Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Cognitive Model for Software Architecture Complexity

Eric Bouwers, Carola Lilienthal, Joost Visser and Arie van Deursen

Report TUD-SERG-2010-009

TUDelft

SERG

TUD-SERG-2010-009

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication as a short paper in the Proceedings of the International Conference on Program Comprehension (ICPC), 2010, IEEE Computer Society.

© copyright 2010, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

A Cognitive Model for Software Architecture Complexity

Eric Bouwers* ‡, Joost Visser *
* Software Improvement Group
Amsterdam, The Netherlands,
Email: {e.bouwers, j.visser}@sig.nl

Carola Lilienthal
C1 WPS GmbH / University of Hamburg
Hamburg, Germany
Email: carola.lilienthal@c1-wps.de

Arie van Deursen‡
‡ Delft University of Technology
Delft, The Netherlands
Email: Arie.vanDeursen@tudelft.nl

***Abstract*—Evaluating the complexity of the architecture of a software system is a difficult task. Many aspects have to be considered to come to a balanced assessment. Several architecture evaluation methods have been proposed, but very few define a quality model to be used during the evaluation process. In addition, those methods that do introduce a quality model do not neccesarilly explain *why* elements of the model influence the complexity of an architecture.**

In this paper we propose a Software Architecture Complexity Model (SACM) which can be used to reason about the complexity of a software architecture. This model is based on theories from cognitive science and system attributes that have proven to be indicators of maintainability in practice. SACM can be used as a formal model to explain existing quality models, and as a starting point within architecture evaluation methods such as ATAM. Alternatively, it can be used in a stand-alone fashion to reason about a software architecture's complexity.

***Keywords*-Software Architecture Evaluation, Software Architecture, Complexity, Cognitive models**

I. Introduction

Software architecture is loosely defined as the organizational structure of a software system including components, connections, constraints, and rationale [1]. The quality of the software architecture of a system is important, since *Architectures allow or preclude nearly all of the system's quality attributes* [2].

Many architecture evaluation methods have been introduced to evaluate the quality of a software architecture (for overviews see [3, 4]). However, many of these evaluation methods do not define a notion of "quality". Usually, the methodology defines a step or a phase in which evaluators are urged to define which quality aspects are important and how these aspects should be measured and evaluated. In a sense, these aspects and their evaluation guidelines define a quality model for the architecture under review.

Defining such a model is not trivial. It requires specialized skills, experience and time. Of course, such a model can be reused in future evaluations, but the initial investment to start performing architecture evaluations is rather high. Babar et al. [5] conclude that this lack of out-of-the box process and tool support is one of the reasons for the low adoption of architecture evaluations in industry.

To counter this lack of adoption, we recently introduced LiSCIA, a Light-weight Sanity Check for Implemented Architectures [6]. This method is based on system attributes that have shown to be indicators for the maintainability of an architecture [7] and represent experience in evaluating software architectures. LiSCIA comes with a set of questions which together form an implicit, informal model based on the system attributes. However, the method lacks a formal model to explain *why* certain attributes influence the maintainability of the architecture.

A formal model that does provide more explanation has been introduced by Lilienthal [8]. This architecture complexity model is founded on theories in the field of cognitive science and on general software engineering principles. The model has been successfully applied in several case studies. Due to the design of the model and the case studies, the scope of this model is limited to explaining the complexity of an architecture from the individual perspective of a developer. In addition, the validation of the model is only based on systems written in a single (object-oriented) language. Lastly, the model does not provide an explanation for all the system attributes found by Bouwers et al [7].

In this paper, we introduce the Software Architecture Complexity Model (SACM). This model extends the architecture complexity model of Lilienthal by taking into account the environment in which a developer has to understand an architecture. SACM provides an explanation for all system attributes of Bouwers et al., and can thus serve as a formal model to support LiSCIA. In addition, SACM can be used in various situations in both industry and research.

This article is structured as follows. First, the scope of SACM is defined in Section II. More information about the background of the creation of SACM is given in Section III. Section IV provides an overview of the model in terms of its goal, factors and criteria. More detailed explanations of these criteria are given in Section V. In Section VI, we discuss whether the objectives of SACM are achieved, and what the limitations of the model are. Finally, Section VII concludes the paper and explores areas for future work.

II. Scope and Contribution

In the IEEE Standard Glossary of Software Engineering Terminology [9], the term "complexity" is defined as: *The degree to which a system or component has a design or implementation that is difficult to understand and verify*.

Based on this definition, the complexity of a software architecture can be found in the *intended*, or *designed* architecture, and in the *implemented* architecture. The intended architecture consists of design documents on paper, whereas the implemented architecture is captured within the source code.

The main contribution of this paper is the definition of the Software Architecture Complexity Model (SACM). SACM is a formal model to reason about 1) why an implemented software architecture is difficult to understand, and 2) which elements complicate the verification of the implemented architecture against the intended architecture.

