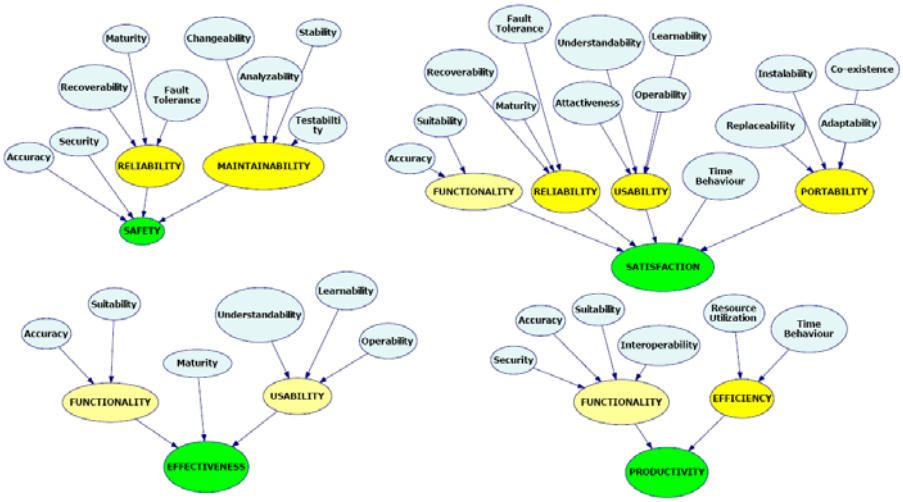

Fig. 8. The 4 BNs for the individual QiU characteristics.

5. Conclusions and lessons learnt

In this paper we have shown how to build a BN for determining the influence of EQ sub-characteristics on the overall QiU of a software product. Fenton et al. [8] have also used BNs to predict some EQ characteristics from the internal quality of software products, but without considering the QiU, and using a “forward” prediction process. Our aim is, however, to focus on the QiU and use a backwards analysis for predicting the minimum acceptance levels for EQ characteristics that ensure the required QiU.

Of course, this is just the first step of a more complete line of work that aims at providing users of given contexts of use with tool support for evaluating software products (including, e.g., the selection of those products that better suit their needs with the less possible costs).

There are several activities that we plan to address in the short term. Firstly, we want to empirically validate the proposal by exercising the BN in several contexts of use, adapting the BN to the specific peculiarities of each context, and checking that it learns as it should for these contexts. The final results of these tests will be a set of trained BNs that we expect to be useful in these environments. The way to validate the results (conducting some experimental validation exercises) is a research activity itself. Furthermore, and as we mentioned in Section 4.1, the table that defines the relationships between the EQ and QiU characteristics may need to be tailored depending on the particular application domain. However, we expect the changes produced by such customizations to be minimal.

Secondly, we also want to refine this approach by combining it with some Principal Component Analysis during the initial definition of the BN, whereby we can confirm (and refine) the relationships defined in Table 2 between the EQ subcharacteristics and the QiU characteristics.

Finally, we expect to provide a useful input to the ISO Working Group defining the new SQUARE family of standards based on our researches, not only confirming the existence of the influence of EQ on QiU, but also quantifying it in some specific contexts of use; and, more importantly, highlighting the primary role that QiU plays in the evaluation of the quality of any software product as the driving force of the rest of the quality views.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work has been funded by the following Research Projects: MOVIS (P07-TIC-03184), IVISCUS (PAC08-0024-5991), VIASCO (PET2006-0682-00), CALIPSO (TIN2005-24055-E) and TIN2005-09405-C02-01.

References

1. ISO/IEC 9126. Software Engineering-Product Quality. Parts 1 to 4, 2001.
2. Bertoa, M.F., Troya, J.M. and Vallecillo, A. “Measuring the usability of software components", Journal of Systems and Software 79:(3):427-439, Mar. 2006.
3. Botella, P., X. Burgués, J. P. Carvallo, X. Franch, J. A. Pastor and C. Quer. “Towards a Quality Model for the Selection of ERP Systems.” In *Component-Based Software Quality*, pp. 225-245, Springer-Verlag, 2003.
4. Dromey, R. G. “A Model for Software Product Quality.” IEEE Transaction on Software Engineering 21(2):146-162, 1995.
5. Jagdish, B.. “A Hierarchical Model for object-oriented Design Quality Assessment.” IEEE Transaction on Software Engineering 28(1):4-17, 2002.
6. Liu, K., S. Zhou and H. Yang. “Quality Metrics of Object Oriented Design for Software Development and Re-Development.” In Proc. of APAQS’00, pp. 127-135, 2000.
7. Moraga, M. Á., C. Calero and M. Piattini. “Comparing different quality models for portals.” Online Information Review 30(5):555-568, 2006.
8. Neil, M., P. Krause, and N. E. Fenton. “Software Quality Prediction Using Bayesian Networks.” In *Software Engineering with Computational Intelligence*, Chapter 6, Kluwer, 2003.
9. Jensen F.V. Bayesian Networks and Decisions Graphs. Springer-Verlag, 2001.
10. IEEE Std 1061-1998, “IEEE Standard for a Software Quality Metrics Methodology” 1998.

Metrics for Analyzing the Quality of Model Transformations

M.F. van Amstel[1], C.F.J. Lange[2], M.G.J. van den Brand[1]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{M.F.v.Amstel|M.G.J.v.d.Brand}@tue.nl
[2] Federal Office for Information Technology
Barbarastraße 1, 50735 Cologne, Germany
mail@christian-lange.com

Abstract. Model transformations become increasingly important with the emergence of model driven engineering of, amongst others, object-oriented software systems. It is therefore necessary to define and evaluate the quality of model transformations. The goal of our research is to make the quality of model transformations measurable. This position paper presents the first results of this ongoing research. We present the quality attributes we have identified thus far and a set of metrics to assess these quality attributes.

