The Seventh International Workshop on Software Quality and Maintainability, March 5, 2013, Genova, Italy

P r e w o r k s h o p P r o c e e d i n g s

Proceedings of the Seventh International Workshop on Software Quality and Maintainability

Bridging the gap between end user expectations, vendors' business prospects, and software engineers' requirements on the ground.

Edited by Eric Bouwers and Yijun Yu
Sponsored by the Software Improvement Group

version 1.0

Foreword

Welcome to the 7th International Workshop on Software Quality and Maintainability. Continuing the debate of *Is software quality in the eye of the beholder?* started seven years ago, this workshop aims at feeding into it by establishing what the state of the practice is and, equally important, what the way forward is.

After a rigorous peer-reviewing process, 4 out of 9 submissions are accepted to be presented at the half-day workshop. This pre-workshop proceedings brings in original topics to the discussion, ranging from code analysis to design:

- **Does fixing a bug in one instance of code clones really fix the problem completely?** Martin Pölmann and Elmar Juergens examined six software systems from their version history to see what percent of bug fixes in clones leads to incomplete changes and potential bug candidates.

- **Can software quality concepts be applied to the design of a robot control framework, and can quality considerations of such a design guide the quality control of software in general?** Max Reichardt, Tobias Föhst and Karsten Berns used their own robot control framework to give a positive answer to both questions.

- **How much does a source code element such as classes or methods contribute to the maintainability index relatively?** Péter Hegedüs, Tibor Bakota, Gergely Ladányi, Csaba Faragó and Rudolf Ferenc measured the drilled downed metrics and compared the ratings to human experts.

- **How to make architecture styles conform to a quality model?** Andreas Goeb adapted the Quamoco quality modeling approach to enforce the conformance between a service-oriented architecture style with quality goals.

If these are not enough, the topics for discussions are enriched by our invited speaker Bram Adams, who will provide an overview of the studies into the impact of release engineering on software quality.

After the workshop, we encourage all the papers will be submitted to the special issue of Electronic Communications of the EASST by taking into account the feedback and comments discussed at the workshop.

To summarize, the complete workshop program is as follows:

14:00 - 14:10	Workshop welcome and introduction by the chairs
14:10 - 15:10	Invited talk by Bram Adams
15:10 - 15:30	Questions and discussion
15:30 - 16:00	Break
16:00 - 17:20	Presentation of 4 selected papers and Q/A
17:20 - 17:30	Closing of workshop

We hope all will enjoy this rather-packed half day event!

Eric Bouwers and Yijun Yu,
SQM 2013 Chairs, March 2013

Chairs

- Eric Bouwers, Software Improvement Group, The Netherlands
- Yijun Yu, The Open University, UK
- Miguel Alexandre Ferreira (publicity chair), Software Improvement Group, The Netherlands

Program Committee

- Arpad Beszedes, University of Szeged, Hungary
- Goetz Botterweck, University of Limerick, Ireland
- Magiel Bruntink, University of Amsterdam, The Netherlands
- Alexander Chatzigeorgiou, University of Macedonia, Greece
- Florian Deissenboeck, Technische Universitaet Muenchen, Germany
- Jürgen Ebert, University of Koblenz-Landau, Germany
- Neil Ernst, University of British Columbia, Canada
- Jesus M. Gonzalez-Barahona, Universidad Rey Juan Carlos, Spain
- Liguo Huang, UT Dalas, USA
- Siket Istvan, University of Szeged, Hungary
- Slinger Jansen, Utrecht University, Netherlands
- Robert Lagerström, the Royal Institute of Technology, Sweden
- Julio Cesar Sampio Do Prado Leite, Pontifca Universidade Catlica do Rio de Janeiro, Brazil
- Sotirios Liaskos, York University, Canada
- Lin Liu, Tsinghua University, China
- Christos Makris, University of Patras, Greece
- Nan Niu, Mississippi State University, USA
- Xin Peng, Fudan University, China
- Daniele Romano, Delft Technical University, The Netherlands
- Joost Visser, Software Improvement Group, The Netherlands
- Vadim Zaytsev, Centrum Wiskunde en Informatica, The Netherlands

Subreviewers

- Markus Buschle

Sponsors

Software Improvement Group, Amsterdam, The Netherlands

Contents

So, you are saying that our software quality was screwed up by ... the release engineer?!

Bram Adams

Abstract

Businesses spend a significant amount of their IT budget on software application maintenance. Each firm's portfolio of applications helps them run their daily operations, report their financials, and help them market and sell their products. Therefore, a firm's ability to improve the quality and maintainability of these applications will have a significant impact on their bottom line as well as establish credibility with their shareholders and customers. However, even though firms have spent significant time and money addressing this, they have achieved mixed results. Why?

Software release engineering is the discipline of integrating, building, testing, packaging and delivering qualitative software releases to the end user. Whereas software used to be released in shrink-wrapped form once per year, modern companies like Intuit, Google and Mozilla only need a couple of days or weeks in between releases, while lean start-ups like IMVU release up to 50 times per day! Shortening the release cycle of a software project requires considerable process and development changes in order to safeguard product quality, yet the scope and nature of such changes are unknown to most practitioners. This presentation will touch on the major sub-fields of release engineering (integration, build and delivery) and their interaction with software quality. We will explore state-of-the-art results in this domain, as well as open challenges for the SQM community. In the end, we hope to convey the message that seemingly innocent factors like shorter release cycles or version control branching structure have a major impact on software quality.

Revealing Missing Bug-Fixes in Code Clones in Large-Scale Code Bases

Martin Pöhlmann, Elmar Juergens
CQSE GmbH, Germany
{poehlmann,juergens}@cqse.eu

***Abstract*—If a bug gets fixed in duplicated code, often all duplicates (so called clones) need to be modified accordingly. In practice, however, fixes are often incomplete, causing the bug to remain in one or more of the clones. In this paper, we present an approach to detect such incomplete bug-fixes in cloned code. It analyzes a system's version history to reveal those commits that fix problems. It then performs incremental clone detection to reveal those clones that became inconsistent as a result of such a fix. We present results from a case study that analyzed incomplete bug-fixes in six industrial and open source systems to demonstrate the feasibility and defectiveness of the approach. We discovered likely incomplete bug-fixes in all analyzed systems.**

***Keywords*-code clones, clone detection, incomplete bug-fixes, evolution analysis, repository analysis**

I. Introduction

"I had previously fixed the identical bug [...], but didn't realize that the same error was repeated over here." [1] This excerpt of a commit message is a common verdict of developers who unveil inconsistencies in duplicated (cloned) code. As research in software maintenance has shown, most software systems contain a significant amount of code clones [2]. During maintenance, changes often affect, and thus need to be carried out, on all clones.

If developers are not aware of the duplicates of a piece of code when they make a change, the resulting inconsistencies often lead to bugs [3]. Awareness of clones in a system is especially important, if a developer fixes a bug that has been copied to different locations in the system. Those clones that are not fixed continue to contain the bug. Many studies have reported discovery of errors in clones in practice, often due to incomplete bug-fixes [3, 4, 5, 6, 7, 8, 9].

To avoid such incomplete bug-fixes, *clone management* [2] approaches have been proposed to alert developers of the existence of clones when they perform changes. However, while such approaches can possibly ease future maintenance, they are of no help with those incomplete bug-fixes that have happed in the past. How can we detect such inconsistent bug-fixes that are already part of the source code of a system?

One approach to detect incomplete bug-fixes in cloned code is to search for clones that differ from each other, e.g. in modified or missing statements. These differences could hint at incomplete bug-fixes. Several clone detection approaches exist that can detect clones with differences (so called type-3 clones) [10]. Unfortunately for the precision of this approach, however, not every difference between a pair of clones hints at a bug. In many cases, a developer copy & pastes a piece of code and modifies it intentionally, since the new copy is required to perform a slightly different function.

During the past five years, we have inspected clones in numerous industrial and open-source systems. Most of them contain substantial amounts of clones—including type-3 clones. Searching for incomplete bug-fixes by manually inspecting type-3 clones is inefficient, simply because many of the differences were introduced intentionally, often already during the creation of the clone. To reveal incomplete bug-fixes more efficiently, we ideally require an approach that can (at least to a certain degree) differentiate between intentional and unintentional differences between clones.

This paper proposes a novel approach to reveal inconsistent bug-fixes. It iterates though the revision history of a system and classifies changes as bug-fixes, if the commit message contains specific keywords like *bug* or *fix*. It then tracks the evolution of code clones between revisions to detect clones which become inconsistent *as consequence of a fix*. Our assumption is that such inconsistencies have a high likelihood of being unintentional. The case study that we have performed for this paper confirms this assumption.

Furthermore, in contrast to clones detected on a single system version, this approach provides information on which change, by which author and for which defect, caused the difference between the clones. From our experience, this information substantially supports developers in judging if and how to resolve differences between clones.

Problem: Bug-fixes in cloned code are often incomplete, causing the bugs to remain in the system. We lack approaches to efficiently reveal such incomplete bug-fixes.

Contribution: This paper presents a novel approach for detecting missing bug-fixes in code clones by combining clone evolution analysis with information gathered from the version control system.

We have implemented the approach based on the incremental clone detection functionality of the open-source program analysis toolkit ConQAT[1]. We have evaluated it in a case study on six industrial and open-source systems in Java and C#. The results of the case study show that the approach is feasible and does reveal missing bug-fixes.

[1] http://conqat.org

II. Related Work

This section gives an overview of approaches to reveal incomplete or missing bug-fixes. We distinguish between work concerning system evolution and clone detection.

A. Evolution-based

Kim et al. [11] proposed a tool called *BugMem*, which uses a database of bug and fix pairs for finding bugs and suggesting fixes. In particular, this system-specific database is built via an on-line learning process, since each revision of the version control system is scanned for a commit message hinting at a bug-fix. As suggested by Mockus and Votta [12] they use the terms "Bug" or "Fix" for identifying bug-fixes, as well as reference numbers to issue-tracking software like Bugzilla. For each fix-commit, the changeset data is extracted and separated into *code-with-bug* and *code-after-fix*. These code regions are normalized to generalize identifiers and stored in the database.

We use the same method for scanning the version history for interesting terms, but refrain from including references to bug tracking reports, since some systems track both bugs and feature requests with such tools. For including them, further work is required to distinguish bugs and requests, as well as making the data available offline.

Zimmermann et al. [13] took a similar approach by mining data from a version control system for a change recommendation system. Their goal is to suggest in an IDE changes and fixes [14] in the manner of shopping applications: "Programmers who changed these functions also changed...". The precision of meaningful suggestions is 26%. From a user-perspective the main difference to our approach is, that the recommendation tool tries to prevent bugs by suggesting changes upon modification of files in the IDE. Yet, the amount of wrong recommendations is rather high.

B. Clone-based

Juergens et al. [3] inspected a set of gapped clones for incomplete bug-fixes using their open source tool suite ConQAT. They proposed an approach for identifying gapped code clones using an algorithm based on suffix-trees and evaluated it on several large-scale systems. The results of this study show an average precision of 28% for detecting unintentional inconsistent clones. Nevertheless, the tool reported for all but one system over 150 inconsistent clones, which is a large amount for initial analyses.

The approach proposed in this paper also builds upon ConQAT, but uses another technique for detecting inconsistent code clones using an index-based algorithm in conjunction with evolution analysis. Hence, we compare our approach to that from Juergens et al. in terms of reported inconsistencies, precision and execution time.

C. Combined – Clone-evolution-based

APPROX of Bazrafshan et al. [15] is a tool for searching arbitrary code fragments in multiple versions and branches for similar fragments to find missing fixes. The search is based on code clone detection, but limited to search for code fragments similar to a search term. In contrast to our approach APPROX requires developers to know beforehand which code snippet contains a bug-fix and is of interest.

Duala-Ekoko and Robillard [16] created an extension for the Eclipse IDE, which reads a clone report from SimScan[2] and tracks the location of the clones automatically as code changes in the editor or between revisions. The approach focuses more on getting an overview about all related clones while editing a file, since they provide automated edit propagation to other clone siblings as well.

In contrast, Kim et al. [17] analyzed clone genealogies by combining CCFinder [18] with a clone tracking approach. In a case study they showed that only up to 40% of all clones are changed consistently during system evolution. Canfora et al. [1] used another clone detection tool as well as other study objects and reproduced the results from Kim et al. with about 43% of all clones being consistently modified. In detail, the inconsistencies sum up to 67% whereas 14% of these were propagated later to become consistent again.

Also Göde and Rausch [19] analyzed the evolution of three open source systems during a time frame of five years. For this, they used an iterative clone detection and tracking algorithm, and showed as well, that 43.1% of all changes to clones are inconsistent with 16.8% being unintentional inconsistent. Again, the total amount of reported inconsistencies is with 131 clones quite high and includes lots of false positives with respect to unintentional inconsistencies. As we go further and filter the revisions by commit message, we significantly reduce the amount of false positives.

Geiger et al. [20] presented an approach for identifying interesting correlations between code clones and change couplings mostly with respect to different subsystems. Change couplings are files that are committed at roughly the same time, from the same author and with the same commit message [21]. Nevertheless, they conclude that a correlation is too complex to be easily expressed and more information is needed to identify harmful clones. In our approach we will not correlate change coupling, but use a prediction of which commit is a fix based on its commit message.

III. Revealing missing Bug-Fixes

This section provides an in-depth explanation of how the analysis process of the proposed approach works. As depicted in Fig. 1 the process is an iteration over the available revisions of the version control system in order to simulate the source code evolution. For each iteration, several steps are performed to identify incomplete bug-fixes in code clones. At the end of each cycle, the iterator is queried for the next revision and a new detection starts. As soon as no newer revision is available, the bug detection results are reported and the analysis process terminates.

A. Get Next Revision

Upon the start of the analysis the version control system is queried for all or a subset of available revisions. During the

[2]http://blue-edge.bg/simscan

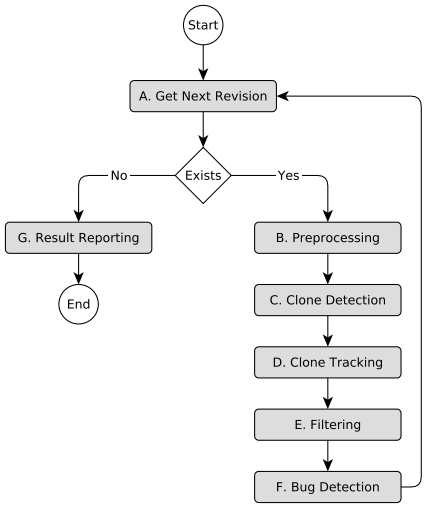

Fig. 1: Overview of the iterative bug detection process

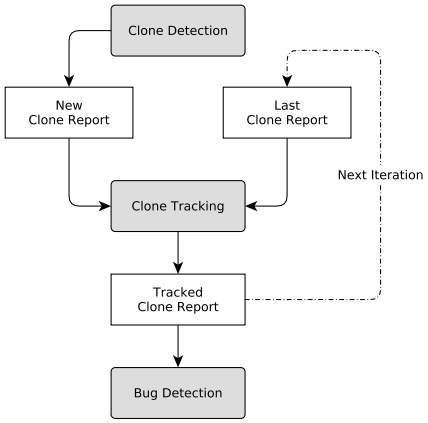

Fig. 2: Clone tracking dataflow process

iteration loop, the revisions are checked out in chronological order and metadata containing the commit message is handed over to the next steps.

B. Preprocessing

First, the source code is read from the disk into memory. Regular expressions are used as include and exclude filters for omitting generated and test code, since the former does not contain manual modifications and the latter is not interesting for production use. We further strip unnecessary statements, like package identifiers or include statements as these are unlikely to contain bugs. Finally, the source code is normalized into a generic representation which is insensitive to method names, variable names and literals.

C. Clone Detection

For detecting code clones, we use an algorithm using a hash index [22], which allows incremental updates of the clone data with high performance. For each revision, we gather the list of altered, added and deleted files and remove all data from the index, that belongs to these files. Afterwards, these files are added to the index again, but with updated content. Thus, we can keep most of the data and update just a small fraction depending on the changeset size.

Detection is configured to keep clones from crossing method boundaries. Hence, we can minimize the amount of semantically not meaningful code clones. Moreover, we do not take gapped clones (type-3) into account, which contain statement additions, removals and modifications after normalization. Including gapped clones in this step will not allow us to determine if the inconsistency in such a clone is related to a bug-fix.

D. Clone Tracking

This step performs the actual evolution analysis for code clones. For this, the reported clones from the clone detection step are mapped to those from the previous iteration cycle (*cf.* Fig. 2). Modifications that are performed inconsistently between clones, turns a group of ungapped cloned into a group of gapped clones. Moreover, these gapped clones are marked as *modified in this revision*.

The clone tracking is also performed by ConQAT and roughly follows the approach proposed by Göde and Koschke [23]. It first calculates the edit operations of a code clone between two consecutive revisions. Then it propagates these edit operations to the clones of the current revision, so the clone positions are updated accordingly. Afterwards, the updated clones are mapped to those from the the current detection step: First, those clones are matched, which positions do not differ. Second, a fuzzy coverage matching is performed on the remaining clones, that determines whether one clone covers an other and reports clones with modifications and gaps, that are of interest for the proposed bug detection approach.

E. Filtering

All clones without gaps are filtered, because they cannot contain incomplete bug-fixes. We also remove clones that differ too much with regards to their length. The threshold we choose is 50% around the average length of all clone instances of a group.

For example, a group of clones with two instances of length 23 and one of length 8 has an average clone length of 18. As the length of the shortest instance lies outside the 50% interval around this average length ($[9, 27]$), it is removed from the group. The other instances remain, because they lie within the interval. If all clones of a group lie outside this interval, the entire group is discarded.

F. Bug Detection

Clones are classified in those that contain an incomplete bug-fix and those that do not, as outlined in Fig. 3. To achieve this, the version control system is first queried for the commit message of the current revision. In the message we search for terms like *fix*, *defect* or *bug* that may indicate a bug-fix. If such a term is found, the whole commit and its modifications are seen as bug-fix commit. Mockus and Votta [12] proposes

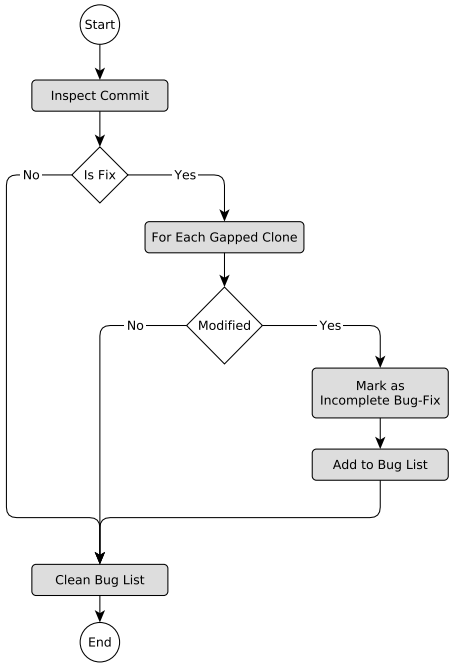

Fig. 3: Incomplete bug-fix detection flow

an even more extensive approach for finding fixes in commit messages. In our case, system specific terms were sufficient.

Afterwards, the list of code clones is searched for those that were marked as *modified in this revision* in the clone tracking step. For those, the approach suggests that the clone contains an incomplete fix, as the commit is a bug-fix and the clone was inconsistently modified in this revision. Consequently, it will be added to the global list of all incomplete bug-fixes. Finally, we also need to clean this list of incomplete bug-fixes as soon as a clone became consistent again or vanished. We continue with a new iteration loop, as long as newer revisions are available.

G. Result Reporting

After the revision iteration terminates, the result reporting is the last step. It writes all incomplete bug-fixes into a XML file for manual inspection with the ConQAT Clone Workbench. The report contains details about the location of clone instances in the source code, which includes file name, start and end line, as well as the position of gaps. Moreover, also information about the revision that caused the clone to become inconsistent are stored, namely the commit message and the revision identifier.

H. Performance Optimization: Revision Compression

After analyzing the version history of some software systems, we found that only a small percentage of the commits represent a bug-fix. In the studied systems roughly 25%. We can exploit this for a notable performance improvement, since we only need to inspect each commit that represents a bug-fix. All revisions between bug-fixes can be compressed into a single composite revision, as depicted in Fig. 4. These

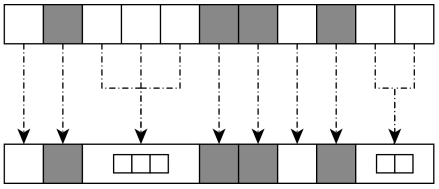

Fig. 4: Compressing non-bug-fix revisions (white) into single commits. Revisions including a bug-fix (gray) are not compressed.

compressed revisions are created by appending the commit messages and merging the changesets of altered files.