While presenting SACM in the following sections, we demonstrate that:

- SACM is defined as a formal model based on the existing theory of quality models
- the factors and criteria that are examined by SACM are grounded in the field of cognitive science and backed up by best practice and architecture evaluation experience
- SACM is independent of programming languages and application context.

In addition, we will argue that SACM can serve as a framework for further research to discover at which point developers experience certain attributes of a software architecture as being too complex.

III. Background

A. Related Work

Over the years, several proposals have been made to define the complexity of an architecture in the form of metrics. The following overview is by no means complete, but is meant to give a small taste of the available work in this area. A more complete overview of complexity metrics can be found in [8, 10].

In many cases, proposed complexity metrics are based on the dependencies between components. For example, it has been suggested to count the number of dependencies [11], or to use the cyclomatic complexity of the used relations [12] between components.

Other metrics defined on the level of the complete architecture can be found in the work of Kazman et al. [13] and Ebert [14]. Kazman defines architecture complexity as a combination of the number of architecture patterns needed to cover an architecture, and which proportion of an architecture is covered by architecture patterns. In contrast, Ebert has constructed a complexity vector based on sixteen measures which have been identified in discussions that took place during a workshop.

In addition to these types of architecture level metrics, both AlSharif et al. [15] and Henderson-Sellers [16] introduce intra-module metrics for complexity. Descending even lower, we can find metrics for complexity on a unit level in, for example, the Chidamber and Kemerer [17] metrics suite and the overview of Zuse [18].

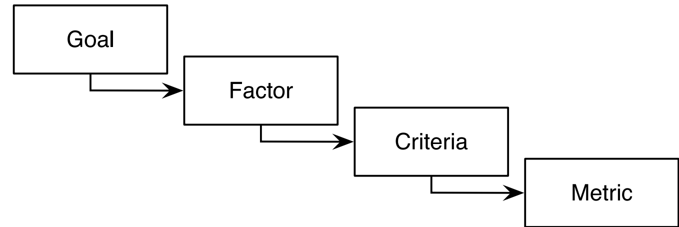

Figure 1. General structure of factor-criteria-metrics-models

A major limitation of all these contributions is the fact that they only provide insight into the complexity of a single or a small set of attributes. Therefore, none of these metrics provide an insight into the complexity of the software architecture as a whole. In order to provide such a complete view, a model which explains the relationship among the separate metrics is needed.

B. Design Process of SACM

One way to provide a framework to express the relationship among metrics is to define a factor-criteria-metric-model (FCM-model) [19]. An FCM-model aims to operationalize an overall goal by reducing it to several *factors*. These factors are still abstract terms and need to be substantiated by a layer of *criteria*. The lowest level in a FCM-model consists of *metrics* that are derived from the criteria. A visual representation of this breakdown can be found in Figure 1.

An existing FCM-model for architecture complexity can be found in the work of Lilienthal [8]. The complexity model of Lilienthal (CML) describes three factors of architecture complexity: *Pattern Conformity*, *Ordering* and *Modularization*. These factors are based upon a combination of theories from cognitive science and general software engineering principles. Each of these factors is translated into a set of criteria (for example the criteria *size of cycle* for the factor *ordering*) which are in turn evaluated using questionnaires and metrics [10].

Unfortunately, two aspects of the design and evaluation of CML prevent it from being used in a general setting. On the one hand, the theories taken from cognitive science focus on a specific level of understanding. On the other hand, the scope of the validation of the CML is limited to systems written in only a single (object oriented) language.

The cognitive science theories underlying CML only consider understanding from within a single person. In other words, the theories only consider how understanding works inside the mind of the individual. In his book *Cognition in the wild* [20], Hutchins argues against the view of the person as the unit of cognition that needs to be studied. Instead, he argues that cognition is fundamentally a cultural process. For instance, in any society there are cognitive tasks that are beyond the capabilities of a single individual. In order to

	Ordering	Modularity	Pattern conformity
Abstraction			X
Functional Duplication			
Layering	X		
Libraries / Frameworks			
Logic in Database			
Module Dependencies	X	X	
Module Functionality		X	
Module Inconsistency		X	X
Module Size		X	
Relation Doc. / Impl.			X
Source Grouping			X
Technology Age			
Technology Usage			
Technology Combination			
Textual Duplication			

Table I
MAPPING OF SYSTEM ATTRIBUTES ONTO THE FACTORS OF CML

solve these tasks, individuals have to work together using a form of *distributed cognition*. The interpersonal distribution of the cognitive work may be appropriate or not, but going without such a distribution is usually not an option.

Within software engineering, the cognitive task of understanding the implemented architecture of a system with, e.g., 4 million lines of code is simply too much for a single individual. Several persons need to work together in order to form a shared understanding and thus be able to reason about changes and improvements of the architecture.

Note that this widened perspective does not mean that personal factors of understanding should be discarded. Instead, a model for software architecture complexity should incorporate factors that deal with the environment in which the software architecture needs to be understood.