1 Introduction

Model driven engineering (MDE) is a software engineering discipline that focuses on models for the development of software. MDE combines domain-specific modeling languages for modeling software systems and model transformations for synthesizing them [1]. Model transformations thus become more and more important. An example of a model transformation is adding getters and setters to a UML class diagram. Similar to other software engineering artifacts, model transformations have to be used by several developers, have to be changed according to changing requirements and should preferably be reused. Because of the prominent role of model transformations in today's and future software engineering, there is the need to define and assess their quality. Quality attributes such as modifiability, understandability and reusability need to be understood and defined in the context of MDE, i.e., for model transformations.

The goal of our research is to make the quality of model transformations measurable. We therefore start by defining the meaning of several quality attributes in the context of model transformations. We plan to do this by creating a quality model specific for model transformations (similar to the general software quality model described by Boehm et al. [2]). This model is a hierarchical decomposition of a number of quality attributes. We propose a set of metrics for assessing these quality attributes. Metrics have been studied extensively to assess the quality of (object-oriented) software [3,4,5] and software designs [6]. Some of the metrics

defined in earlier studies can be adapted such that they can be used to measure certain aspects of model transformations. We will also define new metrics that are specific for model transformations.

This paper presents the first results of our research on the quality of model transformations. In this paper we focus on model transformations created using the ASF+SDF [7,8] term rewriting system, but we expect that our techniques can be applied to model transformations created using different transformation engines as well. An example of a model transformation created using ASF+SDF can be found in [9]. In this paper we present the quality attributes we have identified thus far and a set of metrics to assess these quality attributes.

The remainder of this paper is structured as follows. Section 2 shortly explains the term rewriting system ASF+SDF. In Section 3 we describe the quality attributes we identified as applicable to model transformations. The metrics we propose to assess these quality attributes are described in Section 4. In Section 5 the metrics are related to the quality attributes. Section 6 contains the conclusions of our initial investigation and gives some directions for further research.

2 A Short Introduction to ASF+SDF

In this paper we consider quality attributes of model transformations defined using the term rewriting system ASF+SDF. One of the main applications of ASF+SDF is transformations between languages. These transformations are performed between languages specified in the syntax definition formalism SDF using conditional equations specified in the algebraic specification formalism ASF. The two main advantages of ASF+SDF are its modularity and the syntax-safety it guarantees. Syntax-safety in the context of model transformations means that every syntactically correct source model is transformed into a syntactically correct target model. It is impossible to transform syntactically incorrect source models using ASF+SDF.

A model transformation in ASF+SDF conceptually consists of multiple transformation functions. Transformation functions transform language elements from the source language into language elements of the target language. A transformation function is defined by signatures and equations. The signatures of a transformation function consist of the name of the transformation function, followed by a list of arguments and a return value. Signatures are defined in SDF. Apart from function signatures, SDF is also used to define variables. An example of a function signature and variable definition in SDF is depicted in Figure 1. Every signature can have equations which form the implementation of a transformation function. These equations have to conform to their signature. An equation can have zero or more conditions. These conditions can be used, amongst others, to assign values to variables. Equations are defined in ASF. An example of a function implementation with one condition (line 3) in ASF is depicted in Figure 2.

```
context-free syntax
  transform(Attribute) -> Return_value
variables
  "$Attribute"    -> Attribute
  "$Return_value" -> List[[Attribute]]
```

Fig. 1. Function signature and variable definition in SDF

```
equations
[transform-1]
  $Return_value := [$Attribute]
  ====>
  transform($Attribute) = $Return_value
```

Fig. 2. Function implementation in ASF (equations)

3 Quality Attributes

This section contains a description of the quality attributes that we have identified as relevant for model transformations thus far. Most of these quality attributes can be applied to software artifacts in general. Therefore we mention their relevance for model transformations in particular.

We plan to create a quality model specific for model transformations similar to Boehm's general software quality model described in [2].

Understandability The amount of effort required to understand a model transformation. Understandability is related to modifiability and reusability. The easier it is to understand a model transformation, the easier it is to modify or reuse. Since a model transformation is defined on a source and target (meta)model, their syntax and semantics should also be well understable to understand a model transformation.

Modifiability The extent to which a model transformation can be adapted to provide different or additional functionality. The main reason for modifying a model transformation is changing requirements. Another reason is that the (domain specific) language in which the source and/or target model are described may be subject to changes. Modifiability captures the amount of effort needed to modify a model transformation such as to deal with changes in either the requirements, or the source or target metamodel.

Reusability The extent to which (a part of) a model transformation can be reused by other model transformations. Reusability refers to as-is reuse. Therefore it is different from modifiability, which refers to modifying a model transformation. Reusability is especially relevant for model transformations when a source model has to be transformed into different target models, or vice versa.

Reuse Reuse is the counterpart of reusability. Reuse is the extent to which a model transformation reuses parts of other model transformations. We consider this as a quality attribute since it is good practice to reuse tested units.

MDE advocates reuse of models, i.e., a model is reused throughout the development process and different artifacts are generated from a source model by performing model transformations. Since model transformations in an MDE setting have the same source model as starting point, it is to be expected that the first few transformation steps in these model transformations are similar. So, reuse could be a measure for how well a model transformation adheres to the MDE paradigm.

Modularity The extent to which a model transformation is systematically structured. With systematically structured we mean that every module in a model transformation should have its own purpose. Modularity is related to reusability. If functionality is well spread over modules it is more likely that parts of it can be reused for other model transformations. Therefore, the size of transformation steps is also an important aspect of modularity.

Completeness The extent to which a model transformation is fully developed. A model transformation is complete if it transforms a source model into a target model according to its specifications, i.e., all functionality has been implemented. An incomplete model transformation will result in an incomplete target model or no target model at all.

Consistency The extent to which a model transformation contains no conflicting information. Boehm [2] distinguishes two types of consistency: internal consistency and external consistency. Internal consistency refers to the extent to which a model transformation contains uniform notation. Internal consistency is related to understandability. Internal inconsistency may lead to inconsistencies in the target model. External consistency refers to the extent to which a model transformation adheres to its specification.

