Software Metrics for Measuring the Understandability of Architectural Structures – A Systematic Mapping Study

Srdjan Stevanetic and Uwe Zdun
Software Architecture Research Group
University of Vienna, Austria
srdjan.stevanetic|uwe.zdun@univie.ac.at

ABSTRACT

The main idea of software architecture is to concentrate on the "big picture" of a software system. In the context of object-oriented software systems higher-level architectural structures or views above the level of classes are frequently used to capture the "big picture" of the system. One of the critical aspects of these higher-level views is understandability, as one of their main purposes is to enable designers to abstract away fine-grained details. In this article we present a systematic mapping study on software metrics related to the understandability concepts of such higher-level software structures with regard to their relations to the system implementation. In our systematic mapping study, we started from 3951 studies obtained using an electronic search in the four digital libraries from ACM, IEEE, Scopus, and Springer. After applying our inclusion/exclusion criteria as well as the snowballing technique we selected 268 studies for in-depth study. From those, we selected 25 studies that contain relevant metrics. We classify the identified studies and metrics with regard to the measured artefacts, attributes, quality characteristics, and representation model used for the metrics definitions. Additionally, we present the assessment of the maturity level of the identified studies. Overall, there is a lack of maturity in the studies. We discuss possible techniques how to mitigate the identified problems. From the academic point of view we believe that our study is a good starting point for future studies aiming at improving the existing works. From a practitioner's point of view, the results of our study can be used as a catalogue and an indication of the maturity of the existing research results.

1. INTRODUCTION

The main idea of software architecture is to concentrate on the "big picture" of a software system and to enable architects to abstract away the fine-grained details of the implementation and other development artefacts [48]. The software architecture of the system is defined as: "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [12]. It is generally recognized that the software architecture plays a key role in managing and maintaining the overall software system.

In the context of object-oriented software systems higher-level architectural structures or views above the level of classes (such as package-, module-, component-views) are frequently used to capture the "big picture" (architectural information) of the system [39]. Understanding those higher-level architectural structures and their relations to the lower-level class view of the system that captures the implementation details, can help in bridging the gap that exists in the abstraction level between higher-level and lower-level views of the system. In that way it can improve so-called "shared understanding" of the system referred by some authors as "architecture" [23]. Among the given higher-level architectural views, component and connector views (or component views for short) are frequently used as a central part of the architectural descriptions of software systems [19]. Please note that, according to the software architecture community, an architectural description can comprise multiple views that describe the system concentrating on one of many system concerns, such as logical, implementation, deployment, or process views, and from the viewpoint of different stakeholders, such as end-users, developers, project managers, and business analysts [41, 19]. In this study we concentrate on the architectural information captured in the higher-level structures with regard to their relations to the system implementation (i.e., the implementation view). From the perspective of the system implementation, component views deal with the coarse-grained components that represent major system functionalities, units of runtime computation or data-storage, and the connectors that are the interaction mechanisms between them [49]. Since a component in an architectural component view represents a high-level abstraction of the entities in the source code of the software system, it can be broken down into (i.e., is refined by) more fine-grained, technical components or classes that realize the component in the technical design or implementation of the system.

Understandability is one of the most important characteristics of software quality because the difficulty of understanding the software system limits its reuse and maintenance and therefore can influence cost or reliability of software evolution. In the context of component views (and other mentioned higher-level architectural views), understandability is a critical aspect, as one of the main purposes of software architecture is to " ... enable designers to ab-

EASE '15, April 27 - 29, 2015, Nanjing, China
Copyright 2015 ACM 978-1-4503-3350-4/15/04 ...$15.00.
http://dx.doi.org/10.1145/2745802.2745822.

stract away fine-grained details that obscure understanding and focus on the "big picture:" system structure, the interactions between components, ..." [48]. This, however, is not possible if the given views themselves and/or the links to other design and code artefacts are hard to understand. Understandability concepts that we consider in this study come from the general properties which should be considered for assessing understandability and other software qualities in the system [14, 9, 24]. They include size, complexity, coupling, cohesion, and modularization. Since our study aims at investigating the higher-level architectural structures with regard to their relations to the system implementation, the properties related to understandability are considered from the structural and relational details of a system's design and implementation.

To predict the understandability or some other quality characteristics of the architecture, architectural evaluation analysis is performed. It aims to identify potential risks and verify that the quality requirements have been addressed in the design [43]. Two basic classes of architectural evaluation techniques, questioning and measuring, have been proposed [3, 12]. Questionnaire techniques use qualitative questions to be asked about the architecture, and they can be used to assess any given software quality. It includes scenarios, questionnaires and check lists. Measuring techniques propose quantitative measurements that are used to answer specific questions related to specific software qualities. In that sense they are not as broadly applicable as questioning techniques. Measuring techniques encompass metrics, prototypes, experiences and simulations. Existing literature reviews on architectural evaluation methods [20, 8, 10, 33] mainly focus on scenario-based methods. A scenario describes interaction of a stakeholder with a system. Scenario-based methods provide techniques for evaluating and documenting architecture-related scenarios against the requirements.

In this article we report on a systematic mapping study on software metrics that measure the understandability concepts of the higher-level architectural structures (i.e., metrics that operate at the level above classes) with regard to their relations to the system implementation. Those metrics can add valuable information to a sustainability and maintainability evaluation of those architectural structures [40]. In that context we present the definition of software metrics. According to the IEEE standard glossary of Software Engineering Terms (adapted from [51]), software metrics are "a quantitative measure of the degree to which a system, component, or process possesses a given attribute". Our metrics do not consider the lower-level metrics such as object-oriented class level metrics or metrics based on lower-level artefacts like methods, attributes, variables, etc., because those are not architecture-level metrics. The lowest level artefact that we consider is a class. Please also note that we focus at the lowest level on object oriented software structures.

This study is organized as follows: Section 2 discusses related reviews, Section 3 describes the methodology, Section 4 provides the results of our research questions and the discussions related to them, Section 5 discusses the limitations of the study, and Section 6 concludes the study and discusses possible future trends.

2. RELATED REVIEWS

Regarding related literature reviews, so far, to the best of our knowledge, only one systematic review exists that considers architecture-level metrics as a part of the study (the work provided by Koziolek [40]). Namely, the author provided a systematic review on scenario-based methods for supporting the sustainability of a software architecture both during early design and during evolution. Additionally, he surveyed around 40 architecture-level metrics and emphasized the importance of integration of these two fields. However, comparing to our study the author considered software architecture in a broader sense (i.e., assuming that in a general sense it encompasses multiple views functional, development, concurrent, and can be related not only to the system implementation but also to the system design decisions, architecture requirements, concerns, etc.) and thus provided a more holistic or coarse-grained approach that included just more recent studies. Our study is more fine-grained because we concretely study the architecture-level metrics from the system implementation perspective with regard to the understandability concepts that are important in that context. In our study we identified 25 approaches comparing to 11 approaches identified in the work by Koziolek. Beside the metrics that overlap with the metrics that we identified the author mentioned some metrics related to the design decisions, architectural requirements and architectural concerns or aspects [59, 56] and some metrics that consider the dependencies between lower level artefacts (source code, class methods, functions) [15, 55] that are out of scope of this study. Furthermore, in the work by Koziolek the studies that consider metrics are just shortly explained (within a small paragraph), while our study provides much more information, i.e., the maturity assessment of the studies, the mapping of the studies to the measured artefacts, attributes, quality characteristics, approach type, and the limitations of the studies.

Regarding other related reviews, different reviews exist in the literature related to lower level class-level metrics [18, 53]. For example, Riaz et al. [53] provided a systematic review of class-level OO metrics and discussed their usefulness for maintenance prediction. As already mentioned above literature reviews on architectural evaluation methods mainly focus on scenario-based methods [20, 8, 10, 33].

Software architecture reconstruction is another research direction closely related to our work. It helps in dealing with two problems, the first is that architecture is not explicitly represented in the source code and the second is that successful software applications evolve over time, so their architecture inevitably drifts. Reconstructing the architecture and checking whether it is still valid is therefore an important issue. Ducasse et al. provided a survey on methods and tools for software architecture reconstruction [21].