The case studies used to evaluate CML were all implemented in Java. Because of this, the complexity of, for example, using different programming languages inside a single system is not taken into account.

Moreover, there are more attributes that influence the complexity of a software architecture that are not considered in the original model of Lilienthal. In an earlier study, Bouwers et al. examined over 40 reports which contain, amongst others, evaluations of implemented software architectures [7]. From these evaluations, a list of 15 system attributes which influence the maintainability of an implemented architecture was extracted. The list of system attributes can be found in the rows of Table I.

Since a system attribute has to be understood before it can be maintained, the expectation is that all system attributes can be explained by CML. For some attributes, the mapping onto CML is straight-forward. For example, the definition of the factor "Pattern conformity" directly corresponds to the definition of the attribute "Relation between documentation and implementation". However, despite various discussions, the mapping of other attributes such as "Technology Combination" and "Technology Age" turned out to be impossible.

The final conclusion of our discussions about mapping the system attributes onto CML can be found in Table I. In the end there was no explanation found for seven of the fifteen system attributes. This shortcoming has lead to the design and definition of SACM. And even though the original CML can still be found in this design, we will show that this extension is novel.

IV. OVERVIEW OF SACM

An overview of SACM is given in Figure 2. At the top, the overall goal of SACM is displayed. To simplify matters, this goal is named "complexity", although strictly speaking the goal of SACM is to evaluate and reduce complexity.

The overall goal is divided into five different *factors*. These factors are partitioned into two different categories. The first category are the *personal* factors. This category captures those factors that play an important role in how a single person internally understands the architecture and include small extensions on the factors from CML.

On the right, the two *environmental* factors are shown. These factors reason about the role the environment of the architecture plays in understanding an architecture. These factors are based on the theory of Hutchins [20] and some of the system attributes that were not explained by CML.

Underneath each factor, a list of *criteria* is displayed. Those criteria with a grey background are taken from CML, while the criteria shown in white are newly defined for SACM. To keep the figure simple, the metrics and questions used to evaluate the criteria are not shown. However, examples of metrics are provided with the description of the criteria in Section V. Even though the description of the personal factors and some of the criteria draw heavily from previous work [8, 10], we feel that all elements should be discussed in order to completely understand the foundations and reasoning underlying SACM.

In Section IV-A several basic terms are defined that will be used in SACM. After this, Section IV-B provides a description of the personal factors, after which the two environmental factors are discussed in Section IV-C. The criteria of the different factors are introduced in the next section. This separation of the description of factors and criteria makes it easier to understand the interplay between the factors, without the need to explain the criteria in detail.

A. Definition of terms

The following terms will be used throughout the description of SACM and therefore deserve a definition: *unit*, *module* and *technology*.

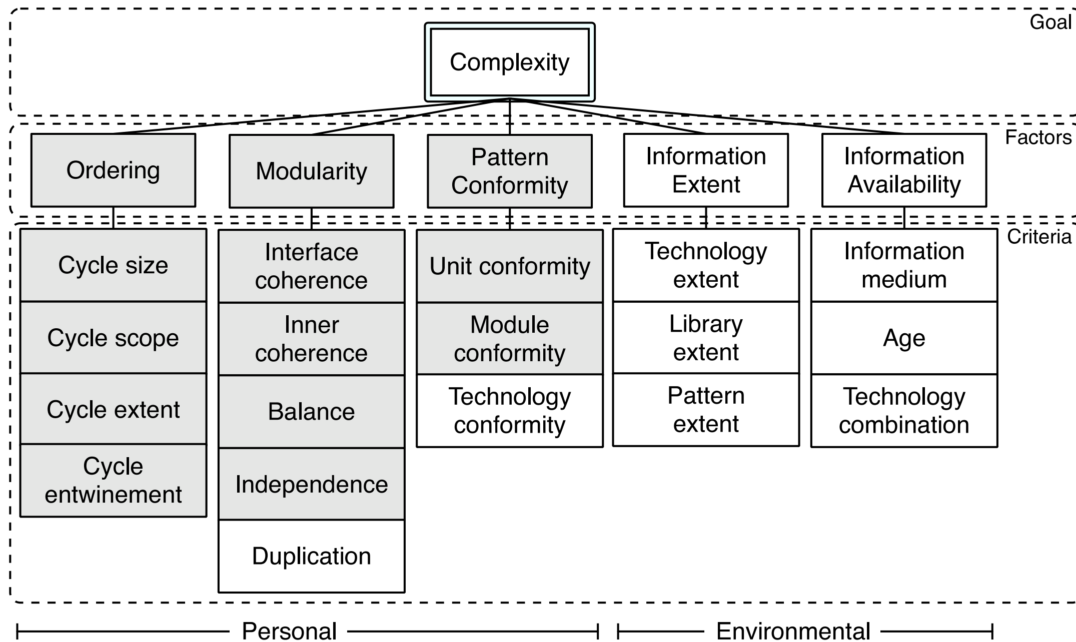

Figure 2. Detailed overview of SACM, grayed factors and criteria denote the original CML [8], white factors and criteria represent our extension.