Conciseness The extent to which a model transformation does not contain superfluous information. Examples of superfluous information are code clones or unnecessary function parameters.

4 Metrics

This section contains the metrics we have defined for assessing the quality attributes for model transformations created using ASF+SDF. In [10] metrics are defined for SDF. Those metrics are applicable to language definitions, but we will focus on model transformations. ASF has the characteristics of a functional language. Therefore we were able to adapt metrics for functional languages, like the ones defined in [11], such that they can be applied to model transformations.

4.1 Size Metrics

The size of a model transformation can be measured in various ways. An obvious size metric is the *number of lines of code*. However, different programming styles may require different counting techniques which can lead to different measurements [12]. Therefore we propose to measure the size of a model transformation by counting the number of transformation rules. For ASF+SDF transformations this results in the following metrics: *number of functions*, *number of signatures*, and *number of equations*. Note that the number of signatures does not have to be equal to the number of transformation functions since transformation functions in ASF+SDF can be overloaded, i.e., a transformation function can have multiple signatures each having different argument lists or return values.

A model transformation usually consists of a domain-specific part and a domain-independent part, i.e., library functions. The proposed size metrics can be adapted to measure the size of the domain-specific (or domain-independent) part of a model transformation only.

4.2 Function Metrics

The size of a transformation function can be measured in different ways as well. Section 2 states that a transformation function has one or more signatures and that every signature has one or more equations. The size of a transformation function can be expressed in terms of its number of signatures or equations. Also, the size of the equations, defined as the number of conditions, can be included. This leads to three different metrics for measuring the size of a transformation function: *number of signatures per function*, *number of equations per function*, and *number of equations plus number of conditions per function*.

A measurement for the complexity of a transformation function is the average number of values it takes as arguments and the number of values it returns. These metrics are known as *val-in* and *val-out*. Note that an ASF equation can return only one value, but this can be a tuple consisting of multiple values.

Transformation functions generally depend on other transformation functions to perform their task. The dependency of a transformation function f on other transformation functions can be measured by counting the number of times function f uses other functions. The dependency of transformation functions on a transformation function f can be measured by counting the number of times function f is used by other functions. These metrics are similar to *fan-out* and *fan-in* as they are used to measure dependencies between components of software architectures.

4.3 Module Metrics

One of the main benefits of ASF+SDF is that it allows the creation of model transformations in a modular way. One aspect of the modularity of a transformation is the *number of (library) modules* that comprise the transformation.

A large number of modules is no guarantee for an understandable model transformation. The modules should be balanced in terms of size and functionality. The *balance of a module* can be measured by comparing the number of functions, signatures, and equations defined in that module with the average over all modules. Actually, we measure unbalance in this way.

In a similar way as for functions the dependency of modules on other modules can be measured. The dependency of modules on a module m can be measured by counting the *number of times module m is imported by other modules*. The dependency of module m on other modules can be measured by counting the *number of import declarations in module m*. Also, the fan-in and fan-out of a module can be measured. *Fan-in* is the number of times a function defined in module m is used by another function that is not defined in module m. *Fan-out* is the number of times a function defined in a module m uses a function that is not defined in module m. These metrics can be combined to measure the *complexity of the information flow* between modules as proposed in [13]:

$$\text{Information flow complexity}(M) = (\text{fan-in}(M) \times \text{fan-out}(M))^2.$$

In a similar way it is possible to combine the fan-in and fan-out metric of transformation functions to measure the information flow complexity of a function.

In general it is good practice to let every module of a model transformation have only one purpose, i.e., it should be concerned with one specific part of the transformation. This leads to a better balance of functionality among modules, and hence to a less complex model transformation. A module should thus contain one main transformation function and helper functions. We consider the function that is responsible for achieving the purpose of the module as the main transformation function. Note that helper functions are of course also transformation functions. If a module contains more than one main transformation function, the module should be split into multiple parts, each containing one main transformation function. Therefore, we propose to measure the *number of main functions per module*. This can be done by creating a call-graph of the module. A call-graph is a visual representation of the dependency of functions on each other. A vertex in the call-graph of a module represents a function defined in that module. A directed edge from vertex a to vertex b represents that the function represented by vertex a uses the function represented by vertex b. A main function f is a function which has only outgoing edges or incoming edges originating from f itself in the call-graph. This metric can also help to identify obsolete functions. A main function that is not used by any function from another module as well could be an unused function.

ASF+SDF enables the creation of parameterized modules. A parameterized module is similar to a generic class in C++. Examples of parameterized modules in ASF+SDF are the container modules list and table. These are generic lists and tables that can be parameterized such that they can contain elements of any type. Parameterized modules increase reusability. Therefore, we propose to measure the *number of parameterized modules* in a model transformation.

4.4 Consistency Metrics

A transformation function is defined as a set of signatures and associated equations. It is possible that there is a signature for a transformation function, but that there are no equations. This can happen for example when the model transformation is still under development. To detect this type of inconsistency, we propose to measure the *number of signatures without equations*. The other way around, i.e., equations without signatures, will be detected by ASF+SDF itself and therefore we will not introduce a metric for this inconsistency.

Usually variables are defined in a `hiddens` section of an SDF file. This means that they can only be used within the same module. This implies that a variable needs to be redefined if it is to be used in other modules. This may lead to inconsistencies because the same variable name in one module can be related to a different type in another module, or vice versa. Therefore, it makes sense to measure the *number of different variable names per type*, the *number of different types per variable name*, and the *number of unused variables*. Note that it is possible in ASF+SDF to define an unlimited number of variables using the Kleene star (`*`). For example the variable definition `"var"[0-9]*` means that `var` can be postfixed with any number of digits, thus enabling the creation of an unlimited number of `var` variables. We consider a variable unused in a module if it is never used in the module it is defined in.