3. RESEARCH METHOD

Our study is based on the guidelines for performing a systematic review proposed by Kitchenham et al. [34, 36, 37]. A systematic review is a key instrument to pursue a systematic, exhaustive search in the relevant field of study in order to address one or more concrete research questions. It uses a well-defined methodology that reduces bias and introduces a wider context that allows for a more general conclusion [50]. As the first step in our study we defined the research questions. Then we pursued the search through the existing literature to collect as much relevant studies and information as possible. After that we focussed on developing a procedure of how to extract and summarize the relevant information that is needed to address our research questions. It included several iterations, consultations and improvements until the final results are obtained.

3.1 Research Questions

Our main research questions are defined as follows:
RQ1: Which software metrics that measure the understandability concepts of higher-level architectural software structures (i.e. metrics that operate at the level above classes) exist?

Within this research question we consider (beside the list of suitable metrics): what is the granularity level of the metrics (i.e., do they measure the whole structure or a single artefact), which representation model for the metrics definitions is used, and how are the metrics distributed in terms of the measured artefacts and software attributes (i.e. size, coupling, cohesion, etc.).
RQ2: What is the maturity level of the identified metrics?

Within this research question we consider different parameters that characterise the metrics' maturity level and their practical applicability: metrics definitions, level of validation, mapping to the measured quality characteristics, comparative analysis, usability, applicability, and tool support (see Section 3.4 for more details).
RQ3: Which limitations in current studies should be addressed in future research?

The purpose of this research question is to identify any shortcomings in the existing studies and to propose possible techniques and future work how to mitigate them.

3.2 Search Process

The search process was started by searching through four digital libraries (DLs): the ACM (Association for Computing Machinery) digital library[1], the IEEE digital library[2], the Scopus digital library[3], and the Springer digital library[4]. We pursued an advanced search within these libraries which include bibliographic references, abstracts, and links to electronic editions of the studies from all major publishers in computer science (i.e., not only the four publishers ACM, IEEE, Elsevier (the owner of Scopus) and Springer themselves). According to the defined research questions we defined the set of words (search string) we have used with the search engines of these four digital libraries to find an exhaustive list of relevant studies. The search string we have used is the following:

1. ((‘metric(s)’)
2. **and** (((‘software’ or ‘object-oriented’ or ‘architectural’ or ‘component-based’) and (‘system(s)’ or ‘design(s)’ or ‘model(s)’ or ‘architecture(s)’)) or ‘class(es)’ or ‘component(s)’ or ‘package(s)’ or ‘module(s)’)
3. **and** (‘understandability’ or ‘comprehensibility’ or ‘comprehension’ or ‘understanding’ or ‘understand’ or ‘comprehending’ or ‘comprehend’ or ‘complexity’ or ‘coupling’ or ‘cohesion’ or ‘modularization’ or ‘size’)
4. **not** (‘hardware’ or ‘memory’ or ‘machine’ or ‘chip(s)’ or ‘microprocessor(s)’))

[1]see: `http://dl.acm.org/`
[2]see: `http://ieeexplore.ieee.org/Xplore/home.jsp`
[3]see: `www.scopus.com`
[4]see: `http://link.springer.com`

The whole search string is composed of four parts linked by the coordinating conjunctions “and” or “not”. The first part contains “metric(s)”, meaning that we only include studies which contain the word metric or its plural form in the abstract. The second part delimits the context of the metrics: It should be related to higher-level software structures (architecture, design, model, system) and artefacts (packages, modules, components) or at least object-oriented structures (classes). The third part is related to the mentioned understandability concepts. The set of used words in this part considers the general properties which should be considered for assessing understandability (see Section 1) together with the word “understandability” and its synonyms. Finally the fourth part of the search string excludes studies related to hardware architecture and similar concepts.

When applying this proposed search string we obtained 3951 possibly relevant studies from all libraries (including possible duplicate studies). In the next step (Step 2 in Figure 1) one author selected the set of studies which could be relevant by reading the titles and abstracts of the studies as well as having cursory looking into the studies in case it was hard to predict from the abstract if the study must be included or not. The selection of the studies which could be relevant, at this stage, is based on the inclusion and exclusion criteria explained in Section 3.3. The second author checked once more all the studies which are included and excluded. After this stage we selected 262 studies which needed to be further analysed.

As our search string might have not included some relevant studies that contain other keywords than the selected ones or because we might have excluded a relevant study accidentally, in the next step (Step 3 in Figure 1) we applied the so-called “snowballing” technique. Snowballing is the process of looking into the references of previously obtained studies in order to find more relevant studies [17]. Therefore, we selected new studies from the references based on the same inclusion/exclusion criteria that we used for the first set of studies. After this stage we have added 6 new studies. The process has been repeated on the newly found studies and then no more additional studies have been found. The “snowballing” process was also useful for creating the final search string. Namely, the initial search string was a bit less complex than the final one. It resulted in many more “snowballing” studies as well as studies that are out of interest showing that the search string can be improved. After adding some keywords to make it more general (keywords connected with “or” conjunction) as well as adding some keyword to make it more specific at the same time (keywords connected with “not” conjunction), the final search string presented above was created.

The last step (Step 4 in Figure 1) in our search process is the detailed analysis of the selected set of studies and collecting the relevant metrics. Each study is again checked to satisfy all the inclusion and exclusion criteria. The whole process is shown in Figure 1 with the number of studies obtained in each stage of the study. After detailed reading of 268 collected studies we have selected 25 studies for which the relevant metrics are discussed in the following sections.

3.3 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria are used to make a decision whether to include or exclude a study. We adopted

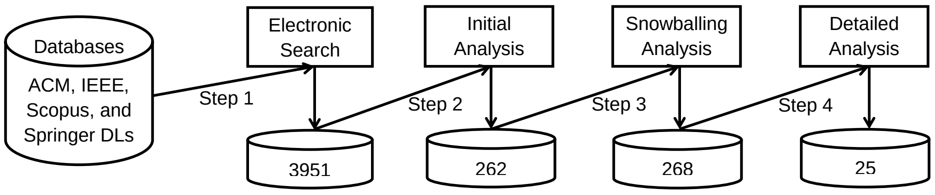

Figure 1: The process for collecting the relevant studies

the following inclusion/exclusion criteria for selecting the relevant studies:

• Include studies about metrics that measure the understandability concepts (explained above) of the higher-level architectural structures (i.e. metrics that operate at the level above classes) with regard to their relations to the system implementation.

• Exclude studies with topics outside the field of software metrics.

• Exclude studies that consider low-level metrics (metrics at the granularity level of methods, attributes, variables and code level metrics), metrics targeting attributes and characteristics other than the understandability concepts, run-time metrics as well as process level or other development metrics because the focus is on software architecture.

• Exclude all inheritance aspects because inheritance is not really relevant from the architectural point of view (for example in UML component diagrams as a kind of component views inheritance is very seldom used).

• Include articles which are published after the year 1990 (till June 2013, the time this study was conducted). Exclude earlier published articles.

• Include the studies published in peer-reviewed journals, conferences, and workshops.

• Exclude books, web sites, technical reports, master thesis and doctoral dissertations (possible low quality studies from grey literature [35]).

The first four criteria aim to make sure that only relevant topics are included. The criterion on the time when the article was published is used to delimit the time frame to the relevant era, as software architecture started to become a research topic in the 90s. For instance, the first IEEE Software special issue on the topic was published in November 1995. The last two criteria were chosen to guarantee a minimum quality level through peer review.

3.4 Maturity Assessment

Assessing the maturity level (i.e. quality assessment) of primary studies in systematic mapping reviews is not essential as discussed by Kitchenham et al. [38]. However, we used it in our study to assess the quality of the proposed metrics in terms of their efficiency and practical applicability. From the practitioners point of view the maturity assessment can serve as an indication of the maturity of the existing research results. From the researchers perspective it can serve as a good starting point for future studies that aim to improve the current state and the identified gaps.

In order to asses the given maturity level we assign the scales to each primary study that concern different aspects of the metrics quality evaluation. The scales are solely assigned based on the information provided in the primary studies. Practical applicability and efficiency are two main aspects that we consider, similar to the study of Kumar et al. [42], but with small differences. In terms of efficiency we consider metric definition, mapping quality [25] (instead of mapping quality, implementation technique is considered in the work of Kumar et al.), level of validation and comparative analysis as main parameters of successful and efficient metrics. In addition to that we consider how difficult is to use or apply the metrics in real situations through three additional parameters usability, applicability and tool support (instead of tool support, extendibility is considered in the work by Kumar et al.). The maturity parameters with corresponding levels are shown in Table 1. Metric Definition (MD) describes the degree of formalism used to define the metrics. Mapping Quality (MQ) represents the level of integration between the metrics and the external quality characteristics (i.e. understandability, maintainability, etc.) they aim to measure. Level of Validation (LV) describes the extent to which the metrics have been empirically validated. Comparative Analysis (CA) provides evidence whether the results obtained from the proposed methodology have been compared with the existing methodologies. Usability (USB) considers how easy it is to use the metrics in projects. In this context we consider how simply the metrics are defined and how much effort is required to understand and calculate them. Applicability (AP) considers how much usage or future prospects of the metrics is specified by the authors. Finally, Tool Support (TS) provides evidence about the tool support for the metrics calculations.