A *unit* is a logical block of source code that implements some sort of functionality. In object oriented programming languages this normally is a class, whereas in imperative languages the usual unit is usually the file. To capture both paradigms, in the context of SACM a unit can be defined as a source file.

A *module* is a coherent chunk of functionality within a system on a higher level than a unit, such as a package in Java or a namespace in C#.

Within the context of SACM, the term *technology* includes programming languages, build tools and runtime-components such as interpreters and servers. This broad scope is chosen in order to reason about, for example, the choice a certain build tool has on the partitioning of the source code into modules.

B. Personal Factors

A single developer needs to understand an existing software system and its architecture before introducing new functionality or fixing bugs. To understand the architecture, the developer needs to handle a large structure that makes up the implemented architecture, including the large number of elements on different levels (units, modules and layers) and the even larger number of relationships between those elements. In addition, the software developer has to map the intended architecture onto the implemented architecture in the source code.

To model the strategies used for this understanding, Lilienthal turned towards the field of cognitive science. This area examines how human beings deal with complex contexts in their daily live. In this context, the question that is answered by cognitive science is “How do human beings understand and learn about complexity, and how do they solve problems related to complexity?”

Within cognitive psychology, three mental strategies have been described that are used by human beings to deal with large structures and complex mappings [21, 22]: chunking, formation of hierarchies and the development of schemata.

Chunking refers to a strategy for making more efficient use of short-term memory by recoding detailed information into larger chunks. This strategy can only be applied successfully if the information to be chunked represents a meaningful cohesive unit.

With the *formation of hierarchies*, human beings try to structure information on different levels, analogous to chapters in a book. Most of the time human beings combine formation of hierarchies and chunking to handle a huge amount of information.

Schemata on the other hand are mental structures that represent some aspects of the world in an abstract way. For example, most people have a teacher schema and can apply it to a person they have never seen before. Just mentioning that someone is a teacher instantly activates our schema and we know the job of the person, and attach certain characteristics to the person. People use schemata to organize current knowledge and provide a framework for future understanding. By organizing new perceptions into existing schemata, people can act efficiently without effort.

A number of investigations have been carried out to verify

that chunking, hierarchies and schemata can help software developers to cope with large structures [23, 24, 25, 26, 14]. In parallel, general design principles have been developed in the last 40 years that support these mental processes: modularization [27, 28] and abstraction through interfaces [27] to foster chunking, avoiding cyclic structures [29, 16, 30] and layering [31, 32] to support hierarchies as well as design patterns and architectural styles [33, 34] to facilitate schemata.

Based on the three mental strategies from cognitive psychology and the above listed general design principles, we define three personal factors for architectural complexity to be applied on all levels of abstraction (from methods to units and to modules):

- Modularity: checks whether the implemented architecture is decomposed into cohesive elements that encapsulate their behavior and offer a cohesive interface.
- Ordering: checks whether the relationships between elements in the implemented architecture form a directed, acyclic graph.
- Pattern conformity: checks whether the pattern of the intended architecture and the technology in use can be recognized in the implemented architecture, and whether their rules are adhered to.

C. Environmental Factors

In an ideal case, all parts of an implemented architecture of a system can be understood by a single person at a global level of abstraction. However, due to the size of modern software systems this cognitive task exceeds the capabilities of a single person. In addition, the speed at which the implemented architecture needs to be adapted to changing business requirements can demand several people to work together in some form of distributed cognition.

This situation is similar to the scenario of navigating a ship as described by Hutchins [20]. A single person can perform all the cognitive tasks to navigate a ship when on open sea, but navigating a ship safely into a port requires a substantial navigation team.

A reason for the need of distributed cognition is the amount of information that needs to be processed in a certain time interval. In order to lower the amount of distributed cognitive work needed, one strategy is to lower the amount of information needed to understand the implemented architecture.

Within the context of SACM, the term "information" refers to every piece of meaningful data that is needed to understand the architecture. How easy it is to understand pieces of information depends on the experiences of a specific person. This is one of the reasons why "complexity" is a subjective term. After all, an experienced Java developer will have less trouble understanding a system programmed in Java than a system implemented in, for example, Cobol. However, when more information needs to be understood, it is less likely that a single person can provide all the context for this information himself. When this happens, the context for the information has to be determined by interacting with external entities, which complicates the process of understanding and is thus more complex. Because of this, the extent of information that needs to be understood should be kept to a minimum.

In order to be able to process the needed information, it is important that the information is actually available. After all, understanding the implemented architecture of a system is virtually impossible when the implementation of the system is not available. In addition, the representation of the available data contributes to the complexity of understanding the data. For example, it is usually easier to grasp the allowed dependencies between modules when these dependencies are shown in a picture, as opposed to when the dependencies are specified as a list of text.