A transformation function can have multiple signatures. A possible reason for this is that the transformation function is defined on a supertype and that each of the signatures deals with a subtype. Since all these signatures and accompanying equations have a similar purpose, it is likely that code clones are present. If the number of code clones (per code clone) exceeds a certain threshold, it may be advisable to create a function that covers the functionality of the code clones. Therefore, we propose to measure the *number of code clones*.

A start-symbol defines a starting point of a transformation. During testing and debugging it is likely that only parts of a transformation are used. To be able to use only a part, a start-symbol has to be defined. If there is more than one start-symbol present in a transformation, this could either mean that it is a leftover of the testing and debugging phase or that the transformation can be used in different ways. We propose to measure the *number of start-symbols*.

5 Relating Metrics to Quality Attributes

In this section we will discuss the relation between the metrics derived for ASF+SDF model transformation and quality attributes. Table 1 summarizes the discussion by indicating the relation between metrics and quality attributes.

Size (lines 1–6 in Table 1.) Size has a negative effect on the understandability and modifiability of a model transformation. The larger a model transformation is, the harder it is to understand or modify.

The size of the domain-specific part of a model transformation has a negative effect on reusability and reuse. This part of a model transformation is specific for

a transformation and it is therefore unlikely that it can be reused for transformations or that parts from other transformations can be reused. It would however be interesting to look for similarities among model transformations that have the same source or target model. In this way reusability, and also reuse, can be assessed more accurately.

The size of the domain-independent part has a positive effect on reuse. The domain-independent part of a model transformation is defined as the part that consists of library functions. Since these functions are in a library, they are already being reused. It can also be the case that during the development of a model transformation certain transformation functions are generic enough to put them in a library. In this case the size of the domain-independent part of a model transformation has a positive effect on reusability.

Function (lines 7–13 in Table 1.) The size of functions has a negative effect on understandability and modifiability of a transformation. Moreover, the number of signatures and equations per function has a negative effect on consistency. If more similar signatures or equations have to be written, it is more likely that a different style is used.

A high value for val-in or val-out generally means that a function is specific. This has a negative effect on reusability. The number of input parameters and return values also has a negative effect on understandability and modifiability.

A high fan-in value means that a function is often used by other functions. This can be an indication that the function is generic, which benefits reusability.

A high fan-out value means that a function uses a lot of other functions, among which may be library functions. Therefore fan-out benefits reuse.

Module (lines 14–21 in Table 1.) The number of modules is a metric for measuring the modularity of model transformations, though not a very good one. It needs to be combined with the metrics (un)balance and number of main functions per module to get an impression of how well the functionality of a model transformation is divided over modules.

Functions are put in a library to be reused. Therefore, the number of library functions has a positive effect on reuse.

Similar to their variants for functions, fan-in and fan-out for modules also have a positive effect on respectively reusability and reuse. The combination of fan-in and fan-out, i.e., information complexity, is a measure of complexity. The more complex a model transformation, the harder it is to understand. Therefore this metric has a negative effect on understandability.

The purpose of a module with multiple main functions is unclear. Therefore the number of main functions per module has a negative effect on understandability. Modularity is also negatively influenced, since the module can be split into modules with only one main function. Because the module is less understandable and could be more fine-grained, it is less reusable.

Parameterized functions are created to be used in multiple forms. Therefore the number of parameterized functions has a positive effect on reusability. It

also has a positive effect on reuse, since container types like list and table are parameterized library functions.

#	Metric	Quality Attributes							
		Understandability	Modifiability	Reusability	Reuse	Modularity	Completeness	Consistency	Conciseness
Size metrics									
1.	Lines of code	−	−						
2.	Number of functions	−	−						
3.	Number of signatures	−	−						
4.	Number of equations	−	−						
5.	Size of domain-specific part			−	−				
6.	Size of domain-independent part			+	+				
Function metrics									
7.	Number of signatures per function	−	−					−	
8.	Number of equations per function	−	−					−	
9.	Number of equations + conditions per function	−	−						
10.	Val-in	−		−					
11.	Val-out	−		−					
12.	Fan-in (function)			+					
13.	Fan-out (function)				+				
Module metrics									
14.	Number of modules					+			
15.	Number of library modules				+				
16.	Unbalance (module size − avg. module size)					−			
17.	Fan-in (module)			+					
18.	Fan-out (module)				+				
Consistency metrics									
19.	Module information flow complexity	−							
20.	Number of main functions per module	−				−	−		
21.	Number of parameterized modules			+	+				
22.	Number of signatures without equations			−			−	−	
23.	Number of variables per type	−						−	
24.	Number of types per variable	−						−	
25.	Number of unused variables						−	−	
26.	Number of code clones (per code clone)		−					−	−
27.	Number of start-symbols							−	

Table 1. Metrics related to quality attributes

Consistency (lines 22–27 in Table 1.) The inconsistency metrics obviously all have a negative effect on consistency. Signatures without equations indicate that parts of the transformation are not finished yet. Therefore this metric has a negative effect on both completeness and reusability.

Variables with the same name but different types and variables with different names but the same type are confusing. Therefore the metrics referring to these inconsistencies have a negative effect on understandability.

Unused variables should be removed. Therefore the number of unused variables has a negative effect on completeness.

Code clones may be replaced by a function, such that the code has to be written only once. This would make a model transformation more concise. Therefore the number of code clones has a negative effect on conciseness. Also, if a part of the model transformation containing code clones has to be modified this has to be done in multiple places. Therefore this metric also has a negative effect on modifiability and consistency.

6 Conclusions and Future Work

In this paper we presented the first results of our ongoing research on the quality of model transformations. The main contributions is a set of eight quality attributes that can be used to assess the quality of model transformations. To refine these quality attributes and make them tangible, we have presented a set of metrics that can be used to assess these quality attributes. Our initial results presented in this position paper are a basis for future work in the direction of quality of model transformations.

To assess the quality of model transformations, first a clear definition of quality is needed. In Section 3 we presented the eight quality attributes we identified thus far. We plan to extend this set of quality attributes and relate them in a quality model such as proposed in [2].