The general performance improvement can be described as follows: Let R be the total number of commits and F be the amount of fixes in a system (determined from the commit message) with of course $F \leq R$. Then we have to inspect R revisions without compression. With revision compression at most $2 * F + 1$ revisions have to be inspected, whereas R is still an upper limit that cannot be exceeded. For a system with a ratio of one fix per four commits, we can skip at least 50% of all revisions.

IV. Case Study

This section presents a case study which examines the amount of fixes in code clones of industrial and open source software systems. Moreover, it evaluates how well bugs can be detected with the proposed clone evolution approach and compares it to gapped clone detection on a single system revision.

A. Research Questions

We investigate the following five research questions:

RQ 1: Can fixes be determined from commit messages? The first question serves as fundamental analysis of how fixes can be detected solely from analyzing commit messages. It analyzes if a set of a few keywords can be chosen in a manner that they identify a commit as fix.

RQ 2: Which amount of code clones is affected by fixes? If inconsistent changes to clones do not occur in software systems, further analyses do not make sense.

RQ 3: How many inconsistently fixed clones qualify as bug candidates?
This question investigates whether the proposed approach is appropriate for detecting bugs and how many false positives are returned. For a toolkit in production use, a high precision in detecting incomplete bug-fixes is desireable.

RQ 4: What is the impact of the commit changeset size on the bug-finding precision?
Commits with a lot of modified files are likely to contain refactorings, feature additions or branch merges besides the actual fix. We suspect that more false positives are reported and try to give evidence by analyzing the precision with respect to limited changeset sizes.

TABLE I: List of the analyzed software systems

System	Organization	Language	History (years)	Size (kLOC)	Commits
A	Munich Re	C#	1.5	81.4	823
B	Munich Re	C#	1.5	370.6	638
C	Munich Re	C#	5	652.7	7483
D	Aol	Java	1.5	47.5	1449
Banshee	Novell	C#	7	165.6	8097
Spring	VMWare	Java	4	417.6	5034

RQ 5: How does evolution-based bug-detection compare to gapped clone detection on a single revision?
Finding incomplete bug-fixes can also be achieved by gapped clone detection. Hence, the question arises if the overhead of analyzing history information is justified compared to gapped clone detection in terms of precision and the amount of reported inconsistently fixed clones needed to be inspected by a developer or quality assurance engineer.

B. Study Objects

The case study was performed on six real-world software systems as listed in Table I. The reason for choosing these projects was on the one hand the requirement of having an evolved version history, on the other hand we need access to the version control system even for non-open-source projects. Thus, we relied on own contacts for industry code. In contrast, *Banshee* and *Spring* are available as open source systems and maintained by Novell and VMWare.

All systems are written by different teams, have individual functionality and evolved independently. They also differ in size and age. System A, B and C are owned by Munich Re, but are developed and maintained by different suppliers. They are written in C# and used for damage prediction and risk modeling. System D is an Android application developed by AOL. The two open source applications are the popular open source cross-platform audio player *Banshee*[3] written in C# by more than 300 contributors and the Java enterprise application framework *Spring*[4] developed by over 50 contributors. All systems are actively developed and in production use.

C. Determining Bug-Fixes — RQ 1

Design and Procedure: This question explores how well fixes can be determined from the commit messages of a version control system. To answer it, we manually inspected commit messages from all study objects and identified reoccurring terms which are used in bug-fix commits. These keywords were compiled to a regular expression so that it matches any of the terms.

Results: The manual inspection of all six systems yielded the following list of keywords suitable for identifying bug-fixes: *Fix, Bug, Defect, Correct*. As system A, B and C are developed by German engineers, some of the commit messages are written in German as well. This yields an extended set of keywords also containing the German words

[3]http://banshee.fm
[4]http://springsource.org/spring-framework

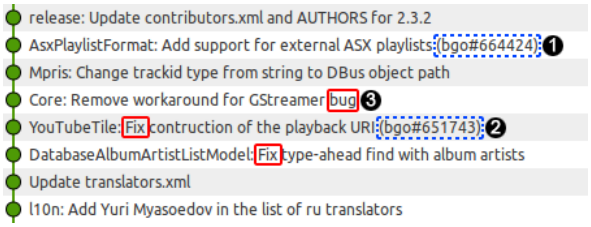

Fig. 5: Excerpt of commit logs from Banshee

TABLE II: Total amount and percentage of identified fixes

System	Commits	Fixes	Fixes (%)
A	823	194	23.6
B	638	203	31.8
C	7483	1754	23.4
D	1449	326	22.5
Banshee	8097	2016	24.9
Spring	5034	648	12.9

Fehler, *Defekt*, *behoben* and *korrigiert*. Additionally, the word *correct* seems to be often misspelled by developers as *corect*, so we also took this variant into account.

We decided against identifying commits as fixes solely from the presence of a reference to a bug-tracking software. As shown in Fig. 5, such references are mostly given by a number code representing the identifier of a bug or feature request in the issue-tracker. In the example, only the second commit with a bug-tracking identifier (❷) is a bug-fix, while the first one (❶) references a feature request. Thus, we suspect to threaten precision by generally taking these identifiers into account. All in all, we can compile the following regular expression to match a commit message against:

```
(?is).*(fix|bug|defe(c|k)t|fehler|behoben|
co(r|rr)ect|ko(r|rr)igier).*
```

Table II summarizes the amount of fixes detected by the above regular expression in each analyzed system. According to this result, almost every fourth commit is a fix.

Discussion: The results show that detecting bug-fixes from commit messages works reasonably well, at least for the systems we studied. Of course, the list of keywords used for detection varies depending on the software system and the language the developers speak. The proposed terms provide an initial starting point. But as soon as the bug detector is used on other systems, the terms need to be customized and tailored.

Still, it might happen that a commit is falsely identified as bug-fix, as depicted in Fig. 5 (❸). However, this seems to be a rare problem and in case of the above example the modification was at least related to a bug-fix.

D. Amount of Incomplete Fixes — RQ 2

Design and Procedure: The second research question investigates the amount of code clones that is affected by fixes and became inconsistent thereby. To answer it, we counted both, the total amount of incomplete fixes during project evolution, as well as those that were still present in the latest revision of the system history. Therefore, our bug detection

TABLE III: Evolution of incomplete bug-fixes

System	Total Incomplete Fixes	Still Present
A	48	28
B	60	50
C	108	61
D	26	15
Banshee	112	21
Spring	35	23

TABLE IV: Results of the bug detection for each system

System	Total Clones	Inconsistently Fixed Clones	Bug Candidates	Precision	Time (min)
A	200	28	11	0.39	2.7
B	765	50	9	0.18	23
C	778	61	23	0.38	116
D	170	15	6	0.40	5
Banshee	165	21	5	0.25	119
Spring	518	23	4	0.17	127

toolkit has been slightly modified to deliver these statistics. The configuration remained as described in Section III with a minimal clone length of 7 statements. Additionally, groups of clones with more than three instances were filtered, because for them the tracking approach is unreliable. Including these clones remains for future work.

Results: Table III shows for each system the amount of incomplete bug-fixes in code clones. For all systems less fixes are present in the last revision than occurred in total. This is due to corrected inconsistencies or completely removed clones. Moreover, a code clone can also be affected by more than one bug-fixed and appear multiple times in the above statistic.

Discussion: Depending on the system, we were able to reduce the total amount of code clones to a small fraction compared to all gapped clones, as shown by Table VI later in the case study.

Comparing the low amount of incomplete fixes for Banshee, which are still present in the last revision, to all inconsistencies found during the analysis shows that the detection algorithm is not stable with regards to big refactorings. The Banshee developers partially restructured the project with regards to sub components, which caused some detected bugs to be lost during tracking. Even gathering renamed files from the version control system will not completely eliminate this issue, since code may be exchanged between files as well. Also system C suffered from this in a minor extent.

Moreover, The Banshee Git repository contains lots of branching on the main development line, but the analysis loops through all commits in a sequential order provided by the *jGit* library. Thus it might switch between branches for consecutive runs and causing some bugs to disappear, due to tracking issues. An adapted evolution analysis is needed for this case, that traverses merged branches separately. This requires some major rework for the entire revision iteration and bug detection process, which is left for future work.

E. Detection Precision — RQ 3

Design and Procedure: This question investigates bug-detection precision. The inconsistently fixed clones, which were gathered with RQ 2, were manually inspected by the researcher and separated in false positives and bug candidates. For answering the question, we calculated the precision of the bug detector according to Equation 1.

$$precision = \frac{\#\ bug\ candidates}{\#\ inconsistently\ fixed\ clones} \quad (1)$$

The decision if an inconsistently fixed clone qualifies as bug candidate was made upon comparing the source code of clones using the ConQAT clone workbench for Eclipse and manual inspection of the commit message and modifications. This is shown in Fig. 6. We used a Laptop with a 2.2 GHz Quadcore CPU and 4 GB of RAM running a 32 Bit Ubuntu Linux with Oracle JDK 7 for the detection.

Our goal is to cost-effectively find missing bug-fixes. We are thus willing to sacrifice recall for higher precision and leave analysis of recall (if feasible at all) for future work.

Results: Table IV summarizes the total amount of code clones, which were detected during history evolution and present in the last revision, as well as the inconsistently fixed clones. Additionally, it lists the amount of bug candidates resulting from manual inspection, the precision calculated with Equation 1 and the time the detection took in minutes.

Besides Spring, the detection algorithm identifies roughly 10% of all clones as inconsistently fixed. The manual inspection of the researcher revealed that 17% to 40% of these reported clones are classified as bug candidates. The execution times varies with regard to the project size and the amount of revisions chosen for evolution analysis. Still, all analyses were performed in less than three hours.

Discussion: The lowest result with respect to precision shows system B and Spring. The latter has a very low rate of bug-fixes in general and just 23 bugs were reported out of over 5000 revisions with more than 400.000 lines of code per revision. According to the documentation it has very strict guidelines[5] for third party contributions with respect to coding style, unit-testing and patch submission and similar rules seem to apply for internal work as well.

For system B, lots of false positives were introduced by big commits that *"fixed"* coding style related issues. We did not count these as bugs. The same problem also decreases the results for system C. RQ 4 tries to alleviate this problem by limiting the amount of modified files per commit in order to be recognized as bug-fix.

F. Changeset Size Impact — RQ 4

Design and Procedure: This question analyzes how the size of the commit changeset influences the bug-finding precision. Analogous to RQ 3, we answered this question by inspecting the returned inconsistently fixed clones and determining the precision according to Equation 1. Therefore, the detection toolkit was executed with the same parameters as described for RQ 3. Additionally, a minimum and

[5]https://github.com/SpringSource/spring-framework/wiki/Contributor-guidelines

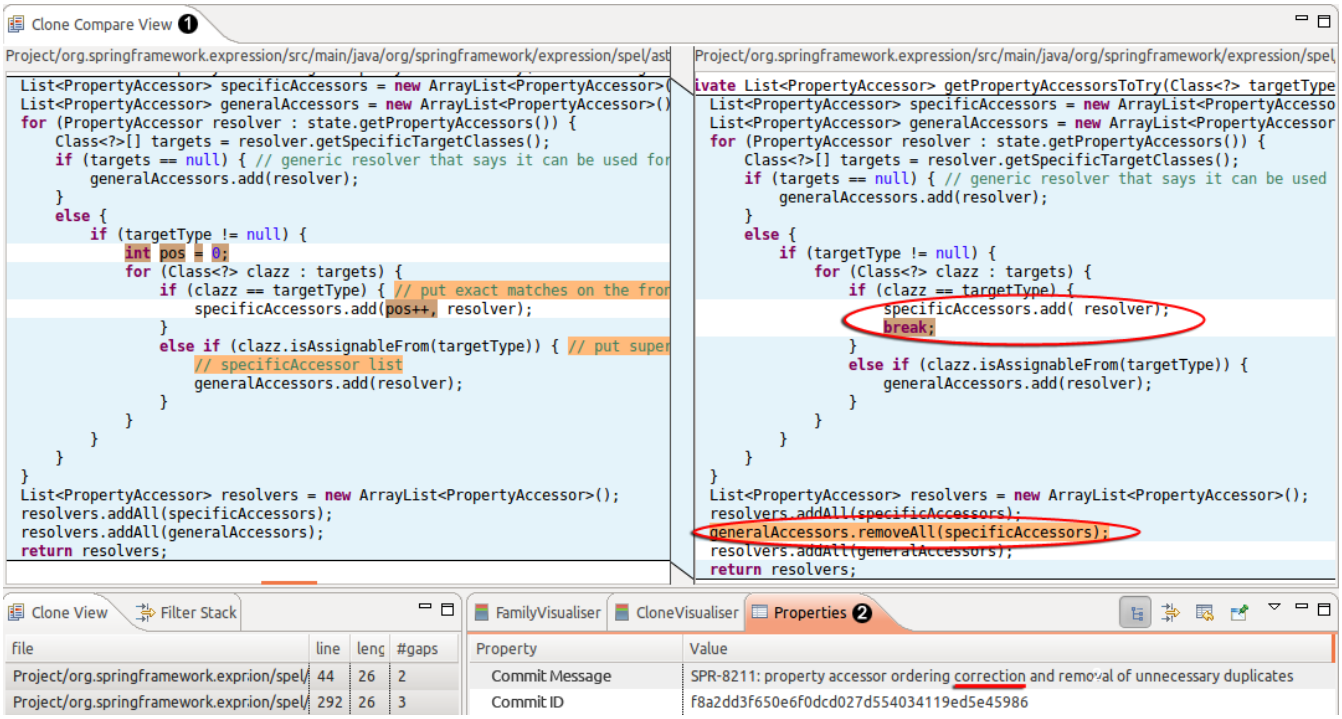

Fig. 6: Screenshot of the ConQAT clone workbench including a clone compare view (❶) and metadata about the commit this inconsistency was introduced by (❷)

maximum threshold to the changeset size of a commit is applied, which is evaluated at the time of revision compression. The changeset size is limited with window of size 5 sliding from 1 to 21. This results in the following intervals: $[1,5]$, $[6,10]$, $[11,15]$, $[16,20]$, $[21,\infty[$

Results: The results are summarized in Table V and grouped by the limit interval. One can clearly see that the precision in intervals $[1,5]$ and $[6,10]$ is almost two times higher than the average precision from Table IV and ranges from 30% to 60%. In contrast, the precision drops off significantly for commits with changeset sizes larger than 10. As discussed, the style related fixes in system B which lowered the results for RQ 3 fall into this category. Nevertheless, the systems performing worse before did not catch up to the other systems in terms of precision, but an increase is still noticeable.

It is worth a remark that the sum of the reported bugs or bug candidates of each system may not be equal to the results from Table IV, since a clone can be altered by fixes of different changeset sizes. Thus, some clones are listed multiple times.

Discussion: Applying a maximum limit to the changeset size of at most 10 altered files, the precision of the approach could almost be doubled and bugs are detected with an acceptable precision of 30% to 60%. We consider this sufficient for use in real-world assessments.

G. Gapped Clone Detection Comparison — RQ 5

Design and Procedure: This research question compares the evolution based bug-finding approach to the less time-consuming gapped clone detection. To answer it, we run a

TABLE V: Results with limits applied to the changeset size

Limit	System	Inconsistently Fixed Clones	Bug Candidates	Precision
$[1,5]$	**A**	11	8	0.73
	B	20	5	0.25
	C	35	13	0.37
	D	11	5	0.45
	Banshee	12	4	0.33
	Spring	17	5	0.29
$[6,10]$	**A**	4	2	0.50
	B	5	2	0.40
	C	11	9	0.82
	D	3	1	0.33
	Banshee	4	1	0.25
	Spring	3	1	0.33
$[11,15]$	**A**	0	—	—
	B	1	0	0.00
	C	5	1	0.20
	D	0	—	—
	Banshee	2	0	0.00
	Spring	0	—	—
$[15,20]$	**A**	3	1	0.33
	B	0	—	—
	C	0	—	—
	D	1	0	0.00
	Banshee	0	—	—
	Spring	0	—	—
$[21,\infty[$	**A**	7	0	0.00
	B	26	3	0.12
	C	5	2	0.40
	D	0	—	—
	Banshee	3	0	0.00
	Spring	0	—	—

TABLE VI: Results for finding bugs with gapped clone detection

System	Reported Clones	Inspected (%)	Bug Candidtates	Precision
A	42	42 (100%)	13	0.31
B	219	109 (50%)	26	0.23
C	192	96 (50%)	25	0.26
D	60	60 (100%)	15	0.25
Banshee	34	34 (100%)	8	0.24
Spring	166	83 (50%)	17	0.20

gapped clone detection with ConQAT on the study objects and filter the returned clones to contain at least one modification. The parameters from RQ 3 are used again with the following parameters being chosen especially for the gapped clone detection according to Juergens [24]: The gap ratio must be at most 20% and the edit distance has a maximum of 5 edits.

Finally, we determined the precision of finding bugs in this set of clones with Equation 1. The results are then compared to those of RQ 3 and RQ 4. Due to the large amount of reported inconsistently fixed clones for some systems, we just inspected a percentage of the reported clones for bug candidates, which are randomly chosen from the entire result set.

Results: Table VI presents the results for the gapped clone detection executed for each of the study objects. Compared to the results from the evolution based approach, one can see that the overall precision is more homogeneously ranging from 20% to 30%. Hence, there is no significant difference to the evolution analysis without taking changeset sizes into account. Nevertheless, the detection reported 2 to 6 times more clones we have to manually inspect. As positive side effect, also more bug candidates were detected. As for the enhanced approach with limits applied to the changeset size, the gapped detection performs clearly worse in terms of precision.

Discussion: The results of the evolution based approach are gained with a time consuming analysis that took over two hours for some systems. Hence it is valid to ask if a simple gapped clone detection, which only takes two minutes to execute, does the job of finding inconsistent clones with similar precision.

Just by taking the results from the basic evolution based approach one may concede this point. Nevertheless, the gapped detection was performed with additional parameters that already filtered lots of false positives. Not applying those filters increases the size of inconsistently changed clones for system A from 42 to 309. For the evolution-based approach, we did not apply these filters and may gain further precision by applying them. Furthermore, when it comes to comparison with limited changeset sizes, the precision is clearly higher than for gapped clone detection. Thus, we consider the long execution times as justified.

Besides precision, there are other valid arguments for favoring the history based approach: First, we can gain important information about the fix from the corresponding commit messages. Furthermore, in an continuous scenario, we just have to update the clone index for altered files and can thus return new results in almost real-time. Related to the future extensions, the revision information can also be used to get knowledge about the person who introduced the inconsistency.

It is noteworthy that the gapped detection unveiled some bugs, we did not encounter before, but vice-versa the evolution based approach reported some bug candidates not found by the gapped approach as well. Hence, also a combination of both methods could be beneficial.

H. Threats to Validity

This section gives an overview of internal and external threats to validity of the case study and how we tried to mitigate them.

Internal Validity: The main error source for the case study may be determining if an inconsistently fixed clone is a bug candidate, since the researcher has no deep knowledge of how the analyzed systems are built and components work together. We tried to mitigate the threat by inspecting the results twice and concluded with the same results. For future work we also want to verify the results by developers.

For answering the research questions, we did not take recall into account. Yet, this is no problem with regards to the aim of the proposed toolkit. The key requirement is to find bugs with high precision combined with context information. This set of tool-reported bugs should contain as less false positives as possible to minimize manual inspection efforts. As long as the time spent searching for bugs is justified by the bugs we find, we do not mind how many we miss: the time invested into finding bugs paid off.

Another group of threats concerns the program evolution. Depending on the version control system used, fixes that happen on feature branches are not visible on the main branch after being merged. All systems that were imported from Subversion and Microsoft Team Foundation server suffer from this problem, whereas the Git based systems Banshee and Spring do not. Similarly, we will not detect clones that were newly created and a fix was applied to the code before committing to the version control system again. These problems are more or less technical restrictions that cannot be prevented and thus the set of reported bugs may be smaller than the actual set of inconsistent fixes that were applied to code clones.

A further problem may be clone false positives, which are code regions that are syntactically similar to each other but contain no semantic similarity. Examples are *e.g.* lists of getters and setters with different identifiers. We included those false positives in the results of the case study and counted them as not representing a bug candidate. Doing so, we penalize the precision of the bug-finding tool.

Finally, the list of keywords for identifying bugs may not be exhaustive. Again, our aim is not to find all possible bugs, but a subset with a high precision. Moreover, adding new keywords for other systems is easy.

External Validity: The systems chosen for the case study as study objects may not represent an average software system. Yet, for closed-source systems, we are limited to existing industry contacts. However, all systems are developed by

different teams and for different purposes as described in Section IV-B. Moreover, RQ 1 and RQ 3 showed that they also have different characteristics in terms of bug evolution and amount of code clones. We are thus convinced that we have no strong bias in the results.