Based on these observations, we define two environmental factors for architectural complexity: *information availability* and *information extent*, where the latter refers to the total amount of information that needs to be understood.

V. Criteria of SACM

Now that the factors of SACM are defined and their relation is made clear, the criteria supporting each factor can be explained. As illustrated in Figure 1, some of the criteria in SACM are based upon the earlier work of Lilienthal [8]. Those attributes that are not based upon CML are partly inspired by the system attributes of Bouwers et al. [7].

The criteria of the personal factors are explored in Section V-A, Section V-B and Section V-C, after which the environmental factor are discussed in Section V-D and Section V-E. We have chosen to define all criteria in full, in order to make sure that the model is self-contained.

A. Pattern Conformity

The factor *pattern conformity* checks whether the patterns that have been defined for a software can be found, and are adhered to in the implemented architecture. There are various sources of patterns in software architecture, such as design patterns, architectural patterns, best practices, coding standards and the intended architecture. Each of these patterns addresses a different aspect of the software architecture. Design patterns give guidance for the interfaces of programming units and the interaction between them. The intended architecture and architectural patterns, such as a layered architecture, apply to the level of modules in a software architecture. Finally, best practices and coding standards stem from the technology that is used.

To capture these three aspects of pattern conformity we have defined the following criteria:

1) Unit conformity: checks patterns on unit level
2) Module conformity: checks patterns on module level

3) Technology conformity: checks the usage of the technology against best practices and coding standards

To evaluate pattern conformity, metrics, (automated) code reviews, information from the documentation and interviews with the system's architects have to be combined. For example, the patterns defining the dependencies between modules should first be extracted from the documentation or from interviews with the architects of the system. This data can then be combined with the (static) dependencies to partly evaluate the module conformity. As another example, a language specific code-style checker can be used to check the conformity on technology level.

B. Ordering

The factor *ordering* checks whether the relationships between elements in the implemented architecture form a directed, acyclic graph. This factor operates on both the unit and module elements within the architecture. The first step in evaluating this factor is to check whether the graph of elements is acyclic. When this is not the case, a more detailed evaluation of the cycles must be conducted to assess the difficulty of removing the cycles from the architecture. To conduct this detailed evaluation we have defined the following criteria:

1) Cycle size: checks how many artifacts and dependencies are part of cycles in the system
2) Cycle extent: checks how many artifacts belong to the largest cycles
3) Cycle entwinement: checks how many bidirectional dependencies can be found in the cycles
4) Cycle scope: checks whether module cycles are based on unit cycles

With the first criteria *cycle size* we get an indication how ordered the evaluated software architecture is. Our expert opinion state that systems with more than 10 percentage of units in cycles tend to be hard to maintain. But if all the cycles are small, for example 2 or 3 units, the problem is not that big. Therefore the second criteria *cycle extent* investigates the extent of the largest cycles. Still the extent of cycles gives us no final indication whether they will be hard to remove or not. The third criteria *cycle entwinement* provides us with an answer to this question. If the artifacts in a cycle are connected by several bidirectional dependencies, they are strongly linked with each other and breaking up the cycle will lead to a redesign of all the artifacts. If there is only one dependency in a cycle going against the ordering and all other dependencies are not bidirectional the case is much easier. Finally, the forth criteria *cycle scope* deals with cycles on the module level that are based on unit cycles. A pure module cycle reveals the unsound sorting of some units. If these units are moved to other modules the module cycle is resolved. If the module cycle is based on a unit cycle resorting the units will not solve the problem, but the units in the cycle will have to be redesigned to remove the module cycle.

On the base of these four criteria a profound evaluation of ordering in software architectures becomes possible. Metrics to evaluate these criteria can be found in tools that check dependencies and in the extensive literature on graph-theory.

C. Modularity

The factor *modularity* checks whether the implemented architecture is decomposed into cohesive elements that encapsulate their behavior and offers a cohesive interface. In other words, each module in the system should have a single responsibility and should offer access to that one responsibility in a consistent way. If this is done correctly and in a balanced way, it becomes relatively easy to recognize these chunks as a whole and treat them as one block of functionality. Apart from these intra-module properties, there are some inter-module properties that are desirable to avoid confusion. Orthogonal to the fact that each module should have a single responsibility, each responsibility in the system should be encoded in a single module. This is to avoid a situation in which a change in functionality causes changing multiple modules. In addition, each module should be fairly independent, in order to avoid that a change in a single module trickles through to many other modules.

The criteria defined for this factor are:

1) Inner Coherence: checks whether each module has a single responsibility
2) Duplication: checks whether each responsibility is encoded in a single module
3) Interface Coherence: checks whether each module offers access to a single responsibility
4) Independence: checks how cross-linked a module is
5) Balance: checks the distribution of system size over the modules

There are many metrics available to help in evaluating these criteria. Within the related work section, several metrics regarding independence have been named. In addition, metrics such as the percentage of textual and functional duplication in the system can be used to evaluate the duplication criteria. To assess the criteria of inner and interface coherence, the description of the functionality and interface of each module can be used. By combining these metrics by manual code inspection of the encoded functionality, a balanced assessment can be made.