In this paper we focused on ASF+SDF model transformations. We expect that our techniques can be generalized and applied to other model transformation formalisms, such as ATL [14] as well. The intended quality model will be the same, but some metrics to assess the quality attributes need to be adapted to the specifics of the transformation formalism. We proposed the metric number of functions as a measure for the size of a transformation created in ASF+SDF. For model transformations created with ATL the number of transformation rules could be used to measure size. However, we expect that most metrics will be conceptually the same for different transformation formalisms.

We want to verify our approach by means of empirical case studies. It is infeasible and inaccurate to extract metrics from model transformations by hand. Therefore we have to implement a tool that can automatically extract the values of all of the metrics from a model transformation. Furthermore we would like to visualize the values of metrics in such a way that outliers and striking values can easily be observed. Something similar has been done for software designs [15].

Once we have identified quality problems in model transformations, we can propose a methodology for improving their quality. This methodology will probably consist of a set of guidelines which, if adhered to, lead to high-quality model transformations.

Acknowledgements This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

References

1. Schmidt, D.C.: Model-driven engineering. Computer **39**(2) (2006) 25–31
2. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merrit, M.J.: Characteristics of Software Quality. North-Holland (1978)
3. Rubey, R.J., Hartwick, R.D.: Quantitative measurement of program quality. In: Proc. of the 1968 23rd ACM national conference, ACM (1968) 671–677
4. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous & Practical Approach. 2nd edn. PWS Publishing Co. (1996)
5. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice Hall (1996)
6. Lange, C.F.J.: Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (2007)
7. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In Wilhelm, R., ed.: Proc. of the 10th International Conference on Compiler Construction, Springer (2001) 365–370
8. van Deursen, A.: An overview of ASF+SDF. In van Deursen, A., Heering, J., Klint, P., eds.: Language Prototyping: An Algebraic Specification Approach. Volume 5. World Scientific Publishing (1996) 1–29
9. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming process algebra models into UML state machines: Bridging a semantic gap? To appear in Proc. of the International Conference on Model Transformation (2008)
10. Alves, T., Visser, J.: SdfMetz: Extraction of metrics and graphs from syntax definitions. In Sloane, A., Johnstone, A., eds.: Proceedings of the 7th Workshop on Language Descriptions, Tools, and Applications. (2007) 97–104
11. Harrison, R.: Quantifying internal attributes of functional programs. Information and Software Technology **35**(10) (1993) 554–560
12. Jones, C.: Programmer Productivity. McGraw-Hill (1986)
13. Ince, D.C., Shepperd, M.J.: An empirical and theoretical analysis of an information flow-based system design metric. In Ghezzi, C., McDermid, J.A., eds.: Proc. of the 2nd European Software Engineering Conference, Springer (1989) 86–99
14. Jouault, F., Kurtev, I.: Transforming models with ATL. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference, Springer (2005) 128–138
15. Lange, C.F.J., Chaudron, M.R.V.: Supporting task-oriented modeling using interactive UML views. Journal of Visual Languages and Computing **18**(4) (2007)

A Basis for a Metric Suite for Software Components

Giovanni Falcone and Colin Atkinson

Chair of Softwareengineering, University of Mannheim
{falcone,atkinson}@informatik.uni-mannheim.de

Abstract. Although software components have been used in software engineering for quite some time, only a small number of metric suites have been designed to capture the idiosyncrasies of components and the systems developed from them. Moreover, these are rather limited because they treat components as if they were objects in an object-oriented programming language. In this paper we outline a more component-oriented metric suite which is based on four distinct views of components - external, shallow, deep and complete. By measuring the various properties of a component from these different viewpoints it is possible to create metrics which capture the relative distribution of "realization" between the component's own application logic, its nested components and its external used components. These, in turn, are likely to be indicators of quality characteristics such as reliability and maintainability.

1 Introduction

Software metrics have been used for over 40 years in industry and academic research for the measurement of non-functional properties of software artifacts. However, during this period, the languages and techniques used to build software artifacts have significantly changed. In the early days of software development, programming languages directly reflected the instructions supported by the underlying hardware platform (e.g. assembler languages). However, so called "high-level" programming languages were soon introduced to cope with the increasing complexity of software artifacts and allow them to be programmed using hardware-independent abstractions.

In the first generation of high-level programming languages, the principle of separating data and code - so called procedural software development - was dominant. Early software metrics were therefore defined to capture the basic properties of software artifacts written using this procedural paradigm. The most prominent metric suite for procedural software is the one developed by Halstead [6], even though Halstead defined his metric suite for text based documents.

In recent years the paradigm of object-oriented programming has become increasingly popular. A second generation of software metrics therefore evolved from the earlier software metric suites to reflect the distinct characteristics of the object oriented development paradigm. The most popular metric suites for object oriented programming are undoubtedly the *Object-Oriented metric suite*

by Chidamber and Kemerer [4] and the *Metrics for Object-Oriented Design* by Brito e Abreu [5].

More recently, component based software development (CBSD) has grown in popularity [14]. CBSD aims to allow developers to reuse already existing software components wherever possible instead of having to develop all functionality from scratch. Software components are nowadays easy to obtain through software component market places like ComponentSource.com and Flashline.com or software component search engines like Merobase.com. Regardless of how software components are obtained, however, fundamental differences exist between traditional OO artifacts- software objects/classes - and CBSD artifacts - software components. The most clearly visible difference between them lies in the scale and complexity of their functionality. Whereas objects typically provide limited functionality, components generally offer more advanced and more complex functionality. Another important difference is that components have explicitly defined "required interfaces" as well as "provided interfaces" whereas objects only explicitly expose their provided interfaces - their required interfaces are usually hidden inside their internal implementation.

Despite these differences, there are a few metric suites available to measure components and component-based architectures. Moreover, these either consider components from an OO like perspective [13] or are based on a graph oriented consideration of the black and white box perspective of components [10] or only concentrate on the relation of component metrics to quality characteristics [2], [8]. Component vendors, however, typically use the older metric suites mentioned above when attempting to measure and describe the properties of their components. Although these are able to capture some of their properties in a high-level way (e.g. number of offered operations) many of the subtle idiosyncrasies of components are lost. A metric suite developed specifically for components would avoid this problem, and if sufficiently general would provide a means for comparing components as the basis of a component marketplace.