V. Conclusion and Future Work

This paper contributed to the analysis of the evolution history of code clones with the goal to find incomplete bug-fixes. A novel approach has been proposed that inspects commit messages for terms indicating a bug-fix in conjunction with unveiling gapped clones from evolution analysis.

We have performed a study on six real-world open source and industrial software systems for evaluating this approach. The results clearly show that inconsistent fixes—although varying in number—are a problem common to many software systems. The proposed toolkit helps revealing this missing bug-fixes in code clones with an acceptable precision of 30% to 60%. Compared to gapped clone detection, which has a precision of 20% to 30%, the evolution analysis produces more precise results. Moreover, bugs are not only reported, but with commit messages and inconsistent clone pairs valuable context information is provided.

The approach is suitable for both first-time analyses of a system which may even be performed by persons not familiar with the system as well as continuous analyses. The latter is supported by the index-based clone detection backend, which supports fast incremental updates.

For future work, the approach can be extended, to gain further precision or performance improvements. We plan to do a combined approach of gapped clone detection and evolution analysis with some kind of weighting of the found incomplete fixes. Also the content of altered source code can be taken into account. Added null-checks, catched exceptions or additional if-clauses are highly suspect to represent missing bug-fixes if applied inconsistently. However, this requires additional research and goes beyond the scope of this paper.

Further performance improvements can be gained by keeping the normalized source code in memory between consecutive iterations and just update modified files. An analogous method is already used for updating the clone index and needs to be applied here as well, since disk operations are one essential bottle neck for large-scale system analyses.

Acknowledgment

The authors would like to thank Munich RE Group and AOL for supporting this study. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “EvoCon, 01IS12034A”. The responsibility for this article lies with the authors.

References

[1] G. Canfora, L. Cerulo, and M. Di Penta, “Tracking your changes: a language-independent approach,” *Software, IEEE*, vol. 26, no. 1, pp. 50–57, 2009.

[2] R. Koschke, “Survey of research on software clones,” in *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 485–495.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: Finding copy-paste and related bugs in large-scale software code,” *IEEE Trans. on Softw. Eng.*, 2006.

[5] E. C. Lingxiao Jiang, Zhendong Su, “Context-based detection of clone-related bugs,” in *Proc. of ESEC/FSE ’07*, 2007.

[6] L. Aversano, L. Cerulo, and M. Di Penta, “How clones are maintained: An empirical study,” in *Proc. of CSMR ’07*, 2007.

[7] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones,” *Empirical Software Engineering*, 2009.

[8] T. Bakota, R. Ferenc, and T. Gyimothy, “Clone smells in software evolution,” in *Proc. of ICSM ’07*, 2007.

[9] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, “Predicting source code changes by mining change history,” *IEEE Trans. on Softw. Eng.*, 2004.

[10] C. Roy, J. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, 2009.

[11] S. Kim, K. Pan, and E. E. Whitehead Jr, “Memories of bug fixes,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 35–45.

[12] A. Mockus and L. G. Votta, “Identifying reasons for software changes using historic databases,” in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120–130.

[13] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004, pp. 563–572.

[14] T. Zimmermann, N. Nagappan, and A. Zeller, “Predicting bugs from history,” *T. Mens, S. Demeyer (Eds.), Software Evolution, Springer*, pp. 69–88, 2008.

[15] S. Bazrafshan, R. Koschke, and N. Gode, “Approximate Code Search in Program Histories,” in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 109–118.

[16] E. Duala-Ekoko and M. P. Robillard, “Tracking code clones in evolving software,” in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 158–167.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 187–196, 2005.

[18] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, no. 7, pp. 654–670, 2002.

[19] N. Göde and M. Rausch, "Clone Evolution Revisited," *Softwaretechnik-Trends*, vol. 30, no. 2, pp. 60–61, 2010.

[20] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of code clones and change couplings," *Fundamental Approaches to Software Engineering*, pp. 411–425, 2006.

[21] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings. International Conference on*, 1998, pp. 190–198.

[22] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1–9.

[23] N. Göde and R. Koschke, "Incremental clone detection," in *Workshop Software-Reengineering (WSR'09)*, 2009, pp. 219–228.

[24] E. Juergens, "Why and How to Control Cloning in Software Artifacts," 2011, Dissertation, Technische Universität München.

On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems

Max Reichardt, Tobias Föhst, Karsten Berns

Abstract— **Frameworks have fundamental impact on software quality of robot control systems. We propose systematic framework design aiming at high levels of support for all quality attributes that are relevant in the robotics domain. Design decisions are taken accordingly. We argue that certain areas of design are especially critical, as changing decisions there would likely require rewriting significant parts of the implementation. For these areas, quality-motivated solutions and benefits for actual applications are discussed. We illustrate and evaluate their implementations in our framework FINROC – after briefly introducing it. This includes a highly modular framework core and a well-performing, lock-free, zero-copying communication mechanism. FINROC is being used in complex and also in commercial robotic projects – which evinces that the approaches are suitable for real-world applications.**

I. MOTIVATION

Reaching high levels of software quality is in many ways a decisive factor for success when developing robot control systems – especially when systems grow beyond a certain size. With the intent to commercially develop and sell increasingly complex autonomous service robots on emerging (mass) markets, importance of this topic will rise even further. Certification for safety is a central requirement in this context.

Complex robot control software is typically implemented based on a robotic framework, toolkit or middleware. As these terms overlap, "framework" will be used in the remainder of this document. By defining a component model and dealing with many common problems, the selected framework has fundamental impact on the quality of robot control software: For example, if the framework is portable, efficient or scalable, robot control software will more likely be, too.

In fact, we observe the framework as *the* central adjustable factor determining software quality – with significant potential to introduce measures that actually guarantee or enforce certain quality requirements. From this point of view, frameworks are an important research topic for making progress in robotics software development. Due to its special nature, characteristics and constraints (e.g. autonomy in complex dynamic environments), it is worthwhile to investigate this topic specifically for the service robotics domain as possibilities and difficulties differ from other areas of (embedded) software development.

In order to implement quality measures and perform experiments, it is sometimes necessary to have full control over the framework core, architecture, tools or code repositories. Therefore, using a framework developed and maintained by a third party can be a limiting factor. Thus, we implemented FINROC[1] (see IV) considering factors altering software quality from the initial design phase in 2008. Nonetheless, as this is in fact a software quality attribute as well, we pay close attention that our efforts are interoperable with other projects in the open source robotics community. The fact that FINROC is being used in complex and also in commercial robotic projects (see VI) evinces that the presented approaches are actually suitable for real-world applications.

With respect to effort spent on software quality assurance, professional product development and most research groups are two different worlds. In academics, spending time on this task is typically not rewarded – as long as systems run sufficiently robust and efficient. In robotics, however, systems often have considerable complexity. Keeping them maintainable across multiple generations of PhD students is a major problem. In practice, many systems are abandoned when their developers leave. Due to time constraints in this context, measures to raise software quality are of particular interest if they cause only little extra effort in application development. Again, we see measures and policies implemented in the framework as a promising area to work on.

TABLE I: Relevant quality attributes in robotics software

Execution Qualities	Evolution Qualities
Performance efficiency	Maintainability
Responsiveness (latency)	Reusability
Safety and Reliability	Portability
Robustness and Adaptability	Flexibility
Recoverability	Extensibility
Scalability	Modularity
Usability and Predictability	Changeability
Functional Correctness	Integrability
Interoperability	Testability

II. SOFTWARE QUALITY IN ROBOTICS

Software quality is strongly related to non-functional requirements as these "characterize software quality and enable software reuse" [1]. "Although the term 'non-functional requirement' has been in use for more than 20 years" and there "is a unanimous consensus" that they are important, "there is

Max Reichardt, Tobias Föhst and Karsten Berns are with the Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Gottlieb-Daimler-Straße, 67663 Kaiserslautern, Germany {reichardt, foehst, berns}@cs.uni-kl.de

[1] http://www.finroc.org

still no consensus in the requirements engineering community what non-functional requirements are" [2]. International standards such as ISO/IEC 9126 and ISO/IEC 25010 structure non-functional requirements in quality characteristics and subcharacteristics. Mari *et al.* [3] distinguish between "execution qualities" and "evolution qualities". Following this, we believe the quality attributes collected in TABLE I are especially relevant across a wide range of control systems for service robots.

Various publications on robotics software deal with the necessity and difficulties reaching these quality attributes, such as [4], [1], [5] on software reuse, [6] on robustness and reliability, [7] on scalability or [8] on interoperability and integration.

Clearly, it is challenging for a software developer to consider all these attributes when designing a robotic application. Using a suitable framework can simplify this task significantly. We distinguish three levels of framework support:

- In the ideal case, quality attributes can be ensured "seamlessly". E.g. if the framework provides convenient facilities for efficient, scalable and robust inter-component data exchange – possibly providing real-time guarantees – the developer does not need to worry and the resulting application will not have deficiencies in this respect. Interoperability is another example.
- If bad code or bad component behavior can be detected automatically, requirements can be enforced by notifying the developer. This way, for instance, memory allocation or locking can be prevented in real-time code. Introducing a total ordering on locks – as shown in [9] – avoids dead-locks.
- There are many other measures to support certain software quality attributes that do not provide any guarantees though. Promoting good software development practices such as separating framework-independent from framework-dependent code will increase reusability and portability of software artifacts.

With FINROC as an object of study and validation, we investigate measures that can be implemented in a framework to support or even guarantee certain quality attributes.

III. CRITICAL AREAS OF DESIGN

Many important decisions must be taken when designing a framework – often involving trade-offs. Ideally, those decisions are well-founded. This requires a thorough understanding of the available options. Studying existing solutions, we identified alternatives, best practices and lessons learnt with respect to different areas of design. Their implications on a robot control's quality attributes were evaluated carefully.

Notably, design decisions greatly differ in criticality. In fact, some decisions can easily be changed later should other options seem superior. Others, however, can only be undone with immense development effort, which might not be realistic as this would often require rewriting major parts of a framework. Adding real-time support to a framework that was not designed with this in mind is an example. It is important to be aware of this fact.

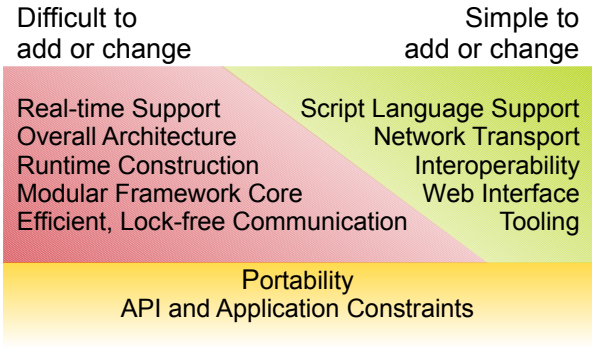

Fig. 1: Subjective criticality ratings for areas of design

Fig. 1 is an attempt to rate features and areas of design with respect to their criticality. Note that these ratings assume that a highly modular framework core is used (see III-B): e.g. a *network transport* mechanism is only easily exchangeable, if it is clearly separated from the rest of the framework. As long as the framework is implemented in a portable programming language and depends – if at all – on platform-independent libraries, *Portability* is not a major issue. Furthermore, any binary serialization mechanisms should take care of endianness. The *API* can always be extended with little effort. However, changing it can become laborious once a considerable amount of components and applications exist.

Having identified critical areas of design, we focused on getting those right in the initial FINROC release. As we believe that all the features on the left – real-time support, a highly modular framework core, runtime construction and efficient, lock-free communication – are beneficial to software quality, they are all supported. As it turned out, it is somewhat challenging to implement those features in combination. FINROC is actually the only framework we are aware of that supports them all. As this is not always obvious, their relevance for the service robotics domain is briefly discussed in the following.

A. Efficient, lock-free real-time implementation

Less computational overhead leads to lower latency and lower power consumption of a robot control. More tasks can be executed on a computing node – or smaller, cheaper nodes can be used. Nesnas [10] shares the view that communication overhead is critical: "An application framework must pay particular attention to avoiding unnecessary copying of data when exchanging information among modules".

Due to the modular application style, using a framework will always induce computational overhead compared to a perfectly engineered monolithic solution. However, frameworks such as Orocos [11] show that computational overhead can be low, despite a relatively loose coupling. In practice, as soon as it comes to buffer management or multithreading, we often observe that framework-based solutions actually outperform custom standalone code – sometimes drastically. This is due the fact that efficient, lock-free buffer management is complex to implement. As monolithic implementations are furthermore detrimental with respect to reusability,

maintainability and tool support, they are no alternative to using an efficient framework for realizing complex robot control software.

Efficiency also influences scalability – imposing limits on the maximum number of components that are feasible and therewith component granularity. In our research on large behavior-based networks, for instance, we develop systems that consist of far more than thousand components.

Notably, locking can be an even bigger issue with respect to latency and scalability – as we experienced exchanging data via blackboards in MCA2 [12]. Although based on an efficient shared memory implementation, locking them exclusively from different components quickly causes significant, varying delays. Such delays in the image processing components of our humanoid robot ROMAN, for instance, hindered natural interaction. Furthermore, overall cycle times are high. Being able to use shorter cycle times in FINROC immediately solved several problems we had with a lateral controller steering an agricultural vehicle. Lock-free implementations are certainly advantageous.

Finally, functionality with real-time requirements can only be realized without completely bypassing the framework (and taking care of multithreading and lock-free data sharing manually) if the framework provides real-time support. While real-time requirements can often be neglected in scientific experiments, this is a central topic for safety-critical parts of commercial robot control systems in order to guarantee that a robot will always react in time to certain events. For our bucket excavator THOR (see VI) this is certainly critical.

Notably, it is possible to separate the component communication mechanism ("transport") from the rest of the framework to make it exchangeable later – an approach taken in Orocos. This might seem to contradict the criticality rating in fig. 1. However, it does not as the framework must be deliberately designed for this. Apart from that, the common API for all transports can limit efficiency: e.g. if empty buffers are not obtained from the API, unnecessary copying cannot be avoided in multithreaded applications.

B. Modular framework core

Makarenko *et al.* [13] discuss the many benefits of frameworks having a slim and clearly structured code base – especially regarding development and maintainability of a framework itself. A modular framework core is furthermore beneficial regarding flexibility, extensibility and changeability of a framework and the resulting robot control systems. Furthermore, portability is increased, as many advanced features such as script language support or a web interface can be made optional. This allows creating slim – possibly single-threaded – configurations of a framework with minimal library dependencies that are suitable for small computing nodes. Generally, this configurability can allow to tailor a framework to applications, computing hardware and network topology. In the end, this makes the framework suitable for a broader range of systems.

The concept of "stacks" in ROS [14] goes into this direction. Some frameworks such as Player [15] feature an exchangeable network layer. In contrast, as MCA2 was tightly coupled to a custom TCP-based protocol, we encountered limitations with respect to robustness and efficiency that could not easily be resolved – especially in the context of tele-operation over weak wireless connections and the internet. With respect to ratings in fig. 1, a modular framework implementation with a clear separation of concerns is essential in order to classify design areas as simple to change.

C. Runtime Construction

The term "runtime construction" refers to the possibility to instantiate, connect and delete components at application runtime. Some frameworks support this on a process level: processes containing components can be started and terminated. If performance is not to be sacrificed, however, runtime construction is also relevant for components located inside the same process. Whether or not a framework should support especially the latter is a controversial topic – due to limited advantages and significantly complicating the framework implementation. We see some use cases for dynamic application structure:

- When operating in smart environments, robots typically need to condense the available sensor information in homogeneous views of the environment. Suitable components to perform such tasks can be created as sensors are encountered.
- Smart [6] proposes graceful degradation in order to increase robustness of systems. This includes dynamic rewiring of components in the case of sensor failure.
- If developers can modify an application while a robot is running – possibly using a graphical tool – effort for recompiling and restarting robot controls during experiments can be reduced significantly.
- It simplifies implementations – e.g. of network transport plugins with a dynamic set of I/Os. Furthermore, it provides the necessary facilities to handle application structure in e.g. XML files instead of source code – with the advantage that structure changes do not require recompiling and tools for round-trip engineering are simpler to realize.

Thus, runtime construction contributes to adaptability and flexibility of a system.

IV. FINROC IMPLEMENTATION AND EVALUATION

For the past decade, we have been using MCA2 [12] for developing robot controls, and learned to appreciate many of its qualities – such as real-time support, its scalability and its application style. By distinguishing between sensor and controller data, it is well-suited for application visualization. MCA2 was originally developed at the FZI[2] ("Forschungszentrum Informatik") in Karlsruhe. Over the years, we realized various modifications and enhancements in the MCA2-KL branch[3]. As its kernel is monolithic, difficulties improving several of the areas listed in fig. 1 were

[2] http://www.fzi.de/
[3] http://rrlib.cs.uni-kl.de/

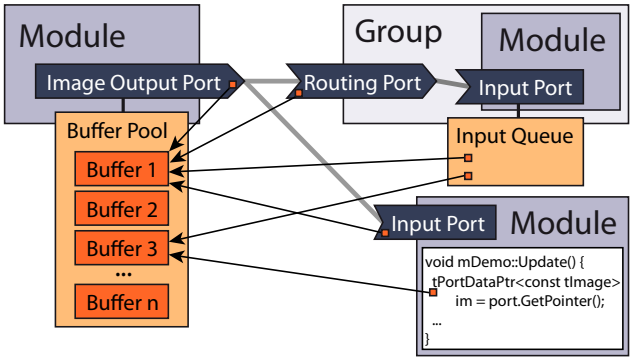

Fig. 2: FINROC's lock-free, zero-copying data exchange

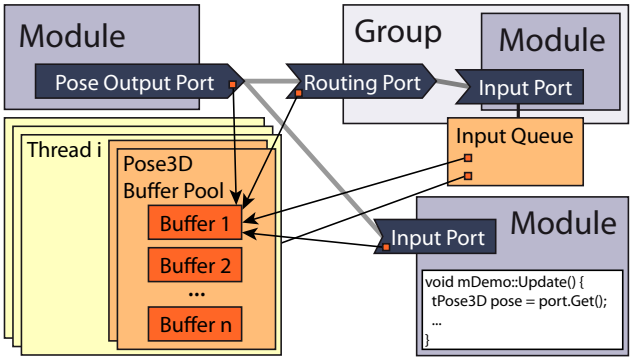

Fig. 3: Optimized implementation for cheaply copied types

encountered. Unable to find an open source framework with the properties we rate critical (see chapter III), we decided to implement FINROC. As Makarenko *et al.* [13] argue, developing and maintaining a framework can be feasible.

System decomposition is similar to MCA2: applications are structured in "modules" (components), "groups" and "parts" (OS processes). The interfaces of modules, groups and parts are a set of ports. These can be connected if their types are compatible. A FINROC application consists of a set of interconnected modules and can be visualized as a data flow graph. "Groups" encapsulate sets of modules (or other groups) that fulfil a common task. They structure an application. Modules and groups can be placed in "thread containers" to assign them to operating systems threads. Unlike in MCA2, it is also possible to trigger execution by asynchronous events – as Nesnas [10] recommends.

Furthermore, we separate framework-dependent from framework-independent code – as also [13], [16], [14] encourage. In our experience, this is the best way to make software artifacts portable and reusable across research institutions. In fact, most of our code base are actually framework-independent libraries (called RRLIBS). Over the last decade, we developed a considerable collection of drivers and libraries for robot controls available as RRLIBS. Many of them have already been integrated in FINROC.

There is a C++11 and also a native Java implementation of FINROC. The C++11 version currently depends on the platform-independent boost libraries[4]. We will remove this dependency, as soon as compilers on our systems have sufficient C++11 support. C++11 being the first C++ standard with a multithreading-aware memory model [9], it is preferable to plain C++ for safe, lock-free implementations. Notably, lock-free code that is safe on one CPU architecture might not be on another, due to different behavior with respect to memory ordering.

FINROC works well on ARM-based platforms such as Gumstix[5] or the PandaBoard[6]. Furthermore, FINROC's Java implementation compiles on Android, so apps can utilize it to communicate with robots.

[4] `http://www.boost.org/`
[5] `http://www.gumstix.com/`
[6] `http://pandaboard.org/`

A. *Lock-free, zero-copying intra-process communication*

Aiming to maximize (intra-process) communication efficiency, FINROC features a lock-free implementation that does not copy data. It is illustrated in fig. 2. The implementation supports input queues and allows switching between pushing and pulling data at application runtime. Notably, port connections can be changed without blocking threads that currently publish data via the same ports. Ports allow *n:m* connections – although connecting multiple output ports to one input port is typically discouraged.

The lock-free implementation is based on buffer pools, typically managed by output ports. Apart from the actual data, buffers contain storage for a timestamp and management data such as a reference counter. In order to publish data via an output port, an unused buffer is obtained from the port and filled with the data to be published. If all buffers are in use, another buffer is allocated and added to the pool. For real-time code, pools need to contain sufficient buffers so that this does not occur.