3.5 Information Extraction and Analysis

The extraction and analysis of the relevant information is the last stage of the study. The information extraction is performed by deeply analysing the set of primary studies obtained through the previous stages. This information allowed us to record all relevant details of the primary studies that are necessary to address our research questions. The extracted information from the primary studies and the corresponding mapping to the research questions of our study is shown in Figure 2.

4. RESULTS AND DISCUSSIONS

In this section we discuss the results of our survey with respect to the given research questions.

4.1 Approach Type (RQ1)

First of all we classified all the studies with respect to the representation model they use in order to describe the methodology and context of the metrics, and to define the metrics they propose. During the data analysis we identified 3 different models that correspond to three different approach types:

• **Internal structure based metrics** – The studies classified in this category propose metrics based on the inter-

	Level 1	Level 2	Level 3	Level 4
Metric Definition (MD)	**Informal:** The authors provide a natural language description of the metrics.	**Semi-formal:** Some degree of formalism is provided. The metrics themselves are defined through mathematical expressions, but the underlying concepts being measured (i.e. coupling, cohesion, etc.) are only informally specified.	**Formal:** A formal definition of the metrics and underlying concepts being measured is provided.	–
Mapping Quality (MQ)	**Rationale:** A discussion on the rationale of the mapping is provided.	**Goal Driven:** The metrics are defined to answer some specific evaluation needs following an approach such as Goal-Question-Metric [11] or some similar approach.	**Validated:** Building on the previous level, the metrics are shown to efficiently fulfil the specific evaluation needs.	–
Level of Validation (LV)	**Anecdotal:** Some usually self constructed examples are provided to motivate the usefulness of the metric.	**Small experiment:** The metrics are empirically validated using a small data set/experiments (i.e. one/few small to medium size systems).	**Large experiment:** The metrics are empirically validated using large data sets/experiments (i.e. a lot of small to medium size systems or one/few large systems).	**Independently validated:** The metrics proposed by some authors are successfully validated by another research teams and their applicability is confirmed in the given context.
Comparative Analysis (CA)	**No:** Not used.	**Yes:** Compared with some existing techniques.	–	–
Usability (USB)	**Difficult:** Very difficult to use because of the abstractness and complexity of terms and concepts used to define the metrics.	**Medium:** It requires some effort to understand and calculate the metrics.	**Easy:** Simple to use (counting-based metrics).	–
Applicability (AP)	**Informational:** Just some information about future prospects of metrics applicability is provided.	**Further improvement required:** The authors provide an analysis that can be used for future research but it needs further improvements or more evaluations for reasonable evaluation criteria. For example, some weighted factors that are part of the metrics' formulas need to be estimated, some threshold values or alarms that distinguish between good and bad design need to be found, or further experiments for replication and corroboration purpose need to be conducted.	**Real world applications:** Successful applicability of the metrics in real world applications is provided.	–
Tool Support (TS)	**No:** No tool support is provided.	**Yes:** Tool support is provided.	–	–

Table 1: Maturity Parameters With the Corresponding Levels

nal structure of the measured higher-level artefacts (components, modules, packages, etc.) and their relations. For instance, they consider relations between classes, interfaces or subsystems that constitute higher-level artefacts as well as the hierarchical structure of those artefacts. They do not require any specific model or representation of the system structure in order to define and apply the proposed metrics.

• **Graph based metrics** – The studies classified in this category propose metrics based on a graph model of the given system. Graphs represent system artefacts and their interactions as a set of nodes and edges. The graph based approach is widely used in software engineering because of its inherent simplicity. In the identified studies different types of graphs are considered that model different system artefacts and different kinds of relations between them.

• **Specific model based metrics** – The studies classified in this category propose metrics based on some specific representation or model of the system structure that is required in order to define the metrics they propose. For instance, models that are used are layered architecture models, composite architecture models, etc. The calculation of these metrics requires additional effort in order to construct the given representations of the system.

The distribution of the studies related to the approach type is shown in Figure 3.

4.2 Measured Software Artefacts and Attributes (RQ1)

Different authors claim the importance of both, metrics that measure individual artefacts (components) in the system and metrics that measure the whole architectural representation [2]. In our study we have identified different metrics that measure higher-level structures as well as metrics related to different single artefacts.

The following measured artefacts are identified during the data analysis: architecture, component, component–to–component, package, module, and graph node. Metrics related to the architecture artefact operate at the level of the whole system, i.e. they take into account all system constituent parts (e.g. components, packages, graph nodes,

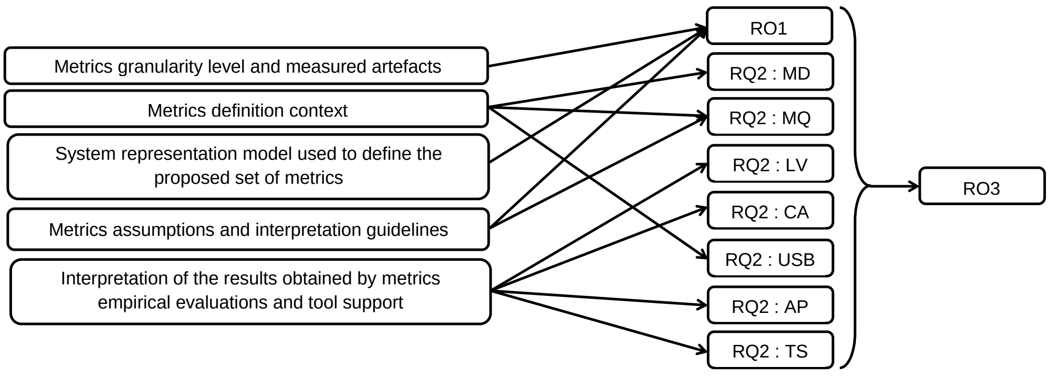
Metrics granularity level and measured artefacts
Metrics definition context
System representation model used to define the proposed set of metrics
Metrics assumptions and interpretation guidelines
Interpretation of the results obtained by metrics empirical evaluations and tool support
RO1
RQ2 : MD
RQ2 : MQ
RQ2 : LV
RQ2 : CA
RQ2 : USB
RQ2 : AP
RQ2 : TS
RO3

Figure 2: Mapping between the information extraction and the research questions

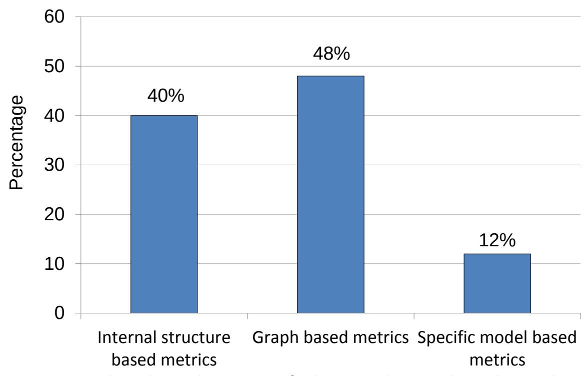

Figure 3: The distribution of the studies related to the approach type

etc.). For example, the metrics like the total number of components or links in the system's component view represent the metrics related to the architecture artefact.

The metrics related to the component artefact are defined from different authors using the term component as a high level artefact in different contexts. Kanjilal et al. [32] considers component as a system element that can be composed with other components, offers a predefined service and is able to communicate with other components. Misic [47] considers component as a set of objects at different abstraction levels (libraries, project objects). Sartipi [58] considers component as a group of system entities in form of a file (to evaluate a design), or module and subsystem (to evaluate the architecture). Shereshevsky et al. [60] consider primitive components (at the lowest level) that exchange the information between each other and do not contain any other components and upper-level components that contain those at the level below them without overlapping. Wei et al. [61] consider components as autonomous pieces of software code with well-defined functionality and interfaces, similarly to the work by Kanjilal et al. At the end Yu et al. [62] consider primitive components, that represent the smallest units, and can be composed into compound components that are further composed into higher level compound components so that the layered component structure is formed (similarly to the work by Shereshevsky et al.). In that context different artefacts can be considered as components such as packages, classes, programs, etc. Taking into account the previous considerations we can say that the term component is used as a higher-level artefact that can be composed of other components or lower-level artefacts and that has well-defined functionality. Component level metrics consider incoming/outgoing interactions of a component, relations between the entities within a component, etc.