After explaining our underlying component model in section 2, in section 3 we present the four fundamental views of components upon which our approach is based. The actual metrics themselves are presented in section 4. We do not aim to be complete or to be rigorous but to convey the core principles behind the approach. Section 5 then concludes with some closing remarks and a summary of ongoing work.

2 Software Component Model

One of the things that complicates the development of metric suites for software components is that there is no consensus on what a software component is. One of the most widely used definitions was formulated at the 1996 at the European Conference on Object Oriented Programming (ECOOP):

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software*

component can be deployed independently and is subject to composition by third parties. [14]

This is obviously a very general definition which can include any cohesive and compact unit of software functionality with well defined interfaces. This includes simple programming language classes/objects as well as more complex artefacts such as web services and Enterprise Java Beans.

One of the most important characteristics of a component model is whether it is flat or hierarchic. In a flat component model such as [11] components are regarded as indivisible (black box) entities which can only be composed at one level to create systems. In contrast, hierarchic component models such as Fractal [3], SCA [12], Darwin [7] and KobrA [1], allow components to be composed of other components in a recursive manner, yielding architectures with an arbitrary number of composition levels. Hierarchic component models, in turn, fall into two distinct groups - those like Fractal, SCA and Darwin, which assume that all application logic is implemented by the components at the leaves of the composition tree, so that higher level component have no application logic of their own, and those like KobrA which do allow higher level components to have their own application logic. Since the former is a special case of the latter, we assume a hierarchic component model of the kind supported by KobrA as the basis for our component metric suite.

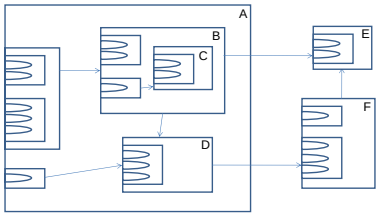

Fig. 1. Component Model

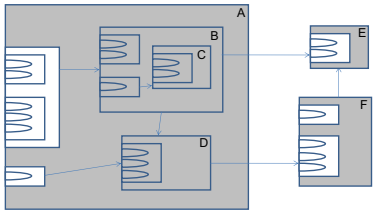

Fig. 2. External view of A

The basic elements of this component model are illustrated schematically in figure 1. In this figure, an operation is represented by a half-ellipse shape, and an interface, which is essentially an unstructured aggregation of operations and/or other interfaces, is shown as an encapsulating rectangle. A component is a rectangle with one or more interface rectangles attached to its periphery. It can also have one or more component rectangles contained within it to indicate that it is **composed** of those components. Finally, an arrow from one component to another indicates that the operation at the tail of the arrow **uses** the component at the head of the arrow, where by "uses" we mean that at least one of the operations of the using component calls an operation of the used component.

Figure 1 shows a system with 6 components, labelled A through F. A is the "largest" component which offers 6 operations arranged into two interfaces, one of which has two sub interfaces. A uses two **external components**, E and F,

and two **internal components**, B and D. B and D are referred to as **sub-components** of A, while E and F are referred to as **acquired-components** of A. B also has its own sub-component, C, and also uses component E. Thus, E is an acquired component of B as well as A, B's **super-component**. An important constraint of our component model is that all components acquired by a component must also be acquired by its super-component. In other words, a sub-component cannot secretly communicate with a component external to its super-component without that component knowing about it. Finally, although C is not a sub-component of A, it is **nested** within it and is thus referred to as a nested component of A.

3 Software Component Views

Most component models in the literature identify two fundamental views of components - a black-box view and a white-box view of components. The former describes the externally visible properties of the component (i.e. its interface in a general sense) and the latter extends this with knowledge about the internal realization of the component. The KobrA component model [1] captures this idea by distinguishing between the specification and realization of a software component. The specification is the black box view which describes all externally visible properties while the realization is the white box view which describes the internal realization of the component.

The problem with this simple black-box - white-box dichotomy of views on components is that it does not distinguish between "realization" that belongs to the component itself and "realization" that belongs to its nested and external components. Simple hierarchic component models such as Fractal, SCA and Darwin do not have this problem because all application logic is owned by the primitive leaf components, but in more general hierarchic models such as KobrA this question is not resolved. Also, the implication in all the above component models is that the realization of a component does not include the realization of its external components. However, this is rarely made explicit either.

To distinguish these different cases and provide a foundation for a comprehensive component metric suite we therefore define four different views of a component- **external view**, **shallow (internal) view**, **deep (internal) view**, and **complete view**. As their full names imply, the last three views are different kinds of realization views, while the first is the specification view of KobrA.

External View: As its name implies, the external view shows all the externally visible properties of a component including the interfaces of its acquired components. Figure 2 shows the information that would typically be included in the external view of component A from figure 1. The parts of the figure in white are included while those greyed out are not. This is intended to schematically identify the parts that are white in the white box sense and black (i.e. grey in this case) in the black box sense. The external view of A essentially includes all the information in the offered interfaces of A plus all

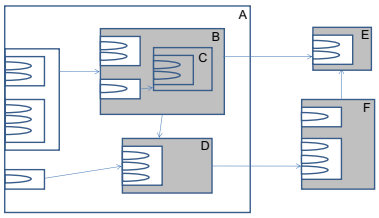

Fig. 3. Shallow View of A

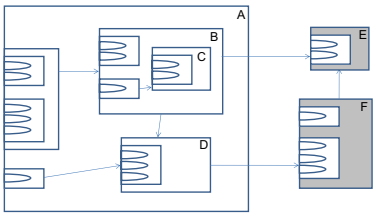

Fig. 4. Deep View of A

the information in the offered interfaces of the acquired components of A. Collectively the latter (i.e. the offered interfaces of the components acquired by A) are typically referred to as the required interface of A. For consistency, however, we refer to it as the **acquired interface** of A.