Ports contain atomic pointers, pointing to the ports' current values (symbolized by small orange squares in fig. 2). Publishing data replaces these pointers and updates reference counters. Obtaining the current buffer from a port is the tricky part: the pointer to the current buffer is read in one atomic operation and the reference counter is increased in another. In a naive implementation things can go wrong if the buffer is returned to the buffer pool (and possibly published again) in between these two operations. Therefore, we use tagged pointers and tagged reference counters. The reference counter may only be increased if it is not zero and if tags match. Otherwise, a new buffer has arrived and the procedure is repeated.

Once published, buffers are immutable (*get* operations on ports return `const` pointers). Almost any C++ type can be used in ports – including types without copy constructor and assignment operator, as well as `std::vectors` of them. Merely, stream operators for binary serialization need to be provided. If the type has no default constructor, a template needs to be specialized, so that buffers of this type can be instantiated.

Lock-free buffer pool management and atomics-based reference counting causes some computational overhead.

TABLE II: Results of image transport benchmark

Framework	Consumers	ØFPS	CPU	RAM
Orocos Locked	0	N/A	9 %	35 MiB
	1	50.00	40 %	52 MiB
	7	50.00	40 %	52 MiB
	15	50.00	41 %	52 MiB
Orocos Lock-free	0	N/A	2 %	29 MiB
	1	50.00	11 %	61 MiB
	7	50.00	49 %	202 MiB
	15	49.20	100 %	392 MiB
FINROC	0	N/A	6 %	32 MiB
	1	50.00	6 %	32 MiB
	7	50.00	6 %	33 MiB
	15	50.00	7 %	36 MiB
MCA2-KL	0	N/A	6 %	19 MiB
	1	50.00	6 %	19 MiB
	7	50.00	6 %	19 MiB
	15	50.00	7 %	19 MiB

Therefore, FINROC uses an optimized port implementation for small data types of constant size (see fig. 3). For such types, thread-local buffer pools are used and only the owner thread accesses the reference counter. Again, pointers are tagged in order to avoid the ABA problem.

In order to evaluate our implementation's impact on computational overhead, a benchmark with uncompressed, high-resolution camera images was set up – as they are quite costly to copy. References are MCA2-KL and Rock/Orocos with a state-of-the-art, intra-process communication model – either locked or lock-free with copying.

A producer-consumer scenario was set up in each of these frameworks[7]: One producer sends HD RGB24 images (1920×1080) at 50 fps to several consumer tasks – filling the image buffers via `memcpy`[8]. Consumers are port-driven and calculate the arriving frames per second. CPU load and memory consumption were determined via `htop`. Only thread-safe communication mechanisms were used.

The results are shown in TABLE II. Using lock-free communication in Orocos, CPU load and memory consumption grow drastically with an increasing number of consumers. In FINROC, on the other hand, adding consumers has minimal impact on CPU load or memory usage. Possibly, an Orocos transport plugin based on the mechanism presented here would be feasible. MCA2-KL uses only a single buffer and therefore has the lowest memory footprint. It has, however, issues with blocking (see III-A).

To measure the theoretical limits imposed by computational overhead from intra-process communication, five simple modules were connected to a control loop – each module reading and publishing a 4×4-matrix in every cycle. This control cycle can be executed with more than 1 MHz by a single thread that never pauses.

B. Highly modular framework core

Targeting a high level of modularity with a clear separation of concerns, we opted for a plugin architecture in

[7]All benchmarks were performed on an Intel Core i7 @2.67 GHz PC running Ubuntu 12.04, 32-bit

[8]Notably, this is not always necessary in FINROC. Consumers directly receive the buffers obtained from the v4l2 driver, for instance.

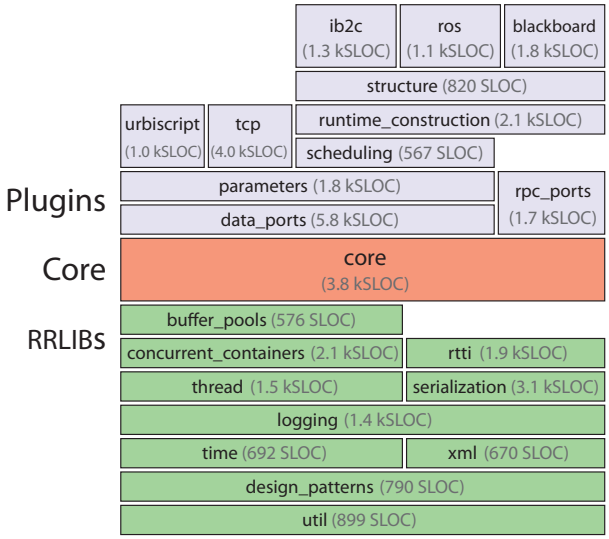

Fig. 4: FINROC's modular core with a selection of plugins

FINROC. Furthermore, framework-independent functionality was realized as independent RRLIBs. Fig. 4 illustrates how this currently looks like for a selection of plugins[9].

Functionality that is not needed in every application, is generally implemented in optional plugins. By combining plugins with the relevant functionality, FINROC can be tailored to the requirements of an application. Furthermore, functionality that supports developers can be added via plugins during the development process of a system. When software is to be deployed – possibly on small embedded nodes – this functionality can simply be removed.

As motivated by Makarenko *et al.* [13], a central target is keeping the code base of FINROC's core components slim as less code means lower maintenance effort and fewer errors. Quality assurance can be focused on important and relatively small core components. Research on better tools and experimental enhancements are ideally conducted in plugins with no impact on quality of the core components that are used in important projects.

As depicted in fig. 4, FINROC consists of many small independent software entities. These typically consist of only a few thousands lines of code. Hence, they can be reimplemented with reasonable effort. As long as interfaces stay the same, each of these entities can be replaced with alternative implementations – possibly optimized for certain hardware, application constraints or in some way certified. Single-threaded implementations, for instance, would be simpler and more efficient.

The plugins *data_ports*, *rpc_ports* and *blackboard* provide different mechanisms for component interaction. *data_ports* contains the implementation presented in section IV-A and is the primary mechanism for data exchange. The RRLIBs *concurrent_containers* and *buffer_pools* comprise lock-free utility classes that are central for its implementation. *structure* is the default FINROC API and provides the base classes

[9]Lines of code were counted using David A. Wheeler's 'SLOCCount'

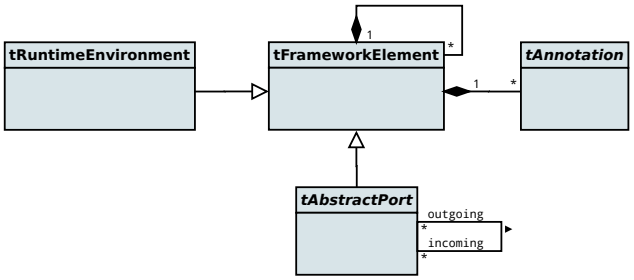

Fig. 5: Central classes in core

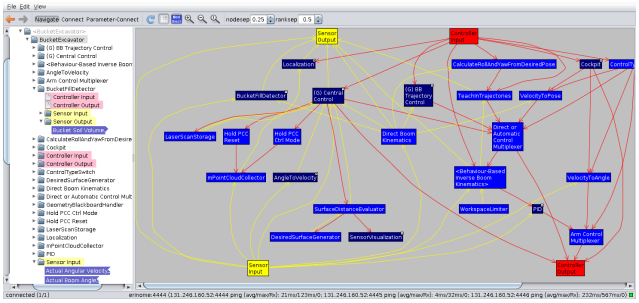

Fig. 6: FINSTRUCT: application visualization, inspection and construction

for modules, groups and parts as introduced in chapter IV, whereas *ib2c* contains the API and base classes for different kinds of behaviors in our iB2C architecture [17]. *urbiscript* adds experimental support for the scripting language from the URBI framework [18]. *ros* enables interoperability with ROS [14]. *tcp* provides a slim TCP-based peer-to-peer network transport supporting the publisher/subscriber pattern and featuring simple quality of service. It is currently the default in FINROC. *runtime_construction* contains functionality presented in IV-C.

There are four fundamental classes in the FINROC core (see fig. 5). We tried to come up with a simple structure to which the elements in MCA2, the FINROC API and possibly other frameworks can be mapped.

The central one is `tFrameworkElement`. It is the base class for all components, ports and structural entities. Framework elements are arranged in a hierarchy. In our tooling, this hierarchy is typically shown in a tree view on the left (see fig. 6). Then there is `tAbstractPort`. Ports of compatible data types can be connected – the core allows n:m. Such connections are network-transparent. The root framework element is the `tRuntimeEnvironment` singleton.

Several plugins need to attach information to framework elements – such as parameter links to config files, or tasks that need to be scheduled. Allowing the attachment of arbitrary annotations appears to be a fortunate design choice with respect to decoupling.

C. Runtime Construction

Decisions on support for concurrency significantly influence the effort required to implement runtime construction. In FINROC, two threads cannot make changes to the application structure (component hierarchy and port connections) at the same time. All operations required in control loops, however, execute concurrently, so that real-time threads in particular are not blocked performing their tasks. This requires using lists that allow iteration concurrently to modifications in several core classes. Furthermore, it is critical to ensure that no thread accesses a port or component when deleting it.

With these preparations in the core, the *runtime_construction* plugin has three central tasks. First of all, it manages a global list of instantiable component types. By default, FINROC component classes register on application startup providing class name and callback for construction. Applications do not necessarily need to be linked against the shared library files containing the components that are used. They can also be loaded dynamically at runtime using this plugin. Finally, it allows to store the current application structure in a simple XML format and restore it on the next run.

Based on these mechanisms, components can be instantiated, connected and removed graphically at application runtime using the FINSTRUCT tool (see fig. 6). Changes made to application structure can be saved to these XML files. In a way, FINSTRUCT is an optional round-trip engineering tool for application structure and is actually used in several projects in this way. Managing the structure information in separate files in a simple format is a reason for this working reasonably well. If the interfaces of components change, it might no longer be possible to instantiate all elements as stored. In this case, components or edges that have become invalid are discarded and possibly need to be recreated.

Elements are automatically arranged utilizing graphviz[10]. Apart from that, FINSTRUCT is the tool to visualize and inspect applications and can display and change the current values of all ports and parameters with supported data types. This includes unknown data types that can be serialized to strings or to XML. FINSTRUCT is the counterpart to MCA-Browser in MCA2.

V. POLICIES TO INCREASE MAINTAINABILITY

Frequent changes are a typical characteristic of software projects in the service robotics domain and it is important to keep the resulting effort for software maintenance low. With projects growing beyond a certain size, this is certainly (also) critical in small institutions such as university groups with the main developers always leaving after a few years. Framework design and policies have significant influence on this issue. In the following, we briefly discuss some of our experience and decisions we have taken in FINROC in this respect.

A. Application interface and constraints

An interesting question is whether imposing contraints on application structure is good practice – and also what they

[10] `http://www.graphviz.org/`

should look like. Frameworks such as ROS [14], Player [19] or Orca [20] strive to prescribe as little as possible or necessary, as unnecessary contraints can be a nuisance and lead to ugly workarounds. In MCA2, for instance, it is forbidden to connect modules in a way that lead to cycles in data dependencies. This led to inserting loop-back modules to realize such cycles in many applications.

On the other hand, strict guidelines help to avoid chaotic implementations. In our university group, many developers contribute to projects and libraries. Most existing code is from developers no longer working here. Experience shows that not all developers write clean code. Enforcing guidelines contributes significantly to keeping large software systems maintainable. Furthermore, controls of different robots have increased similarities, which facilitates reuse. Apart from that, guidelines, such as separating sensor and controller data in MCA2, allow visualizing applications in a clearer way – compared to using a typical layout algorithm on a "raw" data flow graph.

Since different kinds of APIs and programming styles are suitable for different levels of robot controls – e.g. CLARAty [10] explicitly separates a functional from a decisional layer – we decided to add APIs as plugins. This way, they are clearly separated from the framework core. Relatively strict constraints and guidelines are enforced in those APIs only, while the framework core prescribes as little as possible.

Ideally, the relevant API classes can all be mapped on the basic primitives the core provides. This allows using the same tools to interact with application parts based on different APIs – an advantage compared to using unrelated subframeworks.

B. Size of libraries and components

Having refactored a considerable amount of code from members that are no longer in our group, we experienced that – as a rule of thumb – reusable libraries with up to 5000 SLOC are typically comfortable to maintain and can be understood relatively quickly. So we try to keep all our libraries – including the framework core and its plugins – below this boundary (see IV-B). When a library becomes larger, it is checked whether there is a good way of splitting it up. Sometimes libraries contain relatively independent functionality. In other cases, core functionality and optional extensions can be identified. With this policy, we hope to support a clear separation of concerns and avoid heavy-weight software artifacts as well as feature bloat. This also increases suitability for embedded systems.

C. Separating Framework-independent Code

In our experience, reuse of software artifacts across research institutions works best with code that is framework-independent. The OpenCV library [21] is a good example. Developers of some frameworks explicitly encourage separating framework-independent code ([13], [16], [14]) – and we fully agree.

Fig. 7: The autonomous mobile bucket excavator THOR

Over the years, a considerable repository of reusable MCA2 libraries for all kinds of applications evolved. Before porting them to FINROC, we decided to separate the framework-independent code. As it turned out, most of the code actually is – leaving only thin modules that wrap this code for MCA2. Equivalent FINROC modules are even thinner. Notably, using it in an ADTF [22] component for an industrial partner was not a problem either. So this appears to be good practice for migration and for avoiding framework lock-in: If most of the code is independent, migrating existing projects to other frameworks becomes much less of an issue.

D. Coping with variability

A major challenge with respect to reusable software artifacts is handling variability across a broad range of projects (as discussed in [5]). We try to cope with this issue primarily using C++11 templates. In our view, this is a very powerful and appropriate mechanism allowing amazing designs – without any additional tooling – and providing strict type-safety along with high flexibility by decomposing type-behavior into policies [23]. On the downside, however, code can be hard to read for developers not familiar with these concepts.

E. Code Generation

Custom code generators are sometimes integrated into a framework's build toolchain with the intention to reduce development effort. In our experience, constraints, unforeseen side-effects and reduced transparency when tracking bugs can quickly outweigh any benefits – so adoption must be considered carefully. We deliberately minimized the amount of code generated by the framework to optional string constants for enums and port names. With respect to transparency, we want the complete system behavior to be evident from plain, versioned C++ code that an IDE such as Eclipse can index.

VI. CURRENT APPLICATIONS

The autonomous mobile bucket excavator THOR (see fig. 7 and 8) is the first major project we ported to FINROC. Apart from that, we are successfully using it in projects on agricultural machinery. Small forklift robots that we use for education including competitions are completely based

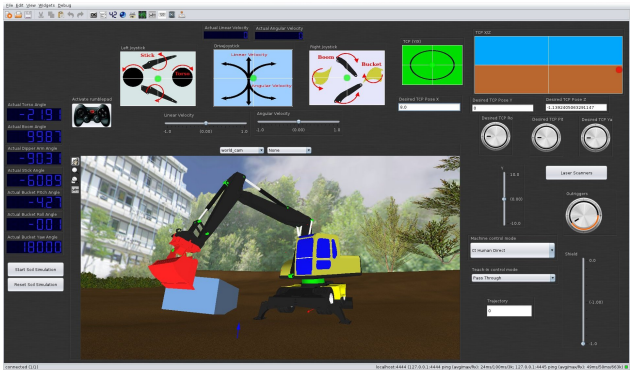

Fig. 8: User interface created in FINGUI tool connected to simulated robot

Fig. 9: Robot VIONA developed by Robot Makers GmbH

on FINROC, too. Several other projects are currently being migrated. As mentioned in chapter IV-B, there is a FINROC-based implementation of our behavior-based architecture iB2C. For the first time, the rules and guidelines worked out in [17] are properly and automatically checked and enforced.

Robot Makers GmbH[11], use FINROC in their product lineup. This includes the mobile offroad robot VIONA (Vehicle for Intelligent Outdoor NAvigation) – a commercially available robot platform with double-ackermann-kinematics (see fig. 9). Furthermore, they have developed FINROC support for the EtherCAT[12] real-time bus. This facilitates integrating standard components from automation industry.

VII. OTHER FRAMEWORKS

Numerous frameworks for robotics exists. However, we are not aware of any solution providing all the features we identified as critical in chapter III in combination.

With their support for efficient, lock-free communication, Orocos [11] and the derived Robot Construction Kit [16] probably come closest.

The "Robot Operating System" (ROS) [14] is currently the best-known and most wide-spread solution in academic institutions. Several frameworks are interopable with it. In this way, ROS has contributed to reusability and integrability of available robotics software. Several frameworks including ROS sacrifice performance for the benefits of an extremely loose coupling: components usually run in separate threads and exchange data exclusively via network sockets. Should the computational overhead be an issue in ROS, it is possible to use frameworks such as Orocos or FINROC inside a ROS node – making them somewhat complementary. Alternatively, ROS itself also has limited support for intra-process communication. This is, however, less sophisticated as the user needs to take care of buffer management manually.

Because of the limitations of MCA2, its original developer – the FZI in Karlsruhe – started working on MCA3 [24].

The many other robotic frameworks include Microsoft Robotics Developer Studio[13], URBI [18], CLARAty [10], OpenRTM [25], YARP [26] or cisst [27].

ADTF [22] is a solution with somewhat similar concepts used in the automotive industry. In this domain, AUTOSAR (AUTomotive Open System ARchitecture)[14] is an emerging architecture for the many control systems in vehicles sold on the mass market. It therefore needs to be slim and provide the high quality standards necessary for safety-critical applications.

VIII. CONCLUSION AND OUTLOOK

In this paper, we discuss the impact of a framework on software quality of robot control systems and propose systematic framework design aiming at high levels of support for all relevant quality attributes. In the scope of this document, we limit discussions to areas we identified as especially critical for initial design. Further areas are investigated in [28]. The sections on the approaches implemented in FINROC show how solutions for these areas can look like. Applications in research and industry indicate that the presented concepts work well in practice. However, it should be noted that these are not necessarily the best solutions.

In its current state, we believe that FINROC provides the necessary means to conveniently create efficient, complex robot control systems. Being interopable with ROS, our behavior-based architecture, for instance, may also be used inside ROS nodes.

Regarding measures to support software quality, the ones implemented are only the very beginning. There is certainly a lot more potential. Identifying, implementing and evaluating suitable such measures should be subject of future research.

There are several other directions of development, we currently pursue. In order to provide real-time guarantees across computing nodes, work on a plugin integrating the open source real-time bus PowerLink[15] is in progress. Apart from that, we intend to come closer to hardware – making FINROC's core components slimmer and more efficient in order to increase suitability for small embedded processors – possibly without an operating system. In fact, FINROC-lite was developed by Robot Makers as a temporary solution for a Nios 2 soft core.

[11] http://www.robotmakers.de
[12] http://www.ethercat.org/
[13] http://www.microsoft.com/robotics/
[14] http://www.autosar.org/
[15] http://www.ethernet-powerlink.org/

REFERENCES

[1] D. Brugali and P. Scandurra, "Component-based robotic engineering part i: Reusable building blocks," *Robotics Automation Magazine, IEEE*, vol. 16, no. 4, pp. 84–96, December 2009.

[2] M. Glinz, "On non-functional requirements," in *15th IEEE International Requirements Engineering Conference. RE '07*, New Delhi, India, October 15-19 2007, pp. 21–26.

[3] M. Mari and N. Eila, "The impact of maintainability on component-based software systems," in *Proceedings of the 29th Conference on EUROMICRO*, ser. EUROMICRO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 25–32.

[4] R. Vaughan and B. Gerkey, "Reusable robot software and the player/stage project," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Springer - Verlag, April 2007, vol. 30.

[5] C. R. Baker, J. M. Dolan, S. Wang, and B. B. Litkouhi, "Toward adaptation and reuse of advanced robotic software," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011, pp. 6071–6077.

[6] W. D. Smart, "Writing code in the field: Implications for robot software development," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30.

[7] A. Shakhimardanov, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschmar, "Analysis of software connectors in robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA, USA, September 25-30 2011, pp. 1030–1035.

[8] C. Cote, D. Letourneau, and C. Ra, "Using marie for mobile robot component development and integration," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30.

[9] A. Williams, *C++ Concurrency in Action: Practical Multithreading*, ser. Manning Pubs Co Series. Shelter Island, NY, USA: Manning Publications, February 2012.

[10] I. A. Nesnas, "The claraty project: Coping with hardware and software heterogeneity," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30.

[11] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006.

[12] K. U. Scholl, J. Albiez, and G. Gassmann, "Mca- an expandable modular controller architecture," in *3rd Real-Time Linux Workshop*, Milano, Italy, 2001.

[13] A. Makarenko, A. Brooks, and T. Kaupp, "On the benefits of making robotic software frameworks thin," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007)*, San Diego, California, USA, October 29-November 2 2007.

[14] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009.