Component-to-component metrics take into account pairs of components. Some examples of those metrics are the total number of interfaces between any pair of components or the number of connectors on the shortest path between a pair of components.

Package artefacts are considered also from different authors but they all consider packages as artefacts that contain basic units of organizing code (e.g., the .py files in Python, .class files in Java, .cpp files in C++) as well as other packages leading to a hierarchical structure. Package level metrics are for example the number of classes in a package, the number of other packages that depend upon classes within a package, etc.

Similarly to software components software modules are also considered in different contexts. Anan et al. [7] consider module as a group of components (component represents any abstract high level artefact). Based on dependency analysis components are grouped into modules (architectural slicings). In the work by Hwa et al. [31] modules contain classes as well as other modules which leads to a hierarchical structure very similar to the package hierarchical structure explained above. Lindvall et al. [44] and Sarkar et al. [57] consider modules as sets of classes similar to the work by Hwa et al. Some examples of module level metrics are the number of classes outside a module that are commonly shared by the classes inside a module, the number of classes inside a module that are used by other classes in other modules, etc.

Finally, graph node metrics consider nodes in the graph that is used to represent a software system in a very abstract way. Graph node metrics are for example the degree of a node in a graph, the importance of a node in a graph, etc.

The distribution of the metrics related to the measured artefacts is shown in Figure 4.

Regarding the software attributes that are measured the following categories emerged during the data analysis:

•**Size Metrics** are related to the number of constituent elements of the corresponding design units (artefacts) in the system or to an information theory based size. For example, the number of components and modules are related to the

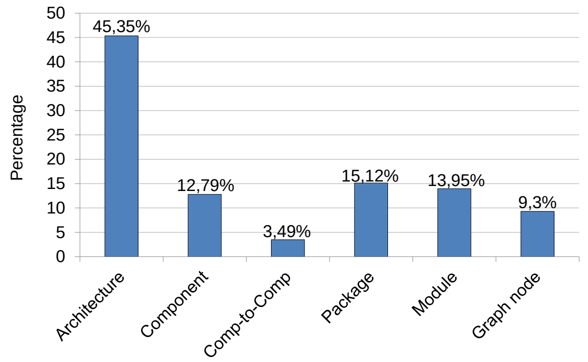

Figure 4: The distribution of the metrics related to the measured software artefacts

overall structure of the system. The number of classes is related to single entities, but also it can be related to the overall structure of the system. Information theory based size metrics calculate the amount of information in the system graph using Shannon entropy.

•**Coupling Metrics** are concerned with the relations between the design units. Those relations are reflected through the number of interfaces, the links or paths between the design units, the extent to which some design units use other design units, the Shannon entropy of the information transmission between design units (information theory based coupling metrics, see for example [5]), etc. Coupling mechanisms are also distinguished in terms of the direction of coupling (import or export coupling), and through direct and/or indirect connections between the design units.

•**Cohesion Metrics** are very similar to the coupling metrics except that they are bound to the relations between the constituting parts of the same design unit (artefact). Functional cohesion introduces external and internal cohesion, where external cohesion considers the relations between the elements inside a given design unit and the elements outside that design unit, while internal cohesion considers relations between the elements inside a given design unit. Cohesion is also measured as the extent to which the elements within one design unit are commonly used from other design units or as information based cohesion that measures the information flow within design units using the aforementioned Shannon entropy.

•**Complexity Metrics** measure the degree of connectivity between elements by taking into account the relationships within design units and between them together. They are concerned with the metrics related to network parameters (graph-based metrics), information theory based complexity, etc. They also measure the hierarchical structure (degree of composition) in the system.

•**Stability Metrics** measure how easy it is to make changes to the elements in a design unit without affecting elements in other design units in the system.

•**Quality Metric** is based on the Multi-Attribute Utility Technique (MUAT) which argues that the quality of a component is decided by its N attributes such as complexity and maintainability [30]. This metric takes into account composite based software architecture which provides a way to separately describe control flow and computation.

Figure 5 shows the distribution of the metrics related to the measured software attributes.

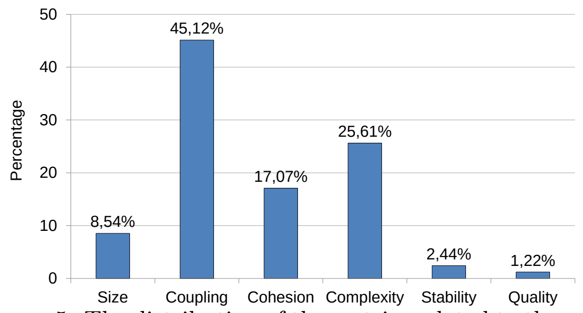

Figure 5: The distribution of the metrics related to the measured software attributes

4.3 Maturity level assessment of the studies (RQ2)

The maturity level of each primary study is assessed by the maturity parameters defined in Section 3.4. Figure 6 shows an overview of the maturity level of each maturity characteristic previously defined. The percentage of the studies that have a given maturity characteristic at the given level is also presented.

Overall, looking at Figure 6, some levels suggest that the metrics defined in the studies have different shortcomings that can be improved. Clearly critical cases are the high percentages of low levels for the MQ (mapping quality) and LV (level of validation) parameters: MQ parameter shows that 68 % of the studies have Maturity Level 1 which means that they just discussed the rationale between the metrics and the external quality characteristics of the system without providing the relationship before using the metrics in order to ensure that the metrics provide a correct evaluation of those attributes that are visible to the user. For example, Allen et al. [5] discussed software attributes that their metrics measure (size, coupling, complexity) without reporting on specific external software quality they can predict. Similarly, Gupta et al. [27] defined package coupling metric without targeting any specific external quality characteristic that the metric can measure. They just informally mentioned that coupling is an important factor that can affect external quality characteristics. However, to prove the real usefulness of the metrics with respect to some external quality characteristic, the rational about the relationship between the metrics and some specific quality characteristic as well as the formal empirical evaluations of that relationship are necessary. 32 % of the studies use a “Goal-Driven” approach where 16 % of the studies correspond to the category “Goal-Driven”, and 16 % of the studies have the highest level, “Validated”. For example, Abdeen et al. [1] defined their metrics according to some principles that support modularization, and facilitate changeability, maintainability and reusability of software systems. Their approach did not provide formal empirical evaluations of those principles and therefore it corresponds to the category “Goal-Driven”. Comparing to them Gupta et al. [28] defined package level cohesion metric using the Goal-Question-Metric (GQM) approach where the quality focus is related to the package reusability, and empirically evaluated the usefulness of the metric in predicting the given quality characteristic (package reusability). Therefore their approach corresponds to the category “Validated”.