Shallow View: The shallow internal view of a component (or shallow view for short) includes the part of the overall realization of the component that is owned directly by the component itself. In the terminology of UML, this corresponds to the behaviour of the component that is realized by its own classifiers, and would be depicted by a behaviour port. As shown in figure 3, the shallow view of A includes everything in the external view of A plus the internal part of A that is not part of the realization of its subcomponents. The interfaces of the subcomponents of A are included however.

Deep View: The deep internal view of a component (or deep view for short) includes everything in the shallow view of the component as well as "realization" belonging to all nested components, recursively. It does not, however, include any realization of external components. Figure 4 shows schematically what is included in the deep internal view of the component A.

Complete View: As its name implies, the complete view of a component provides complete information about how the component is structured and realized. It includes everything in the deep view plus information about how the external components are realized. Figure 1, which was used to introduce our component model and terminology also provides a good schematic representation of what is included in the complete view of A. The difference between figure 1 and figure 4 is that the external components are now white as well.

This picture is a little misleading however. Although it is true from a theoretical perspective that full details of the internal structure and realization of a component's acquired components are included in its complete view, in practice we try to avoid this. We do this by defining metrics which are additive and have an extensive structure in the sense of [15]. This means that when one component contains another, the value of some metric on the containing component can be calculated by simply "adding in" the value of that metric for the contained component. Thus, instead of calculating complete metrics on a component by actually considering the full realization of its

acquired component, we can calculate the metric by "adding in" the values of that metric for the acquired components.

4 Proposed Core Component Metrics

In this section we explain how we propose to develop a more suitable set of metrics for components based on the previously defined views. When defining metrics on software artefacts it is common to distinguish between **direct measures** which represent some basic and directly "readable" properties of a software artefact, like the number of edges and nodes in the complexity measure developed by McCabe [9] and **indirect measures** that are a combination of the directly measurable properties of a software artefact. McCabe's cyclomatic complexity from this particular point of view needs to be regarded as an indirect measure since it is based on a combination of edges and nodes. The metrics we are considering in this section are mainly direct metrics.

Before we discuss true component metrics we first need to discuss two distinct kinds of metrics on component operations. Since, ultimately, all the functionality associated with a component is realized by its operations, and the operations that it uses, and so on recursively down to the primitive operations, component metrics that relate to component functionality must ultimately be based on operation metrics. Traditional software metric suites define many different ways of capturing the "complexity" or "volume" of functionality wrapped up in an operation, but they are all derived from the immediate shallow structure of the operation. For our component metrics, however, we need to support different forms of an operation metric which include the functionally of nested operations down to a certain depth. We therefore define three variants of a given metric:

Shallow: The shallow version of an operation metric is applied to the immediate realization of the metric and no more. Thus, any calls to other operations are treated as primitives with no inherent structure of their own (i.e. no inherent complexity, volume or whatever).

Deep: The deep version of an operation metric includes the immediate realization of the operation but also the realization of the operations it calls, recursively down to truly primitive operations. We are still working on the best semantics for doing this, but one simple approach would be to simply use macro expansion semantics. In other words, when calculating the deep version of a metric on an operation, any calls to other operations are replaced by the realizations of those operations, recursively. Some cut-off heuristics might be needed to avoid cycles, but the principle is straightforward (see also [15]).

Partial: A partial version of a metric is a version in which the inclusion of called operations in the metric for an operation is only performed to a certain depth, not down to all leaves of the call tree. This version therefore lies somewhere between the two extremes of shallow and deep versions of the metrics.

These different forms can be applied to most kinds of metrics on functions. Take McCabes cyclomatic complexity, for example, concretely the modified cylomatic complexity, due to its extensive structure and additiveness [15]. By applying these principles, it is possible to define a shallow cyclomatic complexity, a deep cyclomatic complexity, and several partial cyclomatic complexities (depending on the chosen recursion depth) of a given operation.

Having defined these different forms of operation metrics we are now in a position to discuss the component metrics from the different component views identified above. To organize things as systematically as possible, the metrics for each view are best into four categories: **Quantification Metrics**, **Individual Metrics**, **Average Metrics** and **Ratio Metrics**. We stress that these are not necessarily complete, and are not fully refined. The intent is to show the nature of the metrics how they can be organized. To illustrate how component functionality is handled we used the modified form of McCabes cyclomatic complexity.

4.1 External Metrics

External metrics are defined in terms of the information visible from the external view point.

Individual Metrics
Total Number of Parameters (TNP): The total number of parameters of an individual operation of the component.

Average Metrics
Average Number of Parameters (AvgNP): The average number of parameters for the operations of the component.
Average Number of Operations per Provided Interface (AvgNOPI): The average number of operations per provided interface of the component.
Average Number of Operations per Acquired Interface (AvgNOAI): The average number of operations per acquired interface of the component.

Quantification Metrics
Number of Provided Operations (NPO): The total number of operations provided by the component.
Number of Acquired Operations (NAO): The total number of operations provided by the acquired components.
Number of Provided Interface (NPI): The total number of interface provided by the component.
Number of Acquired Interface (NAI): The total number of interfaces acquired by the component.
Number of Acquired Components (NAC): The total number of acquired components.
Deepest Provided Interface Nesting (DPIN): The depth of nesting of the provided interface of the component with the deepest nesting.

Deepest Acquired Interface Nesting (DAIN): The depth of nesting of the acquired interface of the component with the deepest nesting.

Ratio Metrics

Interface Operation Ratio (IOR): $AvgNOPI/AvgNOAI$
Total Operation Ration (TOR): NOP/NAO

4.2 Shallow Metrics

The shallow metrics use the information that is visible from the shallow view of the component, but not the external view.