[15] B. Gerkey, R. Vaughan, K. Sty, A. Howard, G. Sukhatme, and M. Matarić, "Most valuable player: A robot device server for distributed control," in *Proc. of the IEEE/RSJ Internatinal Conference on Intelligent Robots and Systems (IROS)*, Wailea, Hawaii, October 2001, pp. 1226–1231.

[16] "The robot construction kit," http://rock-robotics.org/.

[17] M. Proetzsch, T. Luksch, and K. Berns, "Development of complex robotic systems using the behavior-based control architecture iB2C," *Robotics and Autonomous Systems*, vol. 58, no. 1, pp. 46–67, January 2010, doi:10.1016/j.robot.2009.07.027.

[18] J.-C. Baillie, "Design principles for a universal robotic software platform and application to urbi," in *2nd National Workshop on Control Architectures of Robots (CAR'07)*, Paris, France, May 31-June 1 2007, pp. 150–155.

[19] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *11th International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, June 30 - July 3 2003, pp. 317–323.

[20] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Orca: A component model and repository," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Berlin / Heidelberg: Springer - Verlag, April 2007, vol. 30.

[21] G. Bradski, "The OpenCV library," *Dr. Dobbs Journal of Software Tools*, vol. 25, no. 11, pp. 120, 122–125, nov 2000.

[22] R. Schabenberger, "Adtf: Framework for driver assistance and safety systems," in *International Congress of Electronics in Motor Vehicles*, Baden-Baden, Germany, 2007.

[23] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[24] K. Uhl, M. Göller, J. Oberländer, L. Pfotzer, A. Rönnau, and R. Dillmann, "Ein software-framework für modulare, rekonfigurierbare satelliten," in *60. Deutscher Luft- und Raumfahrtkongress 2011*, Bremen, Germany, September 27-29 2011.

[25] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, Eds. Springer Berlin / Heidelberg, 2008, vol. 5325, pp. 87–98.

[26] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29–45, January 2008.

[27] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, October 18-22 2010, pp. 6107–6112.

[28] M. Reichardt, T. Föhst, and K. Berns, "Introducing finroc: A convenient real-time framework for robotics based on a systematic design approach," Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Technical Report, July 2012.

A Drill-Down Approach for Measuring Maintainability at Source Code Element Level

Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc
University of Szeged
Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary
{hpeter,bakotat,lgergely,farago,ferenc}@inf.u-szeged.hu

***Abstract*—Measuring source code maintainability has always been a challenge for software engineers. To address this problem, a number of metrics-based quality models have been proposed by researchers. Besides expressing source code maintainability in terms of numerical values, these models are also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved in order to reach higher overall quality.**

In this paper, we propose a general method for drilling down to the root causes of a quality rating. According to our approach, a *relative maintainability index* can be calculated for each source code element for which metrics are calculated (e.g. methods, classes). The index value expresses the source code element's contribution to the overall quality rating.

We empirically validated the method on the jEdit open source tool, by comparing the results with the opinions of software engineering students. The case study shows that there is a high, 0.68 Spearman's correlation, which suggests that relative maintainability indices assigned by our method express the subjective feelings of humans fairly well.

***Keywords*-Relative maintainability index, Method level maintainability, Metrics-based quality model, ISO/IEC 9126**

I. Introduction

Aggregating a measure for maintainability has always been a challenge in software engineering. The ISO/IEC 9126 standard [1] defines six high-level product quality characteristics that are widely accepted both by industrial experts and academic researchers. These characteristics are: functionality, reliability, usability, efficiency, maintainability and portability. Maintainability is probably the most attractive, noticeable and evaluated quality characteristic of all. The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software package. Although the standard provides a definition for maintainability, it does not provide a straightforward way of quantifying it. Many researchers exploited this vague definition and it has led to a number of practical quality models [2]–[5]. The non-existence of formal definitions and the subjectiveness of the notion are the major reasons for it being difficult to express maintainability in numerical terms.

Besides expressing source code maintainability in terms of numerical values, these models are also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved by the programmers in order to reach higher overall quality. Current approaches usually just enumerate the most complex methods, most coupled classes or other source code elements that bear with external values for some source code metric. Unfortunately, this is not enough; constellations of the metrics should also be taken into consideration. For example, a source code method with a moderate McCabe's complexity [6] value might be more important from a maintenance point of view than another method with a higher complexity, provided that the first one has several copies and contains coding problems as well. It follows that a more sophisticated approach is required for measuring the influence of individual source code elements on the overall maintainability of a system.

In this paper, we propose a general method for drilling down to the root causes of problems with the maintainability of a software system. According to our approach, a *relative maintainability index* is calculated for each source code element, which measures the extent to which the overall maintainability of the system is being influenced by it. For measuring the maintainability of a software system, we employed our probabilistic source code maintainability model [5].

We empirically validated the approach on the jEdit open source tool, by comparing the results with the opinions of software engineering students. The case study shows that there is a high, 0.68 Spearman's correlation at $p < 0.001$ significance level, which suggests that relative maintainability indices assigned by our method express the subjective feelings of humans fairly well.

In the next section, we summarize some of the studies related to ours. Then, in Section III we present a detailed description of our approach. Next, in Section IV we present a case study, which evaluates the usefulness of the approach. In Section V we collect some threats to the validity of our approach. Finally, in Section VI we round off with the conclusions and the lessons learned.

II. Related Work

The release of the quality standards like ISO/IEC 9126 [1], ISO/IEC 25000 [7] has created a new and very important direction of software quality measurement by defining the properties of software quality. The majority of researches deal with determining these quality properties for the system as a whole. However, only a few papers

study the quality on finer levels (e.g. methods or classes). The aim of the current work is to present a novel approach and algorithm for calculating quality attributes on this fine level. But first, we introduce the related results and techniques for determining the ISO/IEC 9126 quality characteristics on the level of source code elements.

Machine learning is a widely used technique to approximate subjective human opinions based on different predictors. It is a very powerful tool; usually the built-up models are close to the optimal solution. In this case the test and the learning instances usually come from the same dataset with the same distribution.

Bagheri and Gasevic [8] used this technique to approximate the maintainability property of software product line feature models. They studied the correlation between the low-level code metrics and the high-level maintainability subcharacteristics evaluated by graduate students. They also applied different machine learning models to predict the subjective opinions.

In our previous work [9] we used machine learning algorithms to predict the maintainability subcharacteristics of Java methods. The subjective opinions were collected from 36 experts and the effectiveness of the models was measured by cross-validation. Based on different source code metrics the J48 decision tree algorithm was able to classify the changeability characteristic of the methods into 3 classes (bad, average, good) with 0.76 precision.

The bottom-up methods on the other hand do not use subjective opinions about the characteristics, the validation dataset is independent from the model.

Heitlager et al. [10] described a bottom-up approach developed by the Software Improvement Group (SIG) [11] for code analysis focused on software maintainability. They use threshold values to split the basic metric values into five categories from poor to excellent. The evaluation in their approach means summing the values for each attribute (having the values between -2 and +2) and then aggregating the values for higher properties.

Alves et al. [12] presented a method to determine the threshold values more precisely based on a benchmark repository [13] holding the analysis results of other systems. The model has a calibration phase [14] for tuning the threshold values of the quality model in such a way that for each lowest level quality attribute they get a desired symmetrical distribution. They used the $\langle 5, 30, 30, 30, 5\rangle$ percentage-wise distributions over 5 levels of quality.

The SIG model originally used binary relations between system properties and characteristics, but Correia et al. [15] prepared a survey to elicit weights to the model. They concluded that using weights does not improve the model because of the variance in developers' opinion. Furthermore, as an interpretation of the ranking, an evaluator can point out the really weak points of the examined system regarding the different quality attributes.

Our approach has the same phases as the SIG model, but the phases themselves are very different. The most significant difference is that our approach presented in this paper is created for measuring the maintainability of source code elements (e.g. classes or methods) and not the system as a whole. Our purpose was to determine the critical elements of a system which cause the largest decrease in overall maintainability, providing technical guidelines for the developers.

To convert the low-level source code metrics into quality indices we also used a benchmark with a large amount of system evaluations, but we applied it in a different way. During the calibration, instead of calculating threshold values we approximate a normal distribution function called benchmark characteristic (see Section III), which is used to determine the goodness of the system with respect to a certain metric. The distribution used in the SIG model is also very close to a normal distribution but we had to use a continuous scale since the impact of a source code element on the overall quality of the system could be very small.

The SQUALE model presented by Mordal-Manet et al. [16] introduces the so called practices to connect the ISO/IEC 9126 characteristics with metrics. A practice in a source code element expresses a low-level rule and the reparation cost of violating this rule. The reparation cost of a source code element is calculated by the sum of the reparation costs of its rule violations. The quality of a source code element can be measured by the average cost per line. After calculating this raw value they convert it to a goodness value (A, B, C, D, E) using thresholds. Rule violations have an important role in our approach too, but besides the number of serious and medium rule violations in a source code element we consider other source code metrics as well. Our algorithm does not measure reparation costs, but the extent to which the overall maintainability of the system is being influenced by a source code element.

III. Approach

In this section we will present an approach that is an extension of our earlier research achievement concerning software quality models [5]. First, we will briefly introduce our probabilistic software quality model, which can be used for measuring source code maintainability at system level. In our approach, the so-called *benchmark characteristics* play an important role, therefore, a separate subsection will be devoted to this issue. Finally, we will present the basic idea of how the relative maintainability index for individual source code elements can be calculated.

A. The Probabilistic Source Code Maintainability Model

Our probabilistic software quality model [5] is based on the quality characteristics defined by the ISO/IEC 9126 [1] standard. In our approach, the relations between quality attributes and characteristics at different levels are represented by an acyclic directed graph, called the *attribute dependency graph (ADG)*. The nodes at the lowest level (i.e. without incoming edges) are called *sensor node*s, while the others are called *aggregate node*s. Figure 1 shows an instance of the applied ADG. The description of the different quality attributes can be found in Table I.

The sensor nodes in our approach represent source code metrics that can be readily obtained from the source code.

Table I
THE QUALITY PROPERTIES OF OUR MODEL

Sensor nodes	
McCabe	McCabe cyclomatic complexity [6] defined for the methods of the system.
CBO	Coupling between object classes, which is defined for the classes of the system.
NII	Number of incoming invocations (method calls), defined for the methods of the system.
LLOC	Logical lines of code of the methods.
Error	Number of serious PMD [17] coding rule violations, computed for the methods of the system.[1]
Warning	Number of suspicious PMD coding rule violations, computed for the methods of the system.[1]
CC	Clone coverage [18]. The percentage of copied and pasted source code parts, computed for the methods of the system.
Aggregated nodes defined by us	
Code complexity	Represents the overall complexity (internal and external) of a source code element.
Comprehension	Expresses how easy it is to understand the source code.
Fault proneness	Represents the possibility of having a faulty code segment.
Effectiveness	Measures how effectively the source code can be changed. The source can be changed effectively if it is easy to change and changes will likely not have unexpected side-effects.
Aggregated nodes defined by the ISO/IEC 9126	
Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified.
Changeability	The capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Testability	The capability of the software product to enable modified software to be validated.
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

In the case of a software system, each source code metric can be regarded as a random variable that can take real values with particular probability values. For two different software systems, let $h_1(t)$ and $h_2(t)$ be the probability density functions corresponding to the same metric. Now, the *relative goodness value* (from the perspective of the particular metric) of one system with respect to the other, is defined as

$$\mathcal{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\, \omega(t)\, dt,$$

where $\omega(t)$ is the weight function that determines the notion of goodness, i.e. where on the horizontal axis the differences matter more. Figure 2 helps us understand the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

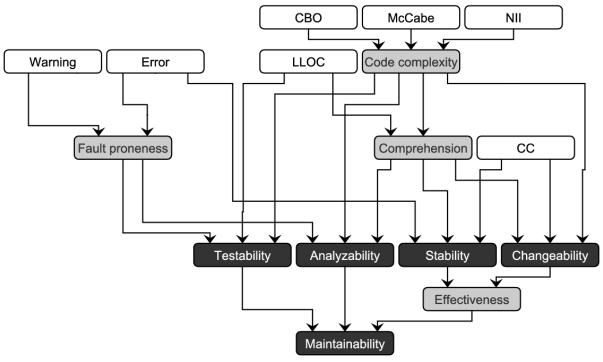

Figure 1. Java maintainability model

For a fixed probability density function h, $\mathcal{D}(h, \_)$ is a random variable, which is independent of any other particular system. We will call it the *absolute goodness*

[1]The full list of applied PMD rules is available online: http://www.inf.u-szeged.hu/~hpeter/SQM2013/PMD.xls

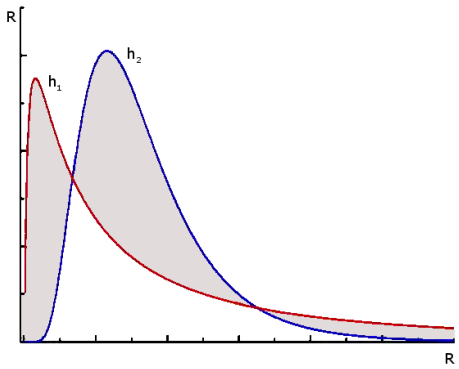

Figure 2. Comparison of probability density functions

of the system (from the perspective of the metric that corresponds to h). The empirical distribution of the absolute goodness can be approximated by substituting a number of samples for its second parameter, i.e. by making use of a repository of source code metrics of other software systems. We used the repository introduced earlier [5], containing the metric results of 100 Java systems. The probability density function of the absolute goodness is called the *goodness function*. The expected value of the absolute goodness will be called the *goodness value*. Following the path described above, the goodness functions for the sensor nodes can be easily computed.

For the edges of the ADG, a survey was prepared, where the IT experts and researchers who filled it were asked to assign weights to the edges, based on how they felt about the importance of the dependency. They were asked to assign scalars to incoming edges of each aggregate node, such that the sum is equal to one. Consequently, a multi-dimensional random variable $\vec{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ will correspond to each aggregate node v. We define the aggregated goodness function for the

node v in the following way:

$$g_v(t) = \int_{\substack{t=\vec{q}\vec{r} \\ \vec{q}=(q_1,\ldots,q_n)\in\Delta^{n-1} \\ \vec{r}=(r_1,\ldots,r_n)\in C^n}} \vec{f}_{\vec{Y}_v}(\vec{q}) g_1(r_1)\ldots g_n(r_n) d\vec{r} d\vec{q}, \quad (1)$$

where $\vec{f}_{\vec{Y}_v}(\vec{q})$ is the probability density function of $\vec{Y}_v$, $g_1, g_2, \ldots g_n$ are the goodness functions corresponding to the incoming nodes, Δ^{n-1} is the $(n-1)$-standard simplex in $\Re^n$ and C^n is the standard unit n-cube in $\Re^n$.

Although the formula may look frightening at first glance, it is just a generalization of how aggregation is performed in the classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the aggregate node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. Now, we are able to compute goodness functions for each aggregate node; in particular the goodness function corresponding to the *Maintainability* node as well.

B. Benchmark Characteristics

After a benchmark containing several systems is available and a particular model is defined, the goodness values for each software in the benchmark can be calculated. In this way – for a particular node in the ADG – several goodness values can be obtained, which can actually be considered as a sample of a random variable with normal distribution based on empirical observations. As each system is compared to every other system twice (with opposite signs), the expected value of the distribution is necessarily zero. The distribution functions obtained in this way are called the *benchmark characteristics*. Figure 3 shows an example of a benchmark characteristic for the Maintainability node.

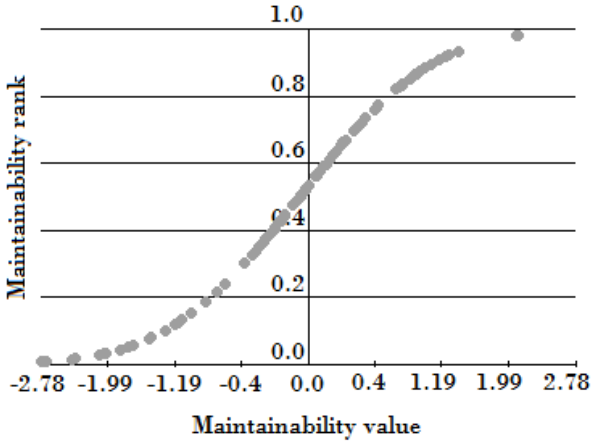

Figure 3. Benchmark characteristic for Maintainability

The characteristic functions map the $(-\infty, \infty)$ interval to the $(0, 1)$ interval. Therefore, we use these functions to transform the goodness values of a particular software system being evaluated to the $(0, 1)$ interval. In this way, a common scale is obtained for the goodness values of different systems; i.e. they become comparable to each other. The normalized goodness value is basically the proportion of all the systems within the benchmark whose goodness values are smaller.

C. The Drill-Down Approach

The above approach is used to obtain a system-level measure for source code maintainability. Our aim is to drill down to lower levels in the source code and to get a similar measure for the building blocks of the code base (e.g. classes or methods). For this, we define the *relative maintainability index* for the source code elements, which measures the extent to which they affect the system level goodness values. The basic idea is to calculate the system level goodness values, leaving out the source code elements one by one. After a particular source code element is left out, the system level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element, will be called the *relative maintainability index* of the source code element itself. The relative maintainability index is a small number that is either positive when it improves the overall rating or negative when it decreases the system level maintainability. The absolute value of the index measures the extent of the influence to the overall system level maintainability. In addition, a relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to different extents.

Calculating the system-level maintainability is computationally very expensive. To obtain the relative indices, it is enough to compute only the goodness values for each node in the ADG; one does not need to construct the goodness functions. Luckily, computing the goodness values without knowing the goodness functions is feasible. It can be shown that calculating goodness functions and taking their averages is equivalent to using only the goodness values throughout the aggregation.

In the following, we will assume, that $\omega(t)$ is equal to t for each sensor node, which means that e.g. twice as high metric value means twice as bad code. While this linear function might not be appropriate for every metric, it is a very reasonable weight function considering the metrics used by the quality model. However, the presented approach is independent of the particular weight function used, and the formalization can be easily extended to different weight functions. Next, we will provide a step-by-step description of the approach for a particular source code element.

1) For each sensor node n, the goodness value of the system without the source code element e can be calculated via the following formula:

$$g_{rel}^{e,n} = \frac{K g_{abs}^n + m}{K-1} - \frac{1}{N}\sum_{j=1}^{N} \frac{M_j}{K-1}$$

where g_{abs}^n is the original goodness value computed for the system, m is the metric value of the source

code element corresponding to the sensor node, K is the number of source code elements in the system for which the sensor node is considered, N is the number of the systems in the benchmark, and $M_j\,(j = 1, \ldots, N)$ are the averages of the metrics for the systems in the benchmark.

2) The goodness value obtained in this way is transformed to the $(0, 1)$ interval by using the characteristic function of the sensor node n. For simplicity reasons, we assume that from now $g_{rel}^{e,n}$ stands for the transformed goodness value and it will be referred to as goodness value as well.
3) Due to the linearity of the expected value of a random variable, it can be shown that Formula 1 simplifies to a linear combination, provided that only the expected value needs to be computed. Therefore, the goodness value of an aggregate node n can be computed in the following way:

$$g_{rel}^{e,n} = \sum_i g_{rel}^i E\left(Y_v^i\right)$$

where $g_{rel}^i\,(i = 1, \ldots)$ are the transformed goodness values of the nodes that are on the other sides of the incoming edges and $E\left(Y_v^i\right)$ is the expected value of the votes on the i^{th} incoming edge. Please note that since $\sum_i E\left(Y_v^i\right) = 1$, and $\forall i, g_{rel}^i \in (0, 1)$, the value of $g_{rel}^{e,n}$ will always fall into the $(0, 1)$ interval, i.e. no transformation is needed at this point.
4) The relative maintainability index for the source code element e and for a particular ADG node n is defined as

$$g_{idx}^{e,n} = g_{abs}^n - g_{rel}^{e,n}$$

The relative maintainability index measures the effect of the particular source code element on the system level maintainability computed by the probabilistic model. It is important to notice that this measure determines an ordering among the source code elements of the system, i.e. they become comparable to each other. And what is more, the system level maintainability being an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. In other words, computing all the relative indices for each software system in the benchmark will give rise to an absolute ordering among them.

IV. Empirical Validation

We evaluated the approach on the jEdit v4.3.2 open source text editor tool (http://www.jedit.org/), by considering a large number of its methods in the source code.[2] The basic properties of jEdit's source code and the selected methods are shown in Table II. These and all other source code metrics were calculated by our *Columbus tool* [19]. A considerable number of students were asked to rate the maintainability and lower level quality aspects of 191 different methods. Here, we reused the data of our earlier empirical case study [20], where over 200 students manually evaluated the different ISO/IEC 9126 quality attributes of the methods in the jEdit tool. Even though we conducted a preliminary survey with IT experts, the amount of collected data was insufficient, therefore we chose to use the student evaluation for validation purposes. For the empirical validation, the averages of students' votes were taken and they were compared to the series of numerical values (i.e. relative maintainability indices) computed by the approach. A repeated study using the median of the votes yielded very similar results.