Regarding the LV (level of validation) parameter, 56 % of the studies have Maturity Level 1 or 2. This means that

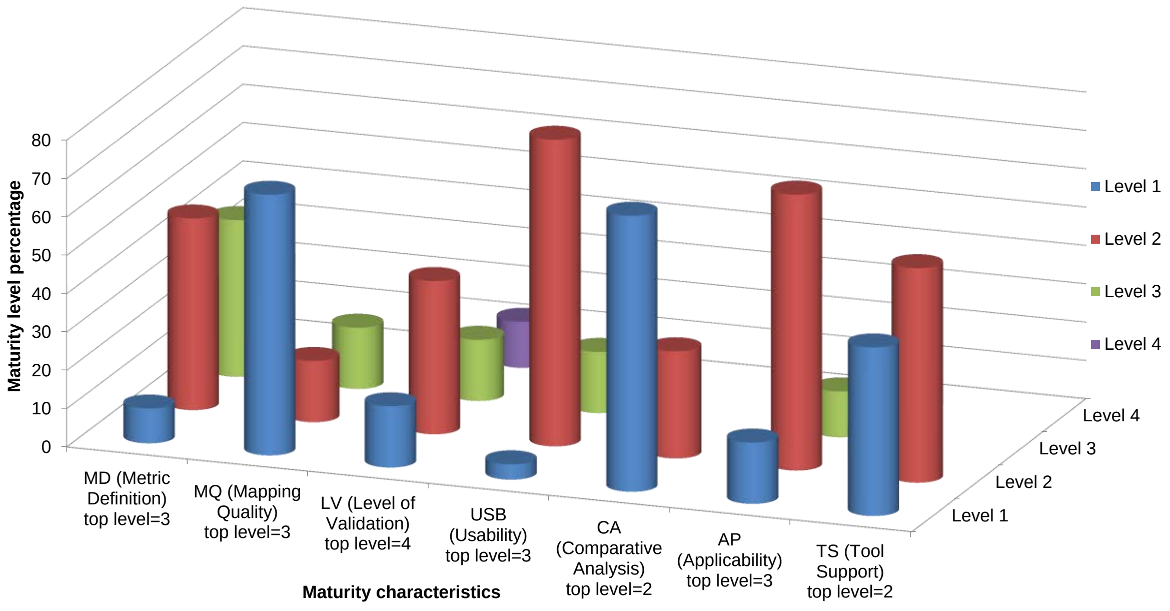

Figure 6: Maturity level assessment of the studies

the metrics validation in those studies is anecdotal (some intuitive/self-constructed examples), or they use small empirical validation (small data sets, i.e. one/few small to medium size systems). For example, Allen et al. [5] provided small experiment validation of their metrics. They used two small systems with 14 and 95 classes. Sartipi [58] also provided small experiment validation using the 5 systems written in C that have 38, 42, 44, 47, and 98 files. Only 16 % of the studies use large empirical validations (large data sets, i.e. a lot of small to medium size systems or one/few large systems), and only 12 % of the studies are independently validated. For example, regarding the large empirical validations Sarkar et al. [57] used a mix of 8 open source systems and business proprietary applications with the number of classes ranging from 308 to 5063 while Salman [54] used 25 systems from graduate students' projects for the validation purposes. 16 % of the collected studies do not have any kind of validation.

An interesting part in the figure is the TS (tool support) parameter. 56 % of the studies have maturity level 2, which means that they provide tool support for metrics calculations. But all of them use proprietary tools or the combination of proprietary and some existing open source tools which limits the metrics portability and comparison as well as knowledge sharing in the research and practitioners' communities.

The rest of the maturity parameters have the following rating scale distributions. Regarding the MD parameter (metrics definitions) 9.1 % of the studies use informal metric definitions, 50 % of them use semi-formal definitions, while the rest of them, 40.9 %, use formal definitions. Semi-formal and informal definitions can hamper metrics calculations and cause misleading interpretations of the results. For example, Lindvall et al. [44] provided informal natural language description of the metrics that capture coupling between modules or modules' classes. From the metrics definitions it is not clear which relationships between classes need to be taken into account (it can just be guessed that all possible relationships should be considered because there were no specified clues of excluding some of them from the consideration). Furthermore, the underlying concept that the metrics measure, i.e. coupling, is not formally specified, i.e. it is not proved whether the metrics satisfy the formal properties of the coupling concept (for the properties of some important measurement concepts see [16]). Hwa et al. [31] provided formal definitions for the metrics but they also did not formally prove that the metrics measure the underlying measurement concepts (e.g. complexity, coupling, cohesion, size). In contrast to the previous two approaches Gupta et al. [27] provided formal definition of the package coupling metric as well as the proof that the metric satisfies the required properties of coupling measure (i.e. non-negativity, monotonicity, null-value, etc.) defined by Briand et al. [16].

The usability parameter (USB) has the following rating scale distribution: 4 % of the studies define metrics that are very difficult to use in projects because of their abstractness and complexity, 80 % of the studies define metrics that have a medium usability level, i.e. they require some effort in order to understand and compute the metrics, and 16 % of the studies contain metrics that can be easily used in projects (counting based metrics). For example, Shereshevsky et al. [60] defined metrics that are very difficult to use. They defined coupling and cohesion metrics based on Shannon entropy, for information exchange between the components in the architecture. The metrics are proposed at a very high abstraction level assuming that the components communicate by requesting the so-called features that have to capture various aspects of information exchange. Salman [54] proposed metrics that are easy to use in a sense that they are based on counting of the corresponding artefacts (e.g. components, links between components, interfaces). Finally, metrics that have "medium" usability require some effort from the user to understand, define and compute them. For example, package coupling metric defined by Gupta et al. [27] requires some effort in terms of understanding which kinds of relations between the classes have to be examined as well as the effort to specify and compute the metric formula.

The CA (comparative analysis) parameter shows that 72 % of the studies did not compare the results of the proposed methodology with the existing techniques, while only 28 % of them did such a comparison. Comparative analysis is one of the main parts of efficient and successful metrics that help users to choose the most appropriate and efficient metric in a given context.

Finally, the applicability (AP) parameter of the metrics shows that 16 % of the studies just discuss the metrics' future prospects and usage, and 72 % of the studies provide good analysis that can be used for future research but still require further improvements or more reasonable evaluation criteria in order to be applied in real projects. Only 12 % of the studies successfully applied the metrics in real projects.

Regarding the metrics validation procedure, the authors tried to show the usefulness of the proposed metrics by validating them with respect to the different quality characteristics. During data analysis we identified the following quality characteristics used for the validity evaluation: maintainability, integrability, complexity, external coherence (how much the artefacts inside one component, package or module are used together from the outside artefacts), understandability, modularization, fault rate or number of modifications, bug severity and reusability. In Figure 7 the percentage of the studies that use the corresponding quality characteristic for the validation of the proposed metrics is presented. Please note that, in contrast to the previously presented distributions of the metrics or studies with regard to the approach type and measured attributes and artefacts, in this case some studies validated metrics with regard to more than one quality characteristic and therefore belong to more than one category (see Table 3 for more details).

In this paragraph we discuss the way how the metrics are validated. One example for each validated quality characteristic is provided. Lindvall et al. [44] validated their metrics with respect to the system maintainability. Based on their coupling metrics they observed that the architectural design of the old version of the system is very tightly coupled while the architectural design of the new improved version that additionally contains 22 new requirements is very loosely coupled. The second observation is that while the structure of the old version indicates that communication between classes in different modules is relatively scattered, the new structure indicates that communication now is localized to a few classes. Sarkar et al. [57] validated their set of metrics with regard to the quality of modularization in large-scale object-oriented systems. The modularization quality is observed with respect to the APIs of the modules, on the one hand, and with respect to different inter-module dependencies, on the other hand. The proposed metrics are validated using a two-pronged approach. In the first approach the values of the metrics conformed to the various attributes gleaned from manual examination of the systems. In the second approach code detuning operators are applied to create different degrees of decay in the analysed systems (they simulate how novice programmers can be expected to extend a code). Metrics values confirmed the degradation of the modularization quality. Reddy et al. [52] used the dependency oriented complexity metrics to evaluate the improvement in quality (reduced complexity) after refactoring. Gupta et al. [27] validated the package coupling metric with regard to the understandability of packages measured by assessing the effort required to fully understand the packages' functionalities and ranking the effort from 1 to 10. They concluded that there is a strong correlation between package coupling and the effort required to understand a package. Misic [47] used some self-constructed examples to show the usefulness of the metrics he proposed for predicting external coherence of a component. The idea of external coherence is to capture the situations where the components are not internally coherent but show functional coherence in terms of common usage from the outside clients. Regarding the fault rate prediction Yu et al. [62] showed that there is a linear relationship between the component coupling metric they proposed and the number of modifications of a component (taken from the change history of the studied system) which is an important indicator of the component fault rate. Bhattacharya et al. [13] tested the metrics in predicting the bug-severity. When a bug for the system is reported, the administrators first review it and then assign it a severity rank based on how severely it affects the program. They showed that modules of high Node Rank (NR) metric are prone to bugs of higher severity. Gupta et al. [28] evaluated the package cohesion metric with regard to the package reusability obtained by developers ratings. The team of developers was required to use each package to complete an unfinished application. The task was to choose the required classes from a given package and integrate them to the application. The authors showed that there is a significant positive correlation between the package cohesion metric and package reusability. Finally, Salman [54] provided evidence about the linear relationship between the total number of links between the architectural components and the integrability of the system defined as the total effort spent on defining inter-component link and component interfaces. Data were collected from the design documents.