D. Information extent

The factor *information extent* checks the amount of information needed to understand the implemented architecture. A large portion of this information need is dictated by the technologies which are used within the implementation. In order to comprehend the function each technology fulfills within the architecture, information about the semantics, the syntax and the task of each technology is needed. The

more technologies are used, the bigger the total extent of information will be. For example, the information extent for a system implemented in only the Java language is smaller then a system which is implemented using Java, HTML and Javascript.

A strategy to reduce the information extent is the usage of pre-existing solutions such as libraries, frameworks and design patterns. In order to use these solutions, a developer only needs to understand *which* task the solution accomplishes, not *how* the task is accomplished. When the information needed to be able to apply a solution is lower then the information needed to encode the solution itself, the information extent is reduced.

For example, in the development of an interactive website it is a good idea to use an existing library to abstract over the implementation details of different web-browsers. The information needed to understand what the library does is less than the information needed to understand how the details are handled. On the other hand, if only a small part of a library is used, the information needed to understand what the library does might not outweigh the amount of information that it hides. Because of this, it is important to not only assess whether libraries can be used, but also to take into account the trade-off between the reduced and required information extent.

Based on these observations, we define the following criteria for this factor:

- Technology extent: checks the used technologies
- Library extent: checks the used libraries, their coverage and whether parts of the system could be replaced by available libraries.
- Pattern extent: checks the used patterns and whether certain parts of the system can be replaced by standard patterns.

In order to evaluate the first criteria, the list of used technologies, the percentage of the system which is implemented in each technology and descriptions of the tasks of each technology has to be assembled. For the second criteria, metrics such as the list of libraries used, the percentage of the usage of each library, and the percentage of functionality within the system that could be captured by an existing library have to be asserted. Lastly, metrics for the third criteria are the list of pattern definitions which are used and a description of common strategies used within the system.

E. Information availability

The aim of the factor *information availability* is to assess the availability of information about the architecture. One aspect that greatly influences the availability of information is age. Due to erosion of the information, getting up-to-date information about an old system, technology or library becomes increasingly complex. For example, locating an experienced Java developer is currently quite easy, in contrast with hiring an experienced Algol developer.

Another aspect that complicates the gathering of information is the way technologies are combined. For example, the combination of Java and Cobol in a single system is quite rare. Because of this, there is little documentation available of how this integration can be accomplished. This in contrast with finding information about using Java together with Java Server Pages.

However, apart from the fact that it should be possible to obtain information, it is also important that the information can be placed in a meaningful context. How easy it is to place information in a meaningful context is greatly dependent on the medium of the information.

For example, consider the difference between just having access to the source code of a system, or having access to a developer who has worked on the system. In the second situation, understanding the architecture becomes a less complex task because 1) the developer is capable of providing both factual information of the system and a context, and 2) it is easier to interact with the developer in order to extract information because the developer can more easily understand what you want to know.

To capture these aspects of this factor, we define the following criteria:

1) Information medium: checks the medium on which information is available
2) Age: checks the age of the used technologies, libraries and the system
3) Technology combination: checks whether the combination of used technologies is common

Examples of metrics for the first criteria are “amount of textual information”, “number of experienced developers on the development team” and “language of the documentation”. For the second criteria, metrics such as “year of first release” and “size of community” can be used. Evaluating the last criteria can be done by counting the number of times a technology is mentioned in the documentation of the other technology, or locating predefined interfaces within the technology itself.

VI. Discussion

Section II states some desirable properties of SACM . We discuss these properties in Section VI-A till Section VI-D. In addition, the application of SACM in industry is discussed in VI-E, after which the application of SACM in a research setting is discussed in Section VI-F. Lastly, a number of limitations of SACM is discussed in Section VI-G.

A. SACM is a formal model

Figure 2 shows the general design of SACM. In this figure, the overall goal of SACM is divided into several factors, which are decomposed into measurable criteria. In Section V, different metrics and questions are given to support the evaluation of the individual criteria. This design

	Ordering	Modularity	Pattern conformity	Information extent	Information availability
Abstraction			X		
Functional Duplication		X			
Layering	X				
Libraries / Frameworks				X	
Logic in Database				X	
Module Dependencies	X	X			
Module Functionality		X			
Module Inconsistency		X	X		
Module Size		X			
Relation Doc. / Impl.			X		
Source Grouping			X		
Technology Age					X
Technology Usage			X		
Technology Combination					X
Textual Duplication		X			

Table II
MAPPING OF SYSTEM ATTRIBUTES ONTO THE FACTORS OF SACM

corresponds to the setup of general FCM-models. Therefore, we conclude that SACM is a formal model.

B. SACM is based on both theory and practice

In Section IV, the relationship between the factors and existing theories from cognitive science are given. This shows that the theoretical basis of SACM can be found within this field.