Besides the manual evaluation, the relative maintainability indices for the same characteristics have been calculated based on the drill-down approach presented in the previous section. The Spearman's rank correlation coefficient was determined for these two series of numbers. This coefficient can take its values from the range $[-1, 1]$. The closer this value is to 1, the higher is the similarity between the manual rankings and the automatically calculated values.

Table II
Basic metrics of the source code of jEdit and the evaluated methods

Metrics	Value
Logical Lines of Code (LLOC)	93744
Number of Methods (NM)	7096
Number of Classes (NCL)	881
Number of Packages (NPKG)	49
Number of evaluated methods	191
Average number of LLOC for the evaluated methods	26.41
Average number of McCC complexity for the evaluated methods	5.52

A. Manual Evaluation

The students who took part in the evaluation were third year undergraduate students and completed a number of programming courses. Some of them already had some industrial experience as well. Each of the students were asked to rank the quality attributes of 10 different methods of jEdit v4.3.2 subjectively. Altogether 191 methods have been shared out among them. For the evaluation, a web-based graphical user interface was constructed and deployed, which provided the source code fragment under evaluation together with the questionnaire about the quality properties of the methods.

The methods for the manual evaluation were selected randomly. We assigned the methods to students in such a way that each method was always supposed to be evaluated by at least ten persons. The students were asked to rate the subcharacteristics of maintainability defined by the ISO/IEC 9126 standard on a scale of zero to ten; zero means the worst, while ten is the best. These attributes were the following: *analyzability*, *changeability*, *testability*, *stability*) and a new quality attribute – *comprehensibility* – defined by us earlier [9]. More details regarding the data collection process and the used web application can be found in our previous work [9].

[2]https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/tags/jedit-4-3-2

Table IV
PEARSON'S CORRELATIONS AMONG THE SOURCE CODE METRICS AND STUDENTS' OPINIONS

Metric	Comprehensibility	Analyzability	Changeability	Stability	Testability	Maintainability
CC	0.07	0.07	0.06	0.05	0.06	0.09
LLOC	-0.45	-0.50	-0.46	-0.44	-0.41	-0.52
McCC	-0.33	-0.37	-0.34	-0.32	-0.29	-0.38
NII	-0.04	-0.03	-0.02	-0.08	-0.03	-0.03
Error (serious rule viol.)	-0.14	-0.11	-0.07	-0.10	-0.08	-0.16
Warning (suspicious rule viol.)	-0.30	-0.32	-0.30	-0.26	-0.30	-0.30

Table III
STATISTICS OF THE STUDENT VOTES

Quality attribute	Avg. std.dev.	Max. std.dev.	Min. std.dev.
Analyzability	1.87	3.93	0.00
Comprehensibility	1.89	4.44	0.44
Stability	2.22	4.31	0.53
Testability	2.04	3.82	0.32
Changeability	2.01	3.62	0.00
Maintainability	1.97	3.93	0.00

Table III contains some basic statistics of the collected student votes. The first column shows the average standard deviation values of student votes for the different quality attributes. The values vary between 1.8 and 2.2 which indicates that the students had an acceptable level of inter-rater agreement in general. The next two columns show the maximum and minimum of the standard deviations. The maximums are approximately two times higher than the averages, but the minimum values are very close to zero. For analyzability, changeability and maintainability attributes the minimum is exactly zero, meaning that there was at least one method that got exactly the same rating from each student.

Table IV shows the Pearson's correlation coefficients for the different metric values used in our quality model (see Figure1) and the average votes of the students for the high-level quality attributes (the CBO metric is not listed because it is a metric defined for classes and not for methods). Based on the data, a number of observations can be made:

- All of the significant Pearson's correlation coefficients (R values) are negative. This means that the greater the metric values are, the worse are the different quality properties. This is in line with our expectations, as lower metrical values are usually desirable, while higher values may suggest implementation or design related flaws.
- The quality attributes correlate mostly with the logical lines of code (LLOC) and the McCabe's cyclomatic complexity (McCC) metrics. The reason for this might be that these are very intuitive and straightforward metrics that can be observed locally, by looking only at the code of the method's body.
- As an analogy to the previous observation, being hard to interpret the metrics locally, the clone coverage (CC) and the number of incoming invocations (NII) metrics show the lowest correlation.
- The number of rule violations also shows a noticeable correlation with the quality ratings. Surprisingly, the suspicious rule violations show a higher correlation than the really serious ones. The reason for this might be that either the students considered different violations as serious or the fact that the number of the most serious rule violations was low (five times lower than suspicious violations), which may have biased the analysis.

B. Model-Based Evaluation

We calculated the quality attributes for all the methods of jEdit by using the implementation of the algorithm presented in Section III. The relative maintainability indices are typically small positive or negative numbers. The negative values indicate negative impact on the maintainability of the system while positive indices imply positive effect. As we are mainly interested in the order of the methods based on their impact on the maintainability, we assigned an integer rank to every method by simply sorting them according to their relative maintainability index in a decreasing order. The method having the largest positive impact on the maintainability gets the best rank (number 1) while the worst method gets the worst rank (which equals to the number of methods in the system). Therefore, the most critical elements will be at the end of the relative maintainability based order (i.e. larger rank means worse maintainability).

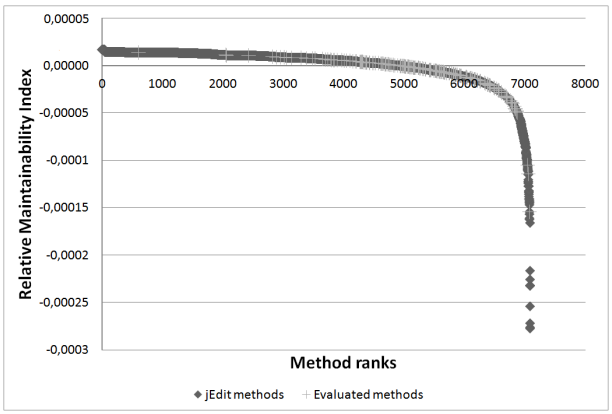

Figure 4. The relative maintainability indices and corresponding ranks

Figure 4 depicts the relative maintainability indices of the methods of jEdit and their corresponding ranks. It can be seen that there are more methods that increase

the overall maintainability than those which decrease it. However, methods having positive impact only slightly improve the overall maintainability, while there are about 500 methods that have a significantly larger negative impact on the maintainability. In principle, these are the most critical methods that should be improved first, to achieve a better system level maintainability.

Figure 5 shows the density function of the computed relative maintainability indices. In accordance with the previous observations, we can see that there are more methods that increase the maintainability, however, their maintainability index values are close to zero. This means that they have only a small positive impact. The skewed left side denotes that there are a smaller number of methods decreasing the maintainability but they have a significantly larger negative effect (their index is farther out from zero). For evaluating the effectiveness of our

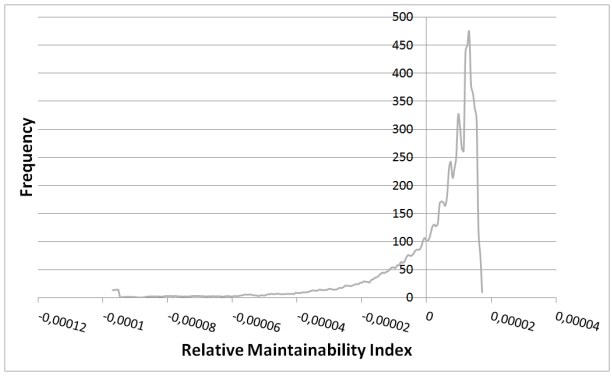

Figure 5. The density function of the relative maintainability indices

approach, we took the calculated quality attributes for each of the manually evaluated methods. Our conjecture was that the model-based assessment of quality rankings will not differ much from the manually obtained values. If this proves to be true, a list of the most critical methods could always be generated automatically, which would correlate well with the opinion's of the developers.

As we are interested in the similarity between the rankings produced by our algorithm and the rankings obtained with the manual evaluation, we performed a Spearman's rank correlation analysis. The Spearman's correlation coefficient is defined as the Pearson's correlation coefficient between the ranked variables. Identical values (rank ties or value duplicates) are assigned a rank equal to the average of their positions in the ascending order of the values.

First, we analyzed the relationship between the quality ranking calculated by our algorithm and the rankings assigned by the students. The Spearman's correlation coefficients and the corresponding p-values can be seen in Table V. In statistical hypothesis testing, the p-value is the probability of obtaining a test statistic that is at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. In our case, the null hypothesis will be that there is no relation between the rankings calculated by the algorithm and the one obtained by manual evaluations.

Table V
SPEARMAN'S CORRELATION VALUES AMONG THE RELATIVE MAINTAINABILITY INDICES AND THE MANUAL EVALUATIONS

Quality attribute	Correlation with students' opinions (R value)	p-value
Analyzability	0.64	<0.001
Comprehensibility	0.62	<0.001
Changeability	0.49	<0.001
Stability	0.49	<0.001
Testability	0.61	<0.001
Maintainability	**0.68**	**<0.001**

As can be seen, the R values of the correlation analysis are relatively high for each quality attribute. All the values are significant at the level of 0.001. This means that there is a significant relationship between the automatically obtained rankings and the one derived from the students' evaluations. The best correlation was found between the data series of the Maintainability characteristic. According to the results we can automatically identify those critical source code elements that decrease the system's maintainability the most. These are the source code elements at the end of the ranked list (having the worst relative maintainability indices). The list of critical elements is crucial in order to improve the quality of a system, or at least to decrease the rate of its erosion.

Although the results are promising in their current form, we went towards tracing down the differences between the manually and automatically obtained rankings. We collected and manually examined several methods that had the largest differences in their rankings. Table VI lists the assessed methods, their rankings and some of their most important metrical values. In the following, we will provide a more detailed explanation regarding the differences, considering the methods one-by-one.

• *org.gjt.sp.jedit.bsh.ClassGeneratorImpl. invokeSuperclassMethodImpl(BshClassManager, Object, String, Object[])*

This method attempts to find and invoke a particular method in the superclass of another class. While our algorithm found this method rather easy to maintain, the students gave it a very low ranking. Despite the fact that syntactically this program fragment is very simple it uses Java reflection which is by its nature difficult to read and comprehend by humans.

• *org.gjt.sp.jedit.buffer.JEditBuffer. fireContentInserted(int, int, int, int)*

This method is a fairly simple function at a first glance, which fires an event to each listener of an object. On the other hand, from the maintainability point of view, changing the method might be risky as it has four clones (copy&paste). All the fire events are duplications of each other. The code contains also a medium rule violation; a catch block that catches all the *Throwable* objects. It can hide problems like runtime exceptions or errors. Nevertheless, human evaluators tend to give more significance to local properties, like the lines of code

Table VI
THE LARGEST DIFFERENCES BETWEEN THE AUTOMATIC AND MANUAL RANKINGS

Method name	Students' ranking	Model ranking	Rank difference	CC	LLOC	McCabe	NII	Rule violations
invokeSuperclassMethodImpl	177	57	120	0	17	2	1	0
fireContentInserted	29	139	110	1	18	3	2	1
fireEndUndo	13	121	108	1	15	3	0	1
move	168	66	102	0	16	3	0	0
fireTransactionComplete	42	142	100	1	16	3	5	1
read	113	16	97	0	10	2	0	0

or McCabe's complexity, because it is hard to explore the whole environment of the code fragment. From this respect, the method is indeed a well-maintainable code.

• *org.gjt.sp.jedit.Buffer.fireEndUndo()*

This is exactly the same type of a fire method than the previous one. Therefore, the same reasoning holds in this case as well.

• *org.gjt.sp.jedit.browser.VFSBrowser.move(String)*

This method is responsible for moving a toolbox. It is short and has low complexity, therefore our algorithm ranked it as a well-maintainable code. However, human evaluators found it hard to maintain. The code is indeed a little bit hard to read, because of the unusual indentation (i.e. every expression goes to new line) of the logical operators which might be the cause of the low human ranking.

• *org.gjt.sp.jedit.buffer.JEditBuffer.fireTransactionComplete()*

This is another method responsible for firing events, just like the earlier ones. Therefore the same reasoning holds in this case as well.

• *org.gjt.sp.jedit.bsh.CommandLineReader.read(char[], int, int)*

This method reads a number of characters and puts them into an array. The implementation itself is short and clear, not complex at all, therefore our algorithm ranked it as well-maintainable. However, the comment above the method starts with the following statement: "This is a degenerate implementation". Human evaluators probably read the comment and marked the method as hard to maintain.

The manual assessment shed light to the fact that human evaluators tend to take into consideration a wider range of source code properties than the presented quality model. These properties are e.g. code formatting, semantic meaning of the comments or special language constructs like Java reflection. The automatic quality assessment should be extended with measurements of these properties to achieve more precise results.

On the other hand, the automatic quality assessment may take advantage from the fact that it is able to take the whole environment of a method into account for maintainability prediction. We found that human evaluators had difficulties in discovering non-local properties like clone fragments or incoming method invocations. Another issue was that while the algorithm was considering all the methods at the same time, the human evaluators assigned their ranks based only on the methods they evaluated. For example, while the best method gets a maximal score from the model, the evaluators may not recognize it as the best one, as they have not seen all the others.

We also analyzed the correlations between the lower level quality attributes calculated by the algorithm and those assigned by the students. Figure 6 shows the diagram of the correlations. The names of the quality attributes are shown in the diagonal. The quality attributes starting with the *Stud_* prefix refer to the ones resulting from the manual evaluation; the rest of the attributes are computed by the model. In the upper right triangle of the diagram, the Spearman's correlation values of the different attributes can be seen. On the side, a graphical view of the correlation coefficients is presented, where the darker shade means a higher correlation.

Based on the diagram, the following notable observations can be made:

- The dark triangle in the upper left corner shows that there is a very high correlation among the quality attributes calculated by our model. This is not surprising, since the model is hierarchical and the higher level attributes depend on the lower level ones.
- Similarly, the lower right corner shows a quite high correlation among the quality attributes evaluated by the students. However, the correlation coefficients are smaller than the coefficients among the attributes of the model. This suggests that students evaluated the different quality properties somewhat independently from each other, not following the ISO/IEC 9126 standard's hierarchy strictly.
- Interestingly, the maintainability property evaluated by the students shows a slightly higher correlation with the algorithm-based approximation of comprehensibility and testability than with the maintainability value of the model. The reason might be that the comprehensibility and testability of the code are more exact concepts than the others, like stability, analyzability, etc. While comprehensibility is easier for the students to understand or evaluate, it makes them prone to equate maintainability and comprehensibility.
- It is promising that the model-based maintainability attribute shows the highest correlation with the maintainability property among the manually assessed ones. It means that our model interprets the quality on a broader scale than students do, i.e. it takes more

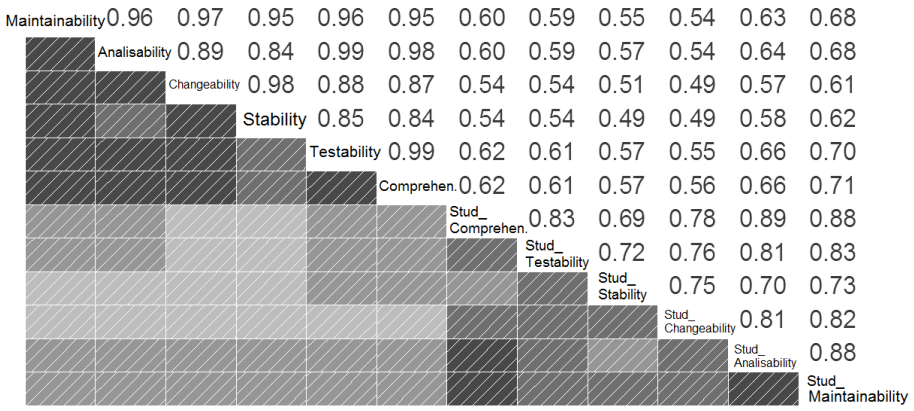

Figure 6. Correlations of the calculated and the manually assigned quality attributes

factors into consideration. This might be because the students do not take into account all the hard-to-get-concepts of the ISO/IEC 9126 standard. According to our previous observation, the students tend to care the most about comprehensibility only. This is in line with the fact that students prefer locally interpretable properties, like the lines of code or McCabe's complexity more.

V. Threats to Validity, Limitations

Our validation is built on an experiment performed by students. This is a threat to the validity of the presented work due to the possible lack of their expertise and theoretical background. To compensate this drawback we selected third year undergraduate students who completed preliminary studies on software engineering and maintenance. Additionally, at least ten evaluations have been collected for each method to neutralize the effect of the possibly higher deviation in their votes.

For efficiency reasons we did not construct the goodness functions (probabilistic distributions, see Section III) but computed only the goodness values for each node to obtain the relative indices. This might seem a loss of information, however, to get an ordering among the methods we should have derived the goodness values from the goodness functions anyway. Therefore, calculating only the goodness values is enough for our purposes.

The presented approach described in Section III assumes that the $\omega(t)$ weight function is equal to t for each sensor node (e.g. twice as high metric value means twice as bad code). This is not necessarily applicable for all the metrics (e.g. the very low and very high values of the Depth of Inheritance Tree metric is considered to be bad while there is an optimal range of good values). Our approach could be generalized to handle arbitrary weight functions, however, it would result a much more complex model. We decided to keep our approach simple and easy to understand as for most of the metrics (especially the ones included in our model) the applied weight function is reasonable.

The results might not be generalizable as we examined only 191 methods of one system. It required a huge amount of manual effort; performing more studies would require an even larger investment. This is actually, why the automatic maintainability analysis of source code elements is important. Despite the relatively small amount of empirical data, we consider the presented results as an important step towards the validation of our model.

VI. Conclusion and Future Work

In this paper we presented the continuation of our previous work [5] on creating a model suitable for measuring maintainability of software systems according to the ISO/IEC 9126 standard [1] based on its source code. Here, we further developed this model to be able to measure also the maintainability of individual source code elements (e.g. classes, methods). This allows the ranking of source code elements in such a way that the most critical elements can be listed, which enables the system maintainers to spend their resources optimally and achieve maximum improvement of the source code with minimum investment. We validated the approach by comparing the model-based maintainability ranking with the manual ranking of 191 Java methods of the jEdit open source text editor tool. The main results of this evaluation are:

- The manual maintainability evaluation of the methods performed by more than 200 students showed a high, 0.68 Spearman's correlation at $p < 0.001$ significance level with the model-based evaluation.
- Some of the differently ranked methods were manually revised and it turned out that humans take into consideration a wider range of properties, while the model is able to explore the environment more effectively.
- The Spearman's correlation of ISO/IEC 9126 attributes (model vs. manual) is high in general. This is especially true in the case of the maintainability property (as already mentioned).

As a future work we would like to perform more case studies to be able to generalize our findings. The current evaluation took only methods into account. We would like also to carry out a case study on manually evaluating the maintainability of classes and correlating the results with our drill-down approach.

ACKNOWLEDGEMENTS

This research was supported by the Hungarian national grants GOP-1.1.1-11-2011-0038 and GOP-1.1.1-11-2011-0006.

REFERENCES

[1] ISO/IEC, *ISO/IEC 9126. Software Engineering – Product quality*. ISO/IEC, 2001.

[2] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey, "A Maintainability Model for Industrial Software Systems Using Design Level Metrics," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 248–256.

[3] J. Bansiya and C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, 2002.

[4] M. Azuma, "Software Products Evaluation System: Quality Models, Metrics and Processes – International Standards and Japanese Practice," *Information and Software Technology*, vol. 38, no. 3, pp. 145–154, 1996.

[5] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A Probabilistic Software Quality Model," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. Williamsburg, VA, USA: IEEE Computer Society, 2011, pp. 368–377.

[6] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, July 1976.

[7] ISO/IEC, *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[8] E. Bagheri and D. Gasevic, "Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics," in *Software Quality Journal 19(3):579-612*. Springer, 2011.

[9] P. Hegedűs, T. Bakota, L. Illés, G. Ladányi, R. Ferenc, and T. Gyimóthy, "Source Code Metrics and Maintainability: a Case Study," in *Proceedings of the 2011 International Conference on Advanced Software Engineering And Its Applications (ASEA 2011)*. Springer-Verlag CCIS, Dec. 2011, pp. 272–284.

[10] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pp. 30–39, 2007.

[11] "Software Improvement Group," `http://www.sig.eu/en/`.

[12] T. L. Alves, C. Ypma, and J. Visser, "Deriving Metric Thresholds from Benchmark Data," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, 2010.

[13] J. P. Correia and J. Visser, "Benchmarking Technical Quality of Software Products," in *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–300.