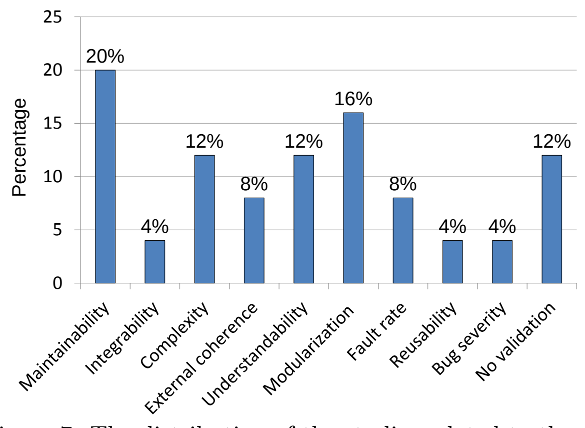

Figure 7: The distribution of the studies related to the validated quality characteristics

Tables 2 and 3 summarize all relevant information related to the results discussed in this section.

4.4 Limitations in the studies and how to mitigate the identified problems (RQ3)

Based on the discussions provided in Section 4.3, we identify the following limitations in the collected set of studies:

- As far as we know, most of the studies did not provide

Reference	**Metrics Names**	**Granularity Level**	**Measured Attributes**
Abdeen et al. [1]	Inter-Package Usage (IPU), Package Changing Impact (PCI), Package Communication Diversion (PCD), Package Goal Focus (PGF), Package Service Cohesion (PSC)	Package, Architecture	Coupling (IPU, PCI, PCD), Cohesion (PGF, PSC)
Alhazbi [4]	Architecture Complexity (ACs)	Architecture	Coupling
Allen et al. [6]	Inter-module Coupling (IeMC), Intra-module Coupling (IaMC), Modular Cohesion (MC)	Architecture	Coupling (IeMC), Cohesion (IaMC, MC)
Allen et al. [5]	Information Size (IS), Information Complexity (ICmp), Information Coupling (ICpl)	Architecture	Size, Complexity, Coupling
Anan et al. [7]	Architectural Module Coupling (AMCpl), Architectural Module Cohesion (AMCoh), Architectural Maintainability Effort (AME)	Module (AMCpl, AMCoh), Architecture (AME)	Coupling (AMCpl, AME), Cohesion (AMCoh)
Bhattach. et al. [13]	Average Degree (k), Clustering Coefficient (C), Node Rank (NR), Graph Diameter (GD), Assortativity (Ass), Modularity Ratio (MR)	Module (C, NR, MR), Architecture (Ass, GD, k)	Complexity
Elish [22]	Number of Classes (NC), Afferent Coupling (Ca), Efferent Coupling (Ce), Instability (I), Distance (D)	Package	Size (NC), Coupling (Ca, Ce), Stability (I, D)
Gupta et al. [27]	Total Coupling of the Package at Given Hierarchical Level (PCM)	Package	Coupling
Gupta et al. [28]	Cohesion of a Package (PCoh)	Package	Cohesion
Haohua et al. [29]	Coupling of Unit (MCC), Importance of Unit (MCI), Efficiency of Message Transmitting (MSE), Modularity and Cohesion (MSM), Metric of Hierarchy (MSH)	Graph node (MCC, MCI), Architecture (MSE, MSM, MSH)	Complexity
Hu et al. [30]	Software Architecture Composite Quality Metric	Architecture	Quality
Hwa et al. [31]	Design Size in Modules (DSM), Module Size in Classes (MSC), Number of API Classes (NAC), Direct Module Coupling (DMC), Number of Disjoint Clusters (NDC), Cohesion by Rest of World (CRW), Depth in Module Hierarchy (DMH)	Architecture (DSM), Module (MSC, NAC, DMC, NDC, CRW, DMH)	Size (DSM, MSC), Coupling (NAC, DMC), Cohesion (NDC, CRW), Complexity (DMH)
Kanjilal et al. [32]	Component Usability Factor (CUF), Component Functionality Factor (CFF), Component Level Coupling (CLC), Node Interaction Factor (NIF), Node-to-Node Coupling (N2NC)	Component (CUF, CFF, NIF), Architecture (CLC), Comp-to-Comp (N2NC)	Coupling
Lindvall et al. [44]	Coupling Between Modules (CBM), Coupling Between Module Classes (CBMC)	Module	Coupling
Ma et al. [45, 46]	Average Shortest Path Length, Degree Distribution, Clustering Coefficient, Betweenness, Degree Correlation	Graph Node (Betweenness), Architecture (others)	Complexity
Misic [47]	Coherence of the Objects Set	Component	Cohesion
Reddy et al. [52]	Dependency Oriented Complexity Metric for the Structure (DOCMS(R)), Dependency Oriented Complexity Metric for an Artefact Causing Ripples (an Artefact Affected by Ripples) (DOCMA(CR), DOCMA(AR))	Architecture (DOCMS(R)), Graph-Node (DOCMA(CR), DOCMA(AR))	Coupling
Salman [54]	Total Number of Interfaces (TNI), Total Number of Components (TNC), Total Number of Links (TNL), Average Number of Links Between Components (ANLC), Average Number of Links per Interface (ANLI), Average Number of Interfaces per Component (ANIC), Depth of the Composition Tree (DCT), Width of the Composition Tree (WCT)	Architecture	Size (TNC), Complexity (WTC, TNC), Coupling (others)
Sarkar et al. [57]	Association Induced Coupling Metric (AC(S)), Size Uniformity Metric (MU), Common Use of Module Classes (CReuM(S))	Architecture (AC(S), MU), Module (AC(S), CReuM(S))	Size (MU), Cohesion (CReuM(S)), Coupling (AC(S))
Sartipi [58]	Self-association Degree (SAD), Association-On Degree (AoD), Association-By Degree (AbD), Modularity Degree (MD)	Architecture (MD), Component (others)	Cohesion (SAD), Coupling (AoD, AbD), Complexity (MD)
Sheresh. et al. [60]	Information Coupling Between two Components (ICC), Information Cohesion of Primitive Components (ICPC), Information Cohesion of Composite Components (ICCC)	Comp-to-Comp (ICC), Component (others)	Coupling (ICC), Cohesion (others)
Wei et al. [61]	Number of Components (NC), Degree of a Vertex (DV), Distance Between Two Components (DTC), Diameter of the System (DS)	Comp-to-Comp (DTC), Component (DV), Architecture (others)	Size (NC), Coupling (others)
Yu et al. [62]	Component Coupling (C (t, d))	Component	Coupling
Zhou et al. [63]	Total Complexity of the Architecture (MT), The Complexity of the Most Easily Affected Component (MS)	Package	Cohesion

Table 2: Summary of the Extracted Data - Metrics Names, Granularity Level, and Measured Attributes