In addition, part of the explanation of the criteria in Section V consist of examples taken from real-world situations. Also, the grey part of the model as displayed in Figure 2 is mainly based on CML, which is already validated in practice. The basis of the new criteria and factors in the model are on the one hand the theories from Hutchins [20], and on the other hand the system attributes of Bouwers et al. [7]. Because of this fact, we conclude that SACM is based upon both theory and practice.

C. SACM is independent of programming languages and application context

In none of the factors, criteria or metrics (excluding examples) there is a reference to a specific type of programming language or application context. In contrast, the environmental factors of SACM explicitly take into account the usage of multiple technologies within a single system. Also, the definition of the terms used in SACM given in Section IV-A are designed to capture all application contexts. More generally, SACM does not make any assumptions towards the domain, the semantics or the requirements of the application. This ensures that SACM can be used as a basis to evaluate a wide range of applications.

D. SACM is novel

SACM is not a straight-forward combination of existing approaches, but rather innovates over earlier work in a number of different ways. First of all, SACM extends CML on both the factor and criteria level, see Figure 2. Secondly, the definition of, e.g., "Pattern Conformity" has been extended in order to be more generally applicable. Lastly, several criteria, such as "Information Medium" and "Technology Usage", are based neither upon CML, nor the system attributes. Instead, these criteria are completely new and explain the new insights obtained during our discussions about the mapping of the system attributes onto CML.

E. SACM in industry

One of our first goals was to define a formal model that could be used to explain the system attributes found by Bouwers et al. Table II shows that SACM is capable of doing this. Because of this, SACM can be used as a model inside the Software Risk Assessment [35] process of the Software Improvement Group (SIG) and as a formal backup for the Light-weight Sanity Check for Implemented Architectures [6]. This shows that there is a direct application of SACM within an industry setting.

More generally, we believe that SACM can be used to reduce the initial investment needed to start performing architecture evaluations. Since there are no external dependencies, the SACM can be "plugged into" existing software architecture evaluation methods such as, for example, the Architecture Tradeoff Analysis Method [2]. Doing this does not only speed up the evaluation process, but the usage of an uniform model across evaluations allows the direct comparison of evaluation results. This is useful in, for example, the process of supplier selection.

F. SACM in research

Using SACM as a basis, we envision two different areas of further research. A first area is the development of new metrics. Currently, in order to provide a balanced assessment of some criteria, metrics need to be augmented by expert opinion. In order to lower this dependency on the opinion of experts, metrics that capture these opinions could be developed. For example, the quality of evaluating whether a certain combination of technologies is common depends mainly on the experience of the evaluator. Developing metrics to support this opinion is one area of future work.

The second area of research can be found in the definition of thresholds. The basic question here is "which value of metric X indicates that criteria Y is too complex?". This question can be answered by several approaches. A first approach is statistical benchmarking. By computing the value of a metric for a large group of systems, the "normal" value of the metric can be found, everything above this value is non-standard and therefore indicates that the criteria is becoming too complex.

A different approach to determine the threshold value is by conducting experiments similar to those performed in the area of cognitive science. For example, one can try to determine the threshold value for a metric by letting subjects examine pieces of source code with different values of the metrics and ask them to answer questions related to understanding. This last approach is especially well suited to find out whether a higher value of a metric always means that the source code is more difficult to understand. Taking into account the different strategies for understanding, our hypothesis is that the statement “a higher value means more complex code” is not correct for all complexity metrics. Validating this assumption is also part of our future work.

G. Limitations

In order to use every part of SACM, an evaluator should have access to both the implemented and the intended architecture. If this is not possible, and only the implemented architecture is available, the assessment of, the factor “pattern conformity” can only be done with the criteria ”Technology Conformity”. This is because no unit or module patterns are available. The remaining factors and criteria that only check the implemented architecture are nevertheless worth to be applied to achieve some results about the complexity of the implemented architecture. The same holds true for utilizing SACM to an intended architecture on paper. “Pattern conformity” can not be checked on a intended architecture alone, but many other factors and criteria will generate valuable results to judge the intended architecture.

A more fundamental limitation of SACM is that there is no formal proof to show that the model is complete. Even though parts of SACM are used in evaluating over sixty systems, it could be that some factors of complexity are not captured by the model. However, since there is also no proof that the model of “understanding” in cognitive science is complete, we accept this limitation of the model. Therefore, during the application of SACM one should always be on the lookout for new factors that can help to capture complexity.

H. Evaluation

The CML part of SACM has been validated in over 25 case studies. In addition, the criteria which are based upon system attributes can build upon data of over forty case studies. However, a study to formally validate SACM as a whole has currently not been conducted. Therefore, only anecdotical evidence of the usefulness of the complete model can be given. In order to formally support these positive first results, we plan to apply SACM in a number of formal case studies within SIG.