[14] R. Baggen, K. Schill, and J. Visser, "Standardized Code Quality Benchmarking for Improving Software Maintainability," in *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

[15] J. P. Correia, Y. Kanellopoulos, and J. Visser, "A Survey-based Study of the Mapping of System Properties to ISO/IEC 9126 Maintainability Characteristics," *IEEE International Conference on Software Maintenance (ICSM 2009)*, pp. 61–70, 2009.

[16] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The Squale Model – A Practice-based Industrial Quality Model," in *Proceedings of the 25rd International Conference on Software Maintenance (ICSM 2009)*. IEEE Computer Society, 2009, pp. 531–534.

[17] "The PMD Homepage," `http://pmd.sourceforge.net/`.

[18] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society, 1998, pp. 368–377.

[19] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – Reverse Engineering Tool and Schema for C++," in *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society, Oct. 2002, pp. 172–181.

[20] P. Hegedűs, G. Ladányi, I. Siket, and R. Ferenc, "Towards Building Method Level Maintainability Models Based on Expert Evaluations," in *Proceedings of the 2012 International Conference on Advanced Software Engineering And Its Applications (ASEA 2012), accepted, to appear*. Springer-Verlag CCIS, 2012.

A Meta Model for Software Architecture Conformance and Quality Assessment

Andreas Goeb
Applied Research
SAP AG
Darmstadt, Germany
Email: andreas.goeb@sap.com

***Abstract*—Software architecture and design suffer from a lack of documented knowledge on how different architectural styles influence software quality. Existing software quality models do not allow engineers to evaluate whether a given software system adequately implements the basic principles of the chosen architectural style, and which architectural properties and best practices beyond these principles contribute to the system's quality. In this paper, I present a meta quality model for software architectures, which can be used not only as a knowledge-base to easily compare architectural styles based on their impact on software quality, but also to increase efficiency of architectural quality analysis by leveraging existing modeling concepts and tools. An experiment performing an architecture assessment using a quality model for the SOA architectural style not only showed that the approach is applicable in practice, but also indicated a reduction of manual effort compared to other architecture assessment approaches.**

***Keywords*-Quality Model; Software Architecture; Design; Conformance**

I. Introduction

A. Motivation

Current trends in the software market show that quality becomes a differentiating factor among software products with decreasing functional diversification. It is widely accepted that software quality problems can be handled easier and more cost efficient, the earlier they are detected during development [1], [2]. In particular, almost all of a software system's quality attributes are influenced by its architecture [3, p. 19]. Consequently, it is particularly relevant for software engineering research and practice to develop means for efficient software quality assessment on the architectural level.

B. Problem

According to Svahnberg and Wohlin [4], there is a lack of documented knowledge on how different architectural styles influence software quality. This forces software architects to base the selection of an architectural style purely on personal experience rather than objective information.

Moreover, software architecture evaluation generally requires a considerable amount of manual effort, because most established techniques are based on the manual analysis of scenarios. In particular with the emergence of new deployment models in the context of cloud applications, where small increments of updated functionality are delivered in very short periods of time, this approach becomes impractical in the long term. Because of the highly individual nature of these techniques, they are not designed to be applied repeatedly. Moreover, applying these techniques to more than one software project involves individual preparation effort for all of them. Within the software development process, this also implies that quality assessment approaches used in the architecture phase fundamentally differ from those used in the implementation phase, leading to media discontinuities during quality assurance. This complicates continuous quality monitoring and control and therefore negatively impacts the costs of quality assurance.

C. Contribution

In this paper, I present an architecture-specific extension to the Quamoco meta quality model [5]. The proposed quality model structure explicitly separates *conformance* concepts from *design* best-practices and can be used both to derive statements on the relationships between architectural properties and product quality, and to increase efficiency of architecture quality analysis due to large automation potential. While building architecture quality models according to this approach still involves much manual work and expert knowledge, these models can be used to repeatedly evaluate software architectures. Moreover, the investment of building such models pays off when they are reused to evaluate a larger number of software systems.

D. Structure

The remainder of this paper is structured as follows: Section II summarizes related work according to software quality models and architecture evaluation approaches. In Section III, I introduce architecture-specific additions to the Quamoco meta quality model. Section IV explains the contents of an architecture conformance and quality model, in particular quality goals, architectural principles, general design properties, and corresponding measures, as well as the overall approach of building architecture quality models

and using them for architecture evaluation. In Section V, the approach is experimentally applied by conducting an architecture evaluation, using a quality model for service-oriented architectures as an example for a specific architectural style. Finally, Section VI concludes the paper and outlines directions for future work.

II. Related Work

A. Software Quality Models

Modeling software quality has been a topic addressed by researchers for several decades. Early quality models date back to the late 1970s, e. g. those by Boehm [6] or McCall [7], which hierarchically decompose software quality into more tangible concepts. This approach has led to international standards for software quality like ISO 9126 or its successor ISO 25010, which are reported to be used as a basis for software quality assurance in many software companies. However, recent studies also show that these standards are too abstract to be directly used for quality assessment [8]. Because of this shortcoming, there have been several initiatives in defining models that not only describe software quality, but can also be used to assess software systems with regard to their quality. The Squale project enhanced the ISO 9126 quality model with so-called *practices* containing information on how to measure certain quality characteristics [9]. They also provide tools to assess source code quality in different programming languages. In summary, software quality models help in developing a common understanding of software quality. Some models can be used for automatic quality assessment, lowering the effort compared to inspection-based approaches. Most of these assessment models, however, are targeted at low-level source code constructs only, not taking architectural properties into account.

B. Software Architecture Evaluation

Software architecture is crucial for software quality. This means that the decision for a particular architecture is highly dependent on the quality goals of the involved stakeholders. Clements et al. [3] phrase this very concisely: "If the sponsor of a system cannot tell you what any of the quality goals are for the system, then any architecture will do."

If architectural decisions are so important for the quality of a software system, architecture assessment appears to be a feasible means to provide statements about its quality. To accomplish this, architecture evaluation methods have been developed, which follow clear rules and provide a certain degree of repeatability. Clements et al. categorize these methods according to the tools they use: *Questioning* methods are based on scenarios, questionnaires, and checklists for system analysis. These are often used in conjunction with system documentation or specification, thereby not requiring the system to be already completely implemented. In contrast, *measuring* methods directly analyze the respective system by means of automatic analysis tools, e. g. calculating software metrics or simulating system behavior. In any case this second group of methods requires the presence of software artifacts and can therefore not be applied as early as scenario-based methods.

Clements et al. [3] propose three scenario-based methods, namely *SAAM*, *ARID*, and *ATAM*. They all start with the elicitation of scenarios in workshops. A scenario might be: "In addition to local deployment and operation, you should also be able to operate the system on a cloud platform". Based on a prioritized list of such scenarios, different architecture alternatives are then evaluated regarding their ability to facilitate these scenarios. Depending on the method and the particular situation, different techniques can be used, e. g. sensitivity and tradeoff analysis in ATAM, scenario walk-throughs in SAAM, or Active Design Reviews [10] in ARID. A comparison of these methods is shown in [3, p. 256]. The authors state that a mid-size architecture evaluation using ATAM would take around 70 person days, assuming a reasonable team size.

Vogel [11] presents a general overview on architecture evaluation methods. Moreover, Zhao [12] provides links to further literature. It is generally observed that according to the classification above, the overwhelming majority of architecture evaluation methods belong to the group of questioning methods, thus requiring large amounts of manual effort. This might be due to the fact that architecture analysis is generally performed in a project-specific context with individual requirements. For domain and project-independent parts of the analysis, tool supported approaches are available, e. g. ConQAT[1] can automatically compare dependencies between components of a software system with the architecture specification and visualize the results.

Losavio et al. [13], [14] present an architecture measuring approach for evaluating software quality according to ISO 9126. They consecutively walk through all quality characteristics and sub-characteristics contained in the ISO standard and present measurement specifications in order to quantify these on an architectural level. In total, they present 16 *attributes* and associated *metrics*. Out of these, nine are defined in a binary fashion and require identifying whether there is a mechanism in the architecture to support the respective sub-characteristic, e. g. *co-existence* is determined by "the presence of a mechanism facilitating the co-existence" [13]. Three of the remaining metrics are defined to aggregate the respective values from the individual components and connectors. In particular, no further adjustments to these individual scores are made based on the architecture, e. g. *maturity* is defined as the sum of the maturities of all components and connectors the architecture consists of [13]. In conclusion, the proposed approach provides a unified process framework for architecture quality assessment. Since

[1] https://www.conqat.org

over half of the metrics are Boolean and require thorough expert assessment, the approach has to be considered mainly checklist-based. Most of the measures are defined on a high granularity that makes it difficult to automate measurement by implementing the proposed metrics in a tool.

In summary, most approaches for software architecture evaluation either do not provide a way to automate assessment steps or require executable software artifacts in order to do so. Although some scenario-based approaches offer sophisticated methodology to support project-specific architectural decisions, none of the existing approaches provides a way to quickly obtain a general statement of the overall conformance and quality of a software architecture.

III. Basic Modeling Concepts

This section briefly describes the meta quality model that we developed in the Quamoco project [5], [15]. It addresses the shortcomings of related approaches in software quality modeling presented in Section II-A. Because this meta model provides the basis for the architecture extensions proposed in Section IV, its elements are introduced in the following.

Entities provide a decomposition of the software product. Starting from the complete Product, entities can refine other entities along an *is-a* or a *part-of* relation, e. g. both entities Source Code and Documentation refine Product. Decomposition is usually performed as required, until a depth is reached that is sufficiently detailed to describe the desired concepts. The entity Return Parameter (of a method) would refer to Parameter using the is-a relation. In turn, the parameter would specify that it is part of a Method, which is in turn part of an Interface. Note that entities describe things on a conceptual level, not individual instances within an assessed system (i. e. the return parameter of a certain operation).

These entities are characterized by *attributes* (e. g. ATOMICITY, PRECISE NAMING, COHESION) in order to define *product factors*. These factors describe observable properties of these entities that may or may not be present in a particular system to a certain degree. The degree is expressed in the factor's *value range*, which includes all real numbers from 0 to 1. The factor [Service | PRECISE NAMING] is completely fulfilled (thus evaluating to 1.0) for a system, whose services are all named precisely.

The Quamoco meta quality model allows for several kinds of *quality aspects* in order to cover a wide range of established ways of decomposing product quality. Wagner et al. [15] structure product quality using the quality characteristics defined in ISO 25010. Other possible quality aspect forms include activity-based quality goals (c. f. [16]). These have been proposed in order to provide more natural and meaningful decomposition semantics, and are therefore used in the following. Activity-based quality aspects are comprised of an *activity*, which is performed on or with the system, and an *attribute* characterizing this activity. A typical quality goal from a service consumer's perspective is the efficient analysis of the functionality provided by a service: [Analysis | EFFICIENCY]. The fact that the presence of a product factor in a software system affects the fulfillment of a quality goal is represented by an *impact*. Since the effect can be positive or negative, the impact is annotated with $+$ or $-$, respectively. For example, the idea that precise naming helps a user to analyze whether a service provides the functionality he needs is represented as: [Service | PRECISE NAMING] $\xrightarrow{+}$ [Analysis | EFFICIENCY].

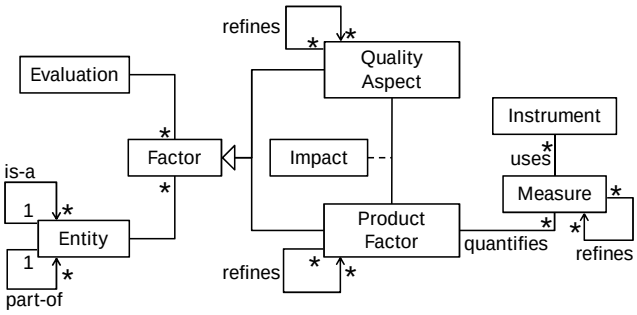

Figure 1. Quamoco Meta Quality Model (Source: [15])

In summary, product factors bridge the gap between abstract categories of quality and observable properties of software artifacts. In order to assess to which degree a factor is fulfilled for a particular system, the quality model contains *measures*. They provide means to actually obtain data about the system. Depending on the particular technology or programming language used in the software product, these measures make use of different kinds of *instruments*, either tool-based ones using static analysis tools and metrics, or manual ones by defining steps for an inspection of the respective entities. For aggregation purposes, *evaluations* translate the values of measures assigned with a factor to a degree of fulfillment between 0 and 1. Figure 1 depicts the Quamoco meta quality model.

Quality model elements can be grouped into *modules* to facilitate reuse and maintenance of quality models. As an example, source code quality models can be split into modules according to programming paradigms or languages, so that general source-code related concepts can be reused within an object orientation module, which is then further operationalized by modules containing measures and instruments for C# or Java. This way, technology-independent information can be reused, while technology-dependent modules add automation by linking general measures to analysis tools. In addition, this modularization concept can be used to extend or adapt quality models. Project-specific quality requirements can be added as an individual module, and evaluation formula can be overridden in order to adapt priorities according to the project goals.

More details on the modeling concepts, the elements that constitute the quality model, as well as the relations between them, can be found in [15]. This paper proposes an extension to this meta model to specifically address software architecture quality so that architectural styles can be compared based on their impact on software quality and existing software architectures can be evaluated with respect to both architecture conformance and quality.

IV. Architecture Model Extension

To specifically address software architecture quality in the context of a given architectural style, I propose an extension to the meta model presented in the previous section. In order to retain compatibility with the existing tools for editing, maintaining, visualizing and executing quality models, this extension is based on conventions, so that e. g. instead of formally adding a new model element type, I propose adding certain semantics to existing element types. Technically, already the Quamoco meta quality model does so by using the *factor* concept for both quality aspects and product factors.

A. Modules

Quality goals like [Adaptation | EFFICIENCY] are usually independent from architectural styles. In particular this is the case for quality standards that do not make any assumptions on the architecture of the software to assess (e. g. ISO 25010). Therefore, quality goals should be usable across different quality models and are hence defined within an independent *Base* module.

According to the overall question, which quality goals are directly influenced by the underlying principles of a certain architectural style, all quality model elements directly related to these principles are subsumed in a module named *Conformance*. Other quality-related concepts beyond these principles constitute the *Design* module.

Within the *conformance* module, the main elements are the *Architecture Principles*, which are modeled as a special case of product factors. Their degree of fulfillment states how well the system under evaluation implements the respective principle. Since these principles are often defined on an abstract level, they are refined by product factors describing directly observable properties of system entities, which in turn are quantified by measures. The details of the conformance module are described in Section IV-C.

Further design aspects contributing to software quality are subsumed in the *design* module, which contains product factors that cannot directly be deferred from principles (e. g. parameter granularity of operations). The design module is described in Section IV-D. The modular structure of the architecture quality model can be found in Figure 2. The *uses* relation between the *design* and the *conformance* module indicates that basic measures defined in the latter could be referenced from the former in order to avoid a duplication of modeling effort.

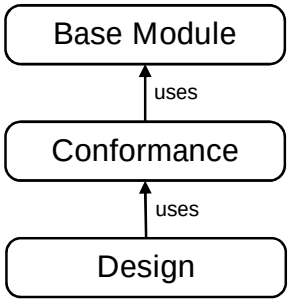

Figure 2. Modular Structure

In addition to the modules described here, the modularization approach can be utilized to extend the quality model with domain or project-specific modules. Moreover, it is also possible to include modules for quality assessment on a different level of detail, e. g. source code, in order to aggregate all assessment results into a single hierarchy of quality goals.

B. Quality Goals

Classical quality attribute hierarchies have been criticized, because their decomposition appears to reflect different, implicit criteria and lacks clear semantics [17]. To tackle this concern, activity-based quality models (ABQM) have been proposed in order to provide more natural and meaningful decomposition semantics [18]. In my approach, I support this view and propose to describe quality goals using activities and attributes.

A set of relevant activities can be obtained from both existing ABQMs and literature on the software life-cycle, e. g. IEEE 1074 [19]. Definitions of traditional quality attributes like ISO 25010's quality characteristics often reference activities and even corresponding attributes. The relation between these quality attributes and activity-based quality goals has been discussed in more detail by Lochmann and Goeb [20]. There, ISO 25010's quality characteristics ("-ilities") are explicitly part of the quality model, represented as high-level product factors that have an influence on the activity-based quality goals, e. g. the property of a software product's UI not to use color as the only means of encoding important information has an influence on the factor [Product | ACCESSIBILITY], which in turn positively impacts the quality goal [Perceive | EFFECTIVENESS][2].

Quality goals can be refined using general techniques from the requirement engineering field (e. g. [21]–[23]). In the quality model, this refinement can either be done along the attributes (e. g. [Use | QUALITY] is refined to [Use | EFFICIENCY],

[2]Although the syntax has been adapted to be consistent with the model presented here, the semantics of the original paper have been preserved.

[Use | EFFECTIVENESS], and [Use | CONTINUITY]), or along the activities (e. g. [Use | EFFICIENCY] is refined to [Perceive | EFFICIENCY], [Understand | EFFICIENCY], and [Act | EFFICIENCY]). For *maintainability*, e. g., Deissenboeck et al. [18] provide a thorough decomposition of maintenance activities.

In the following, I use SOA as an example for an architectural style and derive some activity-based quality goals from typical scenarios. In order for a potential service consumer to decide whether a service offers the desired functionality and is therefore feasible for a given usage scenario, he first has to understand it. The respective quality goal is [Analysis | EFFICIENCY], since this analysis should be as efficient as possible. Similarly, other activities imply the goal of being conducted efficiently, in particular [Composition | EFFICIENCY], [Adaptation | EFFICIENCY], and [Test | EFFICIENCY], which are self-explanatory. The degree to which a service satisfies consumers' needs in terms of functionality and therefore enables effective service consumption can be expressed as [Consumption | EFFECTIVENESS]. Interaction between services is crucial for an SOA to be effective. Since service interoperation is achieved by composing services, this quality goal of effective interaction between services can be represented as [Composition | EFFECTIVENESS].

C. Conformance—Architectural Principles

The *conformance* module of an architecture quality model contains all essential principles that constitute a particular architectural style. As shown in Figure 3, the conformance module consists of two kinds of factors, namely *principle factors* and *conformance factors*. The former provide a general definition and explanation of an architectural principle and its impacts on quality goals, which can be either positive or negative. The latter refine these principle factors into properties that can directly be observed from the system artifacts. In order to quantify the degree of such a property's presence or absence in a software architecture, each conformance factor is assigned with one or several measures. An evaluation function assigned to each factor puts these measurement results into relation and maps them on a scale representing the factors' degree of fulfillment.

An example from the SOA domain would be the principle of [SERVICE COMPOSITION], expressing that services can be composed in order to create new services. This principle is refined into conformance factors: A factor [Service | OUTGOING DEPENDENCIES] could describe the fact that services that depend on other services do make use of composition and hence support the composition principle. A second conformance factor, [Service | CONSUMPTION RATIO], could describe to which degree services wihtin the system consume other services. [SERVICE COMPOSITION] itself has a positive impact on [Consumption | EFFECTIVENESS], because a system that makes use of service composition allows for fine-grained reuse of services and therefore facilitates effective service consumption. To quantify [Service | CONSUMPTION RATIO], e. g. the measure *Consumer Provider Ratio* is defined, which describes the ratio between provider and consumer services within the system. Provider services are services that are consumed by other services within the system, whereas consumer services consume other services. Of course, services can be both providers and consumers at the same time. This way, the rather intangible architectural principle of composition can be refined with the help of conformance factors into observable properties that are quantified by a set of defined measures. At the same time, the effects of adhering to this principle are captured in terms of impacts on quality goals.

D. Design—Architectural Best-Practices

Adhering to a certain architectural style is not sufficient to ensure good quality. Usually, architectural principles are accompanied by guidelines and best-practices. The *design* module contains factors and measures describing these additional properties that are not covered by the basic principles of an architectural style. While the general Quamoco approach does not restrict the type of product factors contained in a model, this module explicitly separates architectural best-practices from other kinds of factors in order to provide a more concise view on the overall product quality.

In contrast to conformance factors, design factors directly define impacts on quality goals. They can, however, be organized hierarchically in order to group similar low-level properties and make navigating the model easier. Typical topics to be covered in the design module are *dependencies*, *documentation and naming*, *granularity*, or *size and complexity*. Each of these topics can be addressed by several product factors, describing respective architectural properties. Concerning granularity, e. g., one of these factors could be [Operation | FUNCTIONAL GRANULARITY], which expresses the property that a service operation should perform exactly one function, e. g. searching a customer database for entries matching a provided search pattern. This factor is quantified using the measure *Operations performing multiple functions*, which provides guidance for system experts to assess service operations and report those, which perform more than one function.