Individual Metrics

Shallow Cyclomatic Complexity (SCC): For an individual operation of the component, the partial complexity of the operation subsuming the realization owned by the component. Note that this is not necessarily the same as the conceptual shallow complexity defined in the previous section because an operation may use other operations that belong to the component. SCC is the partial cyclomatic in which the recursion (i.e. the macro expansion) stops if and when an operation of a subcomponent or external component is invoked.

Average Metrics

Average Number of Subcomponent Interfaces (AvgNSI): The average number of interfaces per subcomponent.
Average Number of Subcomponent Operations (AvgNSO): The average number of operations per subcomponent.
Average Shallow Cyclomatic Complexity Interface (AvgSCCI): The average SCC of the operations of the component.

Quantification Metrics

Number of Subcomponents (NSC): The total number of subcomponents.
Number of Provided Interface (NPI): The total number of provided interfaces of the subcomponents of the component.
Total Shallow Cyclomatic Complexity (TSCC): The sum of the SCC's for all the operations of the component.

4.3 Deep Metrics

The deep metrics use the information that is visible from the deep view of the component, but not in the shallow

Individual Metrics

Deep Cyclomatic Complexity (DCC): For an individual operation of the component, the partial cyclomatic complexity of the operation subsuming the realization internal to the component. Note that this is not necessarily the same as the conceptual deep complexity defined in the previous section because an operation may use operations of external components. SCC is the partial cyclomatic in which the recursion (i.e. the macro expansion) stops if and when an operation of an external component is invoked.

Average Metrics

Average Deep Cyclomatic Complexity (AvgDCC): The average DCC of the operations of the component.

Quantification Metrics

Number of Nested Components (NNC): The total number of nested components.
Deepest Nested Component (DNC): The nesting depth of the most deeply nested component.
Total Deep Cyclomatic Complexity (TDCC): The sum of the DCC's for all the operations of the component.

4.4 Complete Metrics

The complete metrics use the information that is visible from the complete view of the component, but not in the deep view.

Individual Metrics

Complete Cyclomatic Complexity (CCC): For an individual operation of the component, the complete cyclomatic complexity of the operation subsuming the realization owned by the component. This is the same as the conceptual deep cyclomatic complexity of the operation since the all used operation are included including those within.

Average Metrics

Average Complete Cyclomatic Complexity (ACCC): The average CCC of the operations of the component.

Quantification Metrics

Total Complete Cyclomatic Complexity (TCCC): The sum of the CCC's for all the operations of the component.

4.5 Relative Metrics

The different metrics from the different views of a component are interesting in of themselves, but the most interesting information comes from metrics that compare them - so called relative metrics.
Total External Cyclomatic Complexity (TECC): The difference between the TCCC and TDCC. This gives an indication of how much of the functionality of the component is covered by external components.
Total Delegated Cyclomatic Complexity (TDCC): The difference between the TCCC and TSCC. This gives an indication of how much of the functionality of the component is handled by the component itself.
Proportion External Cyclomatic Complexity (PECC): The ratio of the TECC to the TCCC. This gives an indication of what proportion of the functionality provided by the component is implemented by external components.
Proportion Delegated Cyclomatic Complexity (PDCC): The ratio of the TDCC to the TCCC. This gives an indication of what proportion of the functionality provided by the component is implemented by realization owned by the component (i.e. by its own application logic).

5 Conclusion

In this paper we presented the idea behind a novel a metric suite which is customized for software components. We have identified four fundamental views of components and used these to define a range of different metrics based on the core structural and architectural properties of components.

The two final relative metrics in the previous are two of the most interesting from our point of view since we believe they will be strong indicators of important external characteristics of a component. For example, we suspect that the Proportion External Cyclomatic Complexity (PECC) will be a good indicator of likely dependability since it is the use of external functionality in component and service based architecture which bears the highest risk. Similarly, we believe that the Proportion Delegated Cyclomatic Complexity (PDCC) will be a good indicator of the maintainability of a component because it indicates how much of functionality of the component needs to be maintained and how much is obtained from used components.

As mentioned above, although the deep and complete metrics theoretically include information about the nested and external components, we do not want to have to examine them in detail to calculate the deep and complete metrics. Rather the goal is to be able to calculate the deep and complete metrics for a component using the corresponding deep and complete metrics for its sub and external components. This should be possible because with one exception the metrics would appear to be additive. The one exception is when a component that is external to a subcomponent is not external to its super component, such as component D in the example. We are currently working on a strategy to address this issue. We also plan to focus investigate how these metrics relate to quality

characteristics like effort, reliability, maintainability, and so on. Additionally we plan to analyze how these metric relate to other metrics suites like the OO metrics.

References

1. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison Wesley, 2002.
2. Manuel F. Bertoa and Antonio Vallecillo. Usability metrics for software components. In *8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2004), Oslo, Tuesday June 15th 2004*, pages 17–25, October 2004.
3. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *CBSE*, pages 7–22, 2004.
4. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
5. Fernando Brito e Abreu. Toward the design quality evaluation of objectoriented software systems, 1995.
6. Maurice Halstead. *Elements of Software Science*. Elsevier, New York, 1977.
7. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.
8. Joaquina Martín-Albo, Manuel F. Bertoa, Coral Calero, Antonio Vallecillo, Alejandra Cechich, and Mario Piattini. Cqm: a software component metric classification model. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003) Darmstadt, Germany, July*, pages 54–60, October 2003.
9. Thomas J. Mccabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
10. V. Lakshmi Narasimhan and Bayu Hendradjaya. Theoretical considerations for software component metrics. In *Procedings of world academy of science, engineering and technology V10*, pages 165–169, December 2005.
11. OMG: Object Management Group. CORBA Component Model. http://www.omg.org/technology/documents/formal/components.htm, 2008.
12. OSOA: Open Service Oriented Architecture. SCA Assembly Model. http://www.osoa.org/display/Main/, 2007.
13. O. P. Rotaru and M. Dobre. Reusability metrics for software components. In *AICCSA '05: Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, pages 24–I, Washington, DC, USA, 2005. IEEE Computer Society.
14. Clemens Szyperski. *Component Software*. 2002.
15. Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co, 1997.