VII. Conclusion

In this paper, we introduced the Software Architecture Complexity Model. This formal model can be used to reason about the complexity of an implemented software architecture and is founded upon both theories from cognitive science and general software engineering principles. Several desirable properties of SACM have been explained, as well as how the model can be used in both industry and research.

As future work, we plan the use SACM as a theoretical basis within the practice of evaluating software architectures. By doing this, we intend to provide an empirical validation of the newly defined criteria, factors and the model as a whole.

In a different area of the future work, we plan to use SACM as a basis for experiments to find thresholds for complexity metrics based on theories from cognitive science. In addition, we plan to develop new metrics to support the evaluation of the criteria of SACM.

Acknowledgment

The authors would like to thank all their colleagues for the interesting discussions and their willingness to provide feedback.

References

[1] P. Kogut and P. Clements, “The software architecture renaissance,” *The Software Engineering Institute, Carnegie Mellon University*, vol. 3, pp. 11–18, 1994.
[2] P. Clements, R. Kazman, and M. Klein, *Evaluating software architectures*. Addison-Wesley, 2005.
[3] M. Babar, L. Zhu, and D. R. Jeffery, “A framework for classifying and comparing software architecture evaluation methods.” in *ASWEC ’04: Proceedings of the 2004 Australian Software Engineering Conference*. IEEE Computer Society, 2004, p. 309.
[4] L. Dobrica and E. Niemelä, “A survey on software architecture analysis methods,” *IEEE Transactions of Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.
[5] M. Babar and I. Gorton, “Software architecture review: The state of practice,” *Computer*, vol. 42, no. 7, pp. 26–32, 2009.
[6] E. Bouwers and A. van Deursen, “A lightweight sanity check for implemented architectures,” *IEEE Software*, vol. 27, no. 4, 2010.
[7] E. Bouwers, J. Visser, and A. van Deursen, “Criteria for the evaluation of implemented architectures,” in *ICSM ’09: Proceedings of the 25th International Conference on Software Maintenance*. IEEE Computer Society, 2009, pp. 73–83.
[8] C. Lilienthal, “Architectural complexity of large-scale software systems,” in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2009, pp. 17–26.
[9] “IEEE Std 610.12-1990: IEEE standard glossary of software engineering terminology,” 1990.

[10] C. Lilienthal, “Komplexität von Softwarearchitekturen, Stile und Strategien,” PhD Dissertation, Universität Hamburg, Software Engineering Group, 2008.

[11] J. Zhao, “On assessing the complexity of software architectures,” in *ISAW ’98: Proceedings of the third international workshop on Software architecture*. ACM, 1998, pp. 163–166.

[12] T. McCabe and C. W. Butler, “Design complexity measurement and testing,” *Commun. ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.

[13] R. Kazman and M. Burth, “Assessing architectural complexity,” in *CSMR ’98: Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR’98)*. IEEE Computer Society, 1998, p. 104.

[14] C. Ebert, “Complexity traces: an instrument for software project management.” International Thomson Computer Press, 1995, pp. 166–176.

[15] M. AlSharif, W. P. Bond, and T. Al-Otaiby, “Assessing the complexity of software architecture,” in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. ACM, 2004, pp. 98–103.

[16] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[17] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, 1994.

[18] H. Zuse, *Software Complexity: Measures and Methods*. Berlin, Germany: Walter de Gruyter & Co., 1990.

[19] J. A. McCall, P. K. Richards, and G. F. Walters, “Factors in software quality,” in *US Rome Air Development Center*. NTIS AD/A-049 014,015,055, 1977, pp. Nr. RADC TR–77–369, Vols I,II,III.

[20] E. Hutchins, *Cognition in the wild*. MIT Press, 1996.

[21] J. Anderson, *Cognitive psychology and its implications*. W.H.Freeman & Co Ltd, 2000.

[22] D. Norman, *Learning and Memory*. W. H. Freeman & Co, ACM Press, 1982.

[23] M. Haft, B. Humm, and J. Siedersleben, “The architect’s dilemma - will reference architectures help?” in *Quality of Software Architectures and Software Quality QoSA/SOQUA*, R. e. Reussner, Ed., 2005, pp. 106–122.

[24] H. A. Simon, *The Sciences of the Artificial*. MIT Press, 1996.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[26] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, April 1996.

[27] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[28] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[29] B. Brügge and A. Dutoit, *Object oriented Software Engineering Using UML, Pattern, and Java*. Prentice Hall International, 2009.

[30] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, “Cognitive design elements to support the construction of a mental model during software exploration,” *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[31] A. Cockburn, “People and methodologies in software development,” PhD Dissertation, University of Oslo, Faculty of Mathematics and Natural Sciences, 2003.

[32] H. Melton and E. Tempero, “An empirical study of cycles among classes in java,” *Empirical Software Engineering*, vol. 12, no. 4, pp. 389–415, 2007.

[33] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[34] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

[35] A. van Deursen and T. Kuipers, “Source-based software risk assessment,” in *ICSM ’03: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003.

TUD-SERG-2010-009
ISSN 1872-5392