Reference	Approach Type	Validated Quality	Maturity Assessment
Abdeen et al. [1]	Internal structure	–	MD=3, MQ=2, LV=No, USB=2, CA=No, AP=1, TS=1
Alhazbi [4]	Graph-based	–	MD=2, MQ=1, LV=No, USB=2, CA=No, AP=1, TS=1
Allen et al. [6]	Graph-based	Modularization	MD=3, MQ=1, LV=2, USB=2, CA=2, AP=2, TS=2
Allen et al. [5]	Graph-based	Modularization	MD=3, MQ=1, LV=1, USB=2, CA=2, AP=2, TS=1
Anan et al. [7]	Graph-based	Maintainability	MD=3, MQ=2, LV=1, USB=2, CA=1, AP=2, TS=1
Bhattach. et al. [13]	Graph-based	Maintainability, Bug severity	MD=2, MQ=1, LV=3, USB=2, CA=1, AP=3, TS=2
Elish [22]	Internal structure	Understandability	MD=–, MQ=1, LV=4, USB=3, CA=1, AP=2, TS=2
Gupta et al. [27]	Internal structure	Understandability	MD=3, MQ=1, LV=2, USB=2, CA=1, AP=2, TS=2
Gupta et al. [28]	Internal structure	Reusability	MD=3, MQ=3, LV=1, USB=2, CA=2, AP=2, TS=2
Haohua et al. [29]	Graph-based	Complexity	MD=2, MQ=1, LV=3, USB=2, CA=1, AP=2, TS=2
Hu et al. [30]	Specific model	Maintainability	MD=2, MQ=2, LV=2, USB=2, CA=1, AP=2, TS=1
Hwa et al. [31]	Internal structure	Understandability	MD=2, MQ=3, LV=2, USB=2, CA=1, AP=2, TS=2
Kanjilal et al. [32]	Graph-based	Complexity	MD=2, MQ=1, LV=2, USB=3, CA=1, AP=2, TS=1
Lindvall et al. [44]	Internal structure	Maintainability	MD=1, MQ=3, LV=2, USB=2, CA=1, AP=3, TS=2
Ma et al. [45, 46]	Graph-based	Complexity, Fault rate	MD=–, MQ=1, LV=4, USB=2, CA=2, AP=2, TS=2
Misic [47]	Internal structure	External coherence	MD=3, MQ=1, LV=1, USB=3, CA=1, AP=2, TS=1
Reddy et al. [52]	Graph-based	Complexity	MD=2, MQ=3, LV=1, USB=2, CA=2, AP=2, TS=1
Salman [54]	Internal structure	Maintainability, Integrability	MD=3, MQ=1, LV=3, USB=3, CA=1, AP=2, TS=1
Sarkar et al. [57]	Internal structure	Modularization	MD=2, MQ=1, LV=3, USB=2, CA=1, AP=2, TS=2
Sartipi [58]	Graph-based	Modularization	MD=2, MQ=1, LV=2, USB=2, CA=1, AP=3, TS=2
Sheresh. et al. [60]	Specific model	–	MD=2, MQ=2, LV=–, USB=1, CA=1, AP=1, TS=1
Wei et al. [61]	Graph-based	–	MD=2, MQ=1, LV=–, USB=2, CA=1, AP=1, TS=1
Yu et al. [62]	Specific model	Fault rate	MD=1, MQ=1, LV=2, USB=2, CA=1, AP=2, TS=2
Zhou et al. [63]	Graph-based	External coherence	MD=3, MQ=1, LV=2, USB=2, CA=2, AP=2, TS=2

Table 3: Summary of the Extracted Data – Approach Type, Validated Qualities, and Maturity Assessment

an adequate mapping between the proposed metrics and external quality characteristics they aim to measure in order to provide correct evaluation of those attributes. That can obscure the metrics' usefulness for practitioners.

• Insufficient experimental validation leads to a lack of widely accepted metrics that can be used in real world applications. This can reduce trustworthiness of the proposed metrics. It is worth noticing that only three of the proposed studies are validated independently ([22, 46, 45]). Independent validation (not performed by the metrics' main proponents) is very important for the proof of metrics usefulness before common acceptance is sought.

• Insufficient experimental validation leads also to a lack of established metrics threshold values which can obscure interpretations of metrics values by practitioners.

• Ambiguities in metrics definitions occurs when metrics are defined using informal or semi-formal definitions. This leads to misleading and therefore different implementations of metrics collection tools which produces different results for the metrics values calculated by different tools for the same software artefacts.

• Most of the studies use proprietary tools or the combination of proprietary and open-source tools for metrics calculations. This limits metrics portability, their comparisons and knowledge sharing in the research and practitioner communities. This also limits the independent validation of the metrics.

• Insufficient comparative analysis between the given methodology and the existing techniques can hamper the choice of the most efficient and useful metrics that can be applied in a given context.

Mitigating some of the identified problems is proposed by Goulao and Abreu [26]. The proposed approach relies on the usage of a meta-model to formally define the concepts we want to measure and OCL expressions to define the metrics over that meta-model. Applicability of this approach is confirmed through different case studies that are published in that context. The approach can be summarized as follows:

• Defining formal and executable metrics specifications using OCL 2.0.

• Defining formal metrics specifications using a meta-model, such as UML 2.0, thus ensuring that there are no definition and computing ambiguities.

• Packaging metrics specifications in a format that can be used by other practitioners and researchers.

This approach can help in mitigating the problems related to the ambiguities in metrics definitions, metrics portability, accurate calculations and comparisons. The concepts that the metrics measure can also be defined formally together with the guidelines about the mappings between the metrics and/or the measured concepts on one side and external quality characteristics on the other side. Regarding the problems related to the insufficient empirical validations the previously mentioned approach can facilitate the independent metrics validation since it can help in providing the reliable and accurate algorithms for the metrics calculations and therefore enable using the metrics by other practitioners and researchers apart form the metrics main proponents. Accurately computed metrics values can be successfully used as an input for the empirical evaluations of the external software qualities.

5. LIMITATIONS OF OUR STUDY

Even though our study was conducted using the predefined set of rules that lead to systematic investigation of the literature related to the specific research questions, we can identify some limitations in the study. The limitations of our study are: primary study selection bias and inaccuracy in data extraction and therefore quality assessment bias. Primary study selection bias is related to the problem of missing some relevant studies during the search process, while the quality assessment bias is related to the problem of inaccurate extraction and reporting of the study results.

In order to decrease the study selection bias we pursued a systematic search through four digital libraries ACM, IEEE, Scopus, and Springer which include bibliographic references, abstracts, and links to electronic editions of the articles for all major publishers in computer science. In addition, we carried out the snowballing process (see Section 3.2) in order to include potentially excluded studies. Actually the search-query that we used in the search process can lead to false negatives (the studies that are not included in consideration but that are relevant) since some other words or synonyms can be used that are not listed in the query. Looking in the references of the initially found set of papers (snowballing the initial set of papers) gives us an opportunity to find potentially excluded papers and in that way to minimize the given bias. Furthermore, the snowballing process helped us to create the final search-query progressively (see Section 3.2 for more details). The inclusion/exclusion criteria helped us to filter out software metrics outside of the explained scope (see Section 3.3 for more details). The first author performed the search process defined in Section 3.2 and the process of inclusion/exclusion of the studies as defined in Section 3.3 for the initial set of studies and for the snowballing process. The second author independently repeated the process of inclusion/exclusion of the studies for the initial set of studies as well as for the snowballing process.

To reduce the chance of inaccurate data extraction and quality assessment bias, the collected information for each study was checked by the other author. Very few disagreements were resolved by discussions between the authors. The assessment of the metrics' maturity level is solely based on the information provided in the identified studies so that a potential bias related to subjective judgement is minimized.

6. CONCLUSION AND FUTURE WORK

In this article we present a systematic mapping study on software metrics related to the understandability concepts of the higher-level architectural structures with regard to their relations to the system implementation (i.e., metrics that operate at the level above classes). We pursued advanced search through four digital libraries (ACM DL, IEEE DL, Scopus DL, and Springer DL) which include bibliographic references, abstracts, and links to electronic editions of the articles for all major publishers in computer science. Using inclusion/exclusion criteria as well as the snowballing technique (see the Section 3.2) we identified 268 possible studies for detailed analysis. After detailed analysis we identified 25 studies to be reported in our study.

We classify the identified studies and metrics with regard to the measured artefacts, attributes, quality characteristics used for the metrics empirical validation, and representation model used for the metrics definitions. We also provide the maturity level assessment of the identified metrics. Common problems in the current studies are also identified. Overall, there is a lack of maturity in the studies. It is mostly reflected through ambiguities in metrics definitions, lack of empirical validation and lack of information, such as mapping to the quality characteristics they aim to measure and missing comparative analysis, that obscure the metrics' usefulness for practitioners. Tools that exist for metrics calculations are mostly proprietary or the combination of proprietary and open source tools. This limits metrics portability, their comparisons and their knowledge sharing in the research and practitioners' communities. Regarding the identified problems we summarize some existing work that can help in mitigating them.

From the academic point of view we believe that our study can serve as a good starting point for future studies. Identified gaps can help in creating systematic approaches, developing new metrics or tools for metrics calculations or devising empirical experiments for metrics validation, all with the aim to improve the currently existing works. From a practitioner's point of view, the results of our study can be used as a catalogue and an indication of the maturity of the existing research results.

As our future research we aim to focus on empirical evaluation of the identified metrics in terms of their relations to the understandability. This includes carrying out different experiments and user studies to test the utility and the applicability of the metrics, possible metrics adaptations, and creation of new metrics based on the given set of metrics. Such foundational research is essential to provide guidelines and tools to software architects, based on sound evidence, to help them understand how to design the higher-level views that are appropriate for the architectural understanding of a software system and how to best map them to software design models (such as class diagrams) and code.

Acknowledgement

This work was supported by the Austrian Science Fund (FWF), Project: P24345-N23.