These factors and measures typically resemble a collection of design guidelines and best-practices that are known to have an influence on certain quality goals. In order to obtain a comprehensive set of factors and measures, a thorough analysis is required, followed by a validation in order to be sure that all typical aspects of the particular architectural style are appropriately covered by the model. A validation method for quality models has been proposed and applied in the context of a quality model for embedded systems by Mayr et al. [24]. Architectural design factors and measures for SOA have been published by Goeb and Lochmann [25].

E. Instantiation and Usage

The meta quality model defined above can be instantiated to build a quality model for an architectural style by combining various sources of knowledge like personal experience or documented research studies. Usually these sources vary depending on the type of model elements. In order to create the set of principles in the conformance module, literature on that particular architectural style is probably most appropriate. The refinement into factors can be performed based on personal experience as well as existing models or frameworks. Likewise, there is a large amount of well-evaluated research studies on the impacts of particular design properties on different aspects of software quality. The advantage of a formally defined model compared to these textual representations is that the consolidated model can be visualized, navigated and analyzed more easily using appropriate tools. In addition, contradictions or missing information become more evident in a formal model. Literature might not provide a consistent view on how different measures should influence a factor's degree of fulfillment.

As part of a larger research effort I created a corresponding quality model for SOA, containing SOA principles as well as further design factors. This model has been created over the course of the recent years and will be published separately, including an expert-based evaluation of its overall structure as well as its contents. In total, the SOA quality model consists of 111 elements and therefore cannot be presented here in detail. An overview is, however, depicted in Figure 4 on the next page.

For the weighing of model elements against each other in order to allow quality assessment using the model, I propose an iterative approach: First, initial evaluation functions should be manually defined for each factor based on personal experience. Once a quality model is completely defined and operationalized, a benchmarking technique should be employed to calibrate these functions. This is achieved by assessing a certain amount of software systems using the quality model and thus observing typical value ranges in real-world systems. More information on how to use benchmarking approaches for the calibration of software quality models can be found in [26].

In order to perform a model-based architecture quality analysis, all measures defined in the model have to be provided with measurement values. Using the Quamoco tool chain, this can either be achieved by implementing according adapters for automatically obtainable values or by generating a spreadsheet template from the model, which can be filled with the respective values by an inspector. In a second phase, these measurement values are aggregated along the hierarchy of refinement and impact relations defined in the model, evaluating the formulas provided for each model element. Because this is a core functionality of the Quamoco approach, it is not elaborated here in more detail. An exemplary quality assessment can be found in Section V.

F. Summary

The proposed meta quality model for software architecture evaluation is comprised of three modules. The *base* module contains definitions of quality goals and relations between them. Usually, these quality goals are structured hierarchically. To represent quality goals, I propose the activity-based notion, so that each quality goal is expressed as a pair of an activity and an attribute. Hierarchical refinement of quality goals can be done along both activities and attributes.

The *conformance* module contains information regarding the core principles of a certain architectural style. These principles are represented as *principle factors*. In order to provide means for architecture conformance assessment, these principles are refined by *conformance factors*, which resemble observable properties of the software architecture to reflect these principles. These factors are quantified by *measures*, which can either be obtained by measurement tools and metrics, or manually during architecture inspections. Besides architecture conformance assessment, the conformance module can provide valuable insight regarding the effect of an architectural style on software product quality. In order to achieve this, principle factors describe *impacts* on the quality goals defined in the base module.

The *design* module covers quality-relevant architectural properties originating from guidelines and best-practices, which are not directly related to the principles of an architectural style. Usually, conformance to architectural principles helps achieving high quality, but is not sufficient. An analysis of the quality model can easily identify quality goals that are not sufficiently covered by architectural principles and hence lead to the definition of further design guidelines in order to achieve this coverage. These guidelines are represented by *design factors*, which again resemble observable properties of a software architecture. Design factors also define *impacts* on quality goals, so that architecture quality assessment can be done on the combination of conformance and design factors. The resulting meta quality model for software architecture conformance and quality assessment is depicted in Figure 3.

In summary, the proposed structure allows the separation of quality-related effects of an architectural style from general best-practices that improve software quality. In addition, architectural principles become tangible by refining them to observable properties of architecture artifacts. In early project phases, this transparency ensures a common understanding throughout the project. Once according models are available for different architectural styles, they can also serve as a decision basis in order to decide for a particular one.

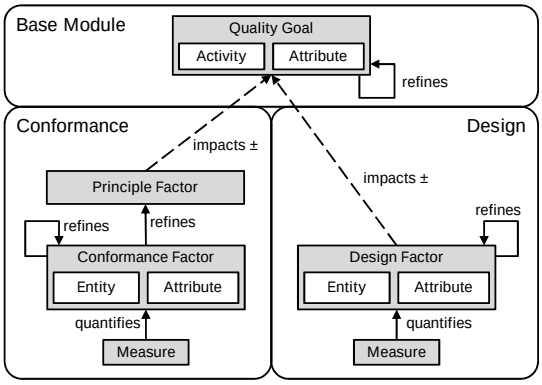

Figure 3. Software Architecture Meta Quality Model

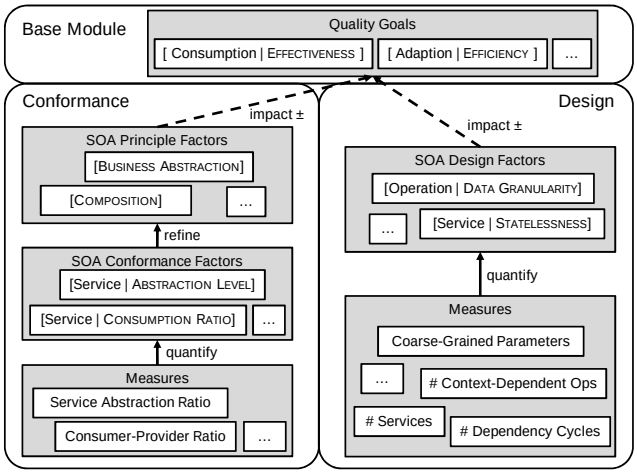

Figure 4. SOA-Model

V. Validation

In order to validate that an architecture quality model built in accordance with the proposed meta model can be used for architecture quality assessment in practice, I designed the following experiment: We used the architecture conformance and quality model for SOA described in Section IV-E to assess an actual SOA-based software system, which was developed as part of the European SmartProducts research project[3]. This project was selected for the experiment because the number of services was small enough to manually trace the evaluation results from the values on the *quality goal* level down to values for individual measures. This was considered an essential requirement for a first applicability test. Figure 4 shows the SOA conformance and quality model's structure as well as *some* of the contained quality goals, factors and measures.

A. Goal

We conducted this experiment to validate the practical applicability of the presented approach for software architecture conformance and quality assessment. Practical applicability is measured by the success of conducting an evaluation based on the set of measures defined in the model. Furthermore, the experiment should demonstrate the compatibility of the underlying concepts with existing tools for quality modeling and assessment.

B. Setup and Procedure

Due to the diversity of analyzed artifacts (specifications, models, source code), tool-based measurement would have required implementing respective measurement tools for each of these artifact types and has therefore been discarded. Instead, I adapted the quality model, converting all measures to be manually assessed. The Quamoco tool chain [27] requires manual measures to be numerical. Following this rule, it was possible to export an Excel template containing all measurement instruments along with their descriptions and a column to enter the manually obtained measurement results. Five researchers familiar with the system's architecture used this template and manually conducted all the measurements contained therein. All of the experts hold a university degree in computer science, two of them also hold a Ph. D. In addition, they all have several years of experience in developing software. Because the only tool the experts used was the Excel template, they did not have to be introduced to the whole quality model in advance, but could focus on collecting measurement data without being biased by knowing how the evaluation would be impacted by either of the measurement values.

Table I shows the measure names as well as the assessed measurement values. The values represent the mean values of the results obtained by the five experts. The filled template has been imported into the assessment tool chain as the basis for further processing according to the SOA model.

Due to the capabilities of the Quamoco tooling in handling manually obtained measurement results, the remaining steps of the quality assessment did not involve any manual effort. The measurement data was processed and transformed into an HTML dashboard presenting scores for each quality goal contained in the quality model based on the evaluation formulas defined there. These results were then presented to the project team, followed by a discussion on their opinion regarding the approach of model-based, semi-automatic architecture quality assessment in general, and the applicability of the SOA quality model in particular.

C. Results

In general, the project team highly appreciated the effort savings due to the automation provided by the quality model and the associated tools. Compared to other architecture assessment techniques that had been used in the project

[3]http://www.smartproducts-project.eu/

Table I
RAW MEASUREMENT RESULTS

Measure Name	Value
Contract Implementation Coupling	0.00
Inconsistent Interface Standards	0.50
Number of Utility Services	5.00
Average Service Fan-In	2.00
Average Service Fan-Out	2.00
Number of Domain Services	3.00
Number of Process Services	4.00
Total Number of Services	12.00
Consistent and Unambiguous Documentation	50.00
Number of Dependency Cycles	0.00
Average Dependency Depth	2.00
Efficiently Groupable Parameters	6.00
Inconcisely Named Operations	28.00
Inconcisely Named Parameters	35.00
Inconcisely Named Services	5.00
Number of Context-Dependent Operations	5.00
Number of Exposed Operations	0.00
Total Number of Operations	55.00
Total Number of Parameters	100.00
Operations with External Dependencies	10.00
Operations Performing Multiple Functions	10.00
Average Dependencies per Service	2.00
Semantically Equivalent Services	0.00
Interface Data Cohesion	0.40
Interface Sequential Cohesion	0.15
Interface Usage Cohesion	0.50
Functional Domain Coverage	11.00

Quality Assessment (Main)
Quality Assessment

Element	Evaluation Result
Property @Product []	
Quality @UseCase []	0,613
Quality @Adaptation [refined]	0,652
Quality @Operation [refined]	0,512
Quality @Ext. Analysis [refined]	0,687
Quality @Int. Analysis [refined]	0,499
Efficiency @Int. Analysis [refined]	0,499
Cohesion @ServiceInterface [impacted]	0,417
Complexity @ServiceInterface [impacted]	0,841
Design Size @SOA System [impacted]	0,240
Interface Complexity @SOA System [impacted]	0,000
Meaningful Names @ServiceInterface [impacted]	0,573
Proper Documentation @ServiceInterface [impacted]	1,000
Quality @Composition [refined]	0,705
Quality @Consumption [refined]	0,498
Effectiveness @Consumption [refined]	0,490
Efficiency @Consumption [refined]	0,523
Low Fragmentation @SOA System [impacted]	0,091
Short Dependency Paths @Service [impacted]	0,956
DDT	threshold=[0;3] value=0,944
DDT [measures]	2,000
SDT	threshold=[0;5] value=0,967
Continuity @Consumption [refined]	0,479
Quality @Test [refined]	0,736

Figure 5. Validation Output View

before (e. g. *Active Design Reviews* [10]), the team reported that especially preparation and data analysis were far less time-consuming, so that the team was able to assess the system within one working day instead of several days for preparing review questionnaires and analyzing the results. At the same time the experts reported that the actual measurement required a profound knowledge of the system and could not easily be conducted by external people. This appears reasonable, taking into account that many of the measures were initially meant to be conducted automatically using appropriate tools, but had been adapted to be conducted manually due to the different development stages of the involved artifacts.

Obviously the time savings for the analysis of a single project's architecture do not compensate for the initial effort of building the SOA model. However, since the model is available and reusable, other projects can make use of it and (optionally) invest some time in project-specific tailoring.

The analysis tool provides the user with different visualizations (see e. g. [15]) and a hierarchical HTML-based table. Figure 5 shows an excerpt from this table. The first column contains the hierarchical decomposition of quality goals, including the impacting factors and the respective measures. The right column shows the analysis results. In the case of measures, these results correspond to the numbers read from the template file. For factors and quality goals, the value range is the interval $[0; 1]$, which represents the respective degree of fulfillment.

Concerning the interpretation of these values it is important to mention that they are not meant to be absolute statements. Experience with model-based quality assessment has shown that quality models only produce objective results if they are properly calibrated using a large number of reference products. The experiment presented here only aimed at showing the general applicability of the approach for automatic quality assessment and the compatibility of the adapted meta quality model with the Quamoco tool chain. The calibration procedure for the Quamoco base quality model for source code is outlined in [15] and can be applied for architecture conformance and quality models as well.

Despite this fact, the results for some of the factors can already provide hints regarding architecture quality. One interesting finding was that the system scored comparably low on the [BUSINESS ABSTRACTION] SOA principle. Discussions revealed that the particular system was indeed not completely following the SOA principle of providing mainly business process-relevant functionality as services, but also offered "low-level APIs". Likewise, other assessment results on a general level matched the overall perception of the involved experts. Hence, model-based architecture conformance and quality analysis can help software architects to increase transparency regarding the conformance to architectural principles as well as potential quality problems with comparably low effort, even before the software product is completely implemented.

Because the main goal of this experiment was the applicability evaluation of our approach for quality assessment,

a more detailed interpretation of assessment results was not performed. In addition, such an analysis would have required a proper calibration of the model's evaluation formulas in order to provide sound results. A more detailed discussion on this topic can be found in the following section.

VI. Conclusions and Outlook

In this paper, I presented a variant of the Quamoco quality modeling approach specifically addressing software architecture conformance and quality. While relying on Quamoco's meta quality model, several conventions have been applied in order to describe both the inherent properties of an architectural style and additional properties the architecture should possess in order to achieve high software quality. These two flavors of architecture quality are represented in the model's *conformance* and *design* modules, respectively. The model relies on the activity-based quality modeling approach, which means that quality goals are expressed via activities conducted with the system and attributes of these activities. This way the multifaceted concept of quality is structured along intuitive decomposition semantics by splitting activities into sub-activities.

The building blocks of a software architecture are represented by entities, which are characterized by attributes in order to describe factors that can be observed in a software architecture. These factors have impacts on quality goals, making them the intermediary between general quality goals and actual measurements and metrics.

An experiment applying an architecture quality model for SOA to an actual software system showed that architecture quality models built according to our proposed structure can be used for architecture evaluation. The project team who built the system used for evaluation stated that the greatest benefit of our approach is the strong reduction of manual analysis and processing effort compared to other architecture evaluation approaches applied in the project before. Given a limited overall time frame, this allows for architecture assessment more often, leading to increased transparency regarding architectural properties and hence to more responsive quality control. In addition, the most significant assessment results matched the perception of the project team. In order to show the value of the SOA quality model beyond these tendencies, further empirical studies are needed.

The architecture meta quality model provides a framework to consistently collect and conserve quality knowledge. Architecture quality models can be used to investigate for a given quality goal, how it is affected by the principles of an architectural style, and which additional design patterns should be implemented in order to reach this quality goal. This helps software architects to argue to which degree a decision for a certain architectural style already has positive impacts on software quality, and whether these might be affected by ignoring further design properties.

Moreover, architecture quality models build the basis for standardized and reproducible architecture quality assessment by formalizing relationships between measures, factors and quality goals. Back in 1992, Grady described his vision regarding the importance of metrics in software engineering in the year 2000 as follows: "First, tools will automatically measure size and complexity for all the work products that engineers develop. Besides warnings and error messages, the tools will predict *potential* problem areas based on metric data thresholds" [28, p. 220]. For source code, this goal might have actually been reached, taking the large amount of dashboards and related tools into account. With respect to software architecture, however, even twelve years after the mentioned date, this vision has not become reality yet. The proposed architecture conformance and quality model might be a step into this direction, since it provides clear relationships between important concepts and allows for automated assessment. Hence, it contributes to the reproducibility of quality analysis, making sure that incremental assessments after changes in the software are based on the same evaluation rules. This also lowers the required effort, since aggregation, processing and visualization of results are automatically conducted using quality analysis tools like ConQAT. Further automation potential is given by the possibility to also use tools to obtain the measurement values.

Further areas of future work include applying the modeling approach to a number of architectural styles. Doing so, the resulting quality models can help to compare architectural styles based on their impact on quality and provide decision support for software architects. For a particular software development project, they are provided with a means to choose the architectural style that most adequately covers the quality requirements defined for that project. In the long term, a framework could emerge to describe architectural styles from a quality perspective, providing a solid basis for decisions in early software development phases.

Acknowledgment

This work has partially been supported by the German Federal Ministry of Education and Research (BMBF) in the project Quamoco (01 IS 08023D). Further thanks go to the SmartProducts project team at TU Darmstadt for taking part in the applicability experiment.

References

[1] A. Rivers and M. Vouk, "Resource-constrained non-operational testing of software," in *Proc. 9th Software Int. Symposium on Reliability Engineering*, 1998, pp. 154–163.

[2] L. Briand, J. Wüst, J. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[3] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2001.

[4] M. Svahnberg and C. Wohlin, "An investigation of a method for identifying a software architecture candidate with respect to quality attributes," *Empirical Software Engineering*, vol. 10, no. 2, pp. 149–181, 2005.

[5] S. Wagner, K. Lochmann, S. Winter, F. Deissenboeck, E. Juergens, M. Herrmannsdoerfer, L. Heinemann, M. Klaes, A. Trendowicz, J. Heidrich, R. Ploesch, A. Goeb, C. Koerner, K. Schoder, and C. Schubert, "The Quamoco Quality Meta-Model," Technische Universitaet München, Tech. Rep. TUM-I128, 2012.

[6] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit, *Characteristics of Software Quality*. North-Holland, 1978.

[7] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality," Rome Air Development Center, Tech. Rep. RADC-TR-77-369, 1977.

[8] S. Wagner, K. Lochmann, S. Winter, A. Goeb, M. Klaes, and S. Nunnenmacher, "Software Quality Models in Practice," Technische Universitaet München, Tech. Rep. TUM-I129, 2012.

[9] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The squale model - A practice-based industrial quality model," in *Proc. IEEE Int. Conf. on Software Maintenance (ICSM)*, 2009, pp. 531–534.

[10] D. L. Parnas and D. M. Weiss, "Active design reviews: principles and practices," in *Proc. 8th Int. Conf. on Software Engineering (ICSE)*, 1985, pp. 132–136.

[11] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, *Software-Architektur: Grundlagen - Konzepte - Praxis*, 2nd ed. Spektrum Akademischer Verlag Heidelberg, 2009.

[12] J. Zhao, "Bibliography of Software Architecture Analysis," *Software Engineering Notes*, vol. 24, no. 4, pp. 61–62, 1999.

[13] F. Losavio, L. Chirinos, A. Matteo, N. Levy, and A. Ramdane-Cherif, "ISO quality standards for measuring architectures," *Journal of systems and software*, vol. 72, no. 2, pp. 209–223, 2004.

[14] F. Losavio, L. Chirinos, N. Lévy, and A. Ramdane-Cherif, "Quality Characteristics for Software Architecture." *The Journal of Object Technology*, vol. 2, no. 2, pp. 133–150, 2003.

[15] S. Wagner, K. Lochmann, L. Heinemann, M. Klaes, A. Seidl, A. Goeb, J. Streit, A. Trendowicz, and R. Ploesch, "The Quamoco Product Quality Modelling and Assessment Approach," in *Proc. 34th Int. Conf. on Software Engineering (ICSE)*, 2012.

[16] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying Maintainability," in *Proc. 4th Workshop on Software Quality (WoSQ)*, 2006, pp. 21–26.

[17] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring Software Product Quality: A Survey of ISO/IEC 9126," *IEEE Software*, vol. 21, no. 5, pp. 88–92, 2004.

[18] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard, "An Activity-Based Quality Model for Maintainability," in *Proc. IEEE Int. Conf. on Software Maintenance (ICSM)*, 2007, pp. 184–193.

[19] IEEE, "Std 1074-2006 – IEEE Standard for Developing a Software Project Life Cycle Process," 2006.

[20] K. Lochmann and A. Goeb, "A Unifying Model for Software Quality," in *Proc. 8th Int. Workshop on Software Quality (WoSQ)*, 2011, pp. 3–10.

[21] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 978–1005, 2000.

[22] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," in *Proc. Int. Symposium on Requirements Engineering*, 2001.

[23] S. Doeweling, B. Schmidt, and A. Goeb, "A model for the design of interactive systems based on activity theory," in *Proc. ACM Conf. on Computer Supported Cooperative Work (CSCW)*, 2012, pp. 539–548.

[24] A. Mayr, R. Plösch, M. Kläs, C. Lampasona, and M. Saft, "A Comprehensive Code-based Quality Model for Embedded Systems," in *Proc. 23rd Int. Symposium on Software Reliability Engineering (ISSRE)*, 2012.

[25] A. Goeb and K. Lochmann, "A software quality model for SOA," in *Proc. 8th Int. Workshop on Software Quality (WoSQ)*, 2011, pp. 18–25.

[26] K. Lochmann, "A Benchmarking-inspired Approach to Determine Threshold Values for Metrics," in *Proc. 9th Int. Workshop on Software Quality (WoSQ)*, 2012.

[27] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner, "The quamoco tool chain for quality modeling and assessment," in *Proc. 33rd Int. Conf. on Software engineering (ICSE)*, 2011, pp. 1007–1009.

[28] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.