7. REFERENCES

[1] H. Abdeen, S. Ducasse, and H. Sahraoui. Modularization metrics: Assessing package organization in legacy large object-oriented software. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 394–398, Washington, DC, USA, 2011. IEEE Computer Society.

[2] M. Abdellatief, A. B. M. Sultan, A. A. A. Ghani, and M. A. Jabar. A mapping study to investigate component-based software system metrics. *Journal of Systems and Software*, 86(3):587 – 603, 2013.

[3] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, Software Engineering Institute (SEI) Carnegie Mellon University, 1997.

[4] S. Alhazbi. Measuring the complexity of component-based system architecture. *Information and Communication Technologies From Theory to Applications 2004 Proceedings 2004 International Conference on*, In Press,(593):593–594, 2004.

[5] E. B. Allen, S. Gottipati, and R. Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Control*, 15(2):179–212, June 2007.

[6] E. B. Allen and T. M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '99, pages

119–, Washington, DC, USA, 1999. IEEE Computer Society.

[7] M. Anan, H. Saiedian, and J. Ryoo. An architecture-centric software maintainability assessment using information theory. *J. Softw. Maint. Evol.*, 21(1):1–18, Jan. 2009.

[8] M. A. Babar and I. Gorton. Comparison of scenario-based software architecture evaluation methods. In *11TH ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, (2004) 600*, page 607, 2004.

[9] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, Jan. 2002.

[10] R. Barcelos and G. Travassos. Evaluation approaches for Software Architectural Documents: A systematic Review. In *Ibero-American Workshop on Requirements Engineering and Software Environments (IDEAS)*, 2006.

[11] V. R. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Approach*, volume I. John Wiley & Sons, 1994.

[12] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[13] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE'12*, pages 419–429, 2012.

[14] G. Booch. *Object-oriented Analysis and Design with Applications (2Nd Ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[15] L. Briand, S. Morasca, and V. Basili. Measuring and assessing maintainability at the end of high level design. In *Software Maintenance ,1993. CSM-93, Proceedings., CONFERENCE on*, pages 88–87, Sep 1993.

[16] L. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1):68–86, Jan 1996.

[17] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Softw. Pract. Exper.*, 41(4):363–392, Apr. 2011.

[18] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[19] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, Boston, MA, 2003.

[20] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, 28(7):638–653, July 2002.

[21] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, July 2009.

[22] M. O. Elish. Exploring the relationships between design metrics and package understandability: A case study. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 144–147, Washington, DC, USA, 2010. IEEE Computer Society.

[23] M. Fowler. Who needs an architect? *IEEE Softw.*, 20(5):11–13, Sept. 2003.

[24] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. 2002.

[25] M. Goulao and F. B. E. Abreu. Software components evaluation: an overview. In *In Proceedings of the 5th Conferência da APSI*, 2004.

[26] M. Goulao and F. B. e Abreu. Independent validation of a component metrics suite, 2004.

[27] V. Gupta and J. K. Chhabra. Package coupling measurement in object-oriented software. *J. Comput. Sci. Technol.*, 24(2):273–283, Mar. 2009.

[28] V. Gupta and J. K. Chhabra. Package level cohesion measurement in object-oriented software. *J. Braz. Comp. Soc.*, 18(3):251–266, 2012.

[29] Z. Haohua, Z. Hai, C. Wei, and A. Jun. The method for measuring large-scale object-oriented software system. In *Proceedings of the 6th international conference on Fuzzy systems and knowledge discovery - Volume 3*, FSKD'09, pages 603–606, Piscataway, NJ, USA, 2009. IEEE Press.

[30] C. Hu, F. Jiao, and C. Zhao. An architectural quality assessment for domain-specific software. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, CSSE '08, pages 143–146, Washington, DC, USA, 2008. IEEE Computer Society.

[31] J. Hwa, S. Lee, and Y. R. Kwon. Hierarchical understandability assessment model for large-scale OO system. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 11–18, Washington, DC, USA, 2009. IEEE Computer Society.

[32] A. Kanjilal, S. Sengupta, and S. Bhattacharya. CAG: A Component Architecture Graph. In *TENCON, IEEE Region 10 International Conference*, 2008.

[33] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. M. Northrop. A basis for analyzing software architecture analysis methods. *Software Quality Journal*, 13(4):329–355, 2005.

[34] B. Kitchenham. Procedures for performing systematic reviews. Technical report, Departament of Computer Science, Keele University, 2004.

[35] B. Kitchenham, P. Brereton, M. Turner, M. Niazi, S. Linkman, R. Pretorius, and D. Budgen. The impact of limited search procedures for systematic literature reviews a participant-observer case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 336–345, Washington, DC, USA, 2009. IEEE Computer Society.

[36] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[37] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*,

51(1):7–15, Jan. 2009.

[38] B. A. Kitchenham, D. Budgen, and O. P. Brereton. The value of mapping studies: A participantobserver case study. In *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering*, EASE'10, pages 25–33, Swinton, UK, UK, 2010. British Computer Society.

[39] K. Knoernschild. *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Agile Software Development Series. Prentice Hall, 2012.

[40] H. Koziolek. Sustainability evaluation of software architectures: A systematic review. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 3–12, New York, NY, USA, 2011. ACM.

[41] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, Nov. 1995.

[42] V. Kumar, A. Sharma, R. Kumar, and P. Grover. Quality aspects for component-based systems: A metrics based approach. *Softw. Pract. Exper.*, 42(12):1531–1548, Dec. 2012.

[43] W. Li and S. Henry. Object-oriented metrics that predict maintainability. 23(2):111–122, February 1993.

[44] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 77–, Washington, DC, USA, 2002. IEEE Computer Society.

[45] Y. Ma, K. He, D. Du, J. Liu, and Y. Yan. A complexity metrics set for large-scale object-oriented software systems. In *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, CIT '06, pages 189–, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Y. Ma, K. He, B. Li, J. Liu, and X.-Y. Zhou. A hybrid set of complexity metrics for large-scale object-oriented software systems. *J. Comput. Sci. Technol.*, 25(6):1184–1201, 2010.

[47] V. B. Misic. Cohesion is structural, coherence is functional: Different views, different measures. *Software Metrics, IEEE International Symposium on*, 0:135, 2001.

[48] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[49] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[50] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *in Proceeding of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, page 1, 2008.

[51] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, european edition, 1994. Adapted by Darrel Ince.

[52] K. N. Reddy and A. A. Rao. A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics. In *Proceedings of the 2009 Second International Conference on Emerging Trends in Engineering & Technology*, ICETET '09, pages 1011–1018, Washington, DC, USA, 2009. IEEE Computer Society.

[53] M. Riaz, E. Mendes, and E. Tempero. A systematic review of software maintainability prediction and metrics. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 367–377, Oct 2009.

[54] N. Salman. Complexity metrics as predictors of maintainability and integrability of software components. pages 39–50. Journal of Arts and Sciences, 2006.

[55] R. Sangwan, P. Vercellone-Smith, and P. Laplante. Structural epochs in the complexity of software over time. *Software, IEEE*, 25(4):66–73, July 2008.

[56] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. J. P. Lucena. On the modularity of software architectures: A concern-driven measurement framework. In *Proceedings of the First European Conference on Software Architecture*, ECSA'07, pages 207–224, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] S. Sarkar, A. C. Kak, and G. M. Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Trans. Softw. Eng.*, 34(5):700–720, Sept. 2008.

[58] K. Sartipi. A software evaluation model using component association views. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 259–268, 2001.

[59] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *SOFTWARE ARCHITECTURE, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 269–272, Sept 2009.

[60] M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, and H. H. Ammar. Information theoretic metrics for software architectures. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 151–, Washington, DC, USA, 2001. IEEE Computer Society.

[61] G. Wei, X. Zhong-Wei, and X. Ren-Zuo. Metrics of graph abstraction for component-based software architecture. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 07*, CSIE '09, pages 518–522, Washington, DC, USA, 2009. IEEE Computer Society.

[62] L. Yu, K. Chen, and S. Ramaswamy. Multiple-parameter coupling metrics for layered component-based software. *Software Quality Journal*, 17(1):5–24, 2009.

[63] T. Zhou, B. Xu, L. Shi, Y. Zhou, and L. Chen. Measuring package cohesion based on context. In *Proceedings of the IEEE International Workshop on Semantic Computing and Systems*, WSCS '08, pages 127–132, Washington, DC, USA, 2008. IEEE Computer Society.