Available online at www.sciencedirect.com
ScienceDirect
Advanced Engineering Informatics 21 (2007) 169–179
ADVANCED ENGINEERING INFORMATICS
www.elsevier.com/locate/aei

Classifying architectural constraints as a basis for software quality assessment

Simon Giesecke [a,*], Wilhelm Hasselbring [a], Matthias Riebisch [b]

[a] Carl von Ossietzky University Oldenburg, Software Engineering Group, 26111 Oldenburg, Germany
[b] Technical University Ilmenau, Department of Software Systems/Process Informatics, P.O. Box 10 05 65, 98684 Ilmenau, Germany

Received 10 October 2006; accepted 12 November 2006

Abstract

Architectural styles and patterns have been studied since the inception of software architecture as a discipline. We generalise architectural styles, patterns and similar concepts by introducing the notion of *architectural constraints*. An architectural constraint is a vehicle for the reuse of architectural design knowledge and for the improvement of software quality. It may be used for improving architectural analyses of quality characteristics of the software system to be realised. We present the method for surveying the literature on architectural constraint concepts, and provide a taxonomy covering various definitions of architectural styles and patterns.

Keywords: Architectural style; Architectural constraints; Software quality assessment

1. Introduction

Software architecture as a discipline within Software Engineering has been emerging since the seminal paper on ''Foundations for the study of software architecture'' by Perry and Wolf [25] and the book by Shaw and Garlan [29], while the term ''software architecture'' is much older, tracing back to Sharp [26, p. 12] in the 1960s [19].

We survey the current state of research in the field of software architecture under a specific view. Hereby, we aim to provide tangible results by focusing on what we call *architectural constraint concepts*, which form part of the rationale used to establish, maintain and employ an architectural description. Their rationale has a strong influence on the way of making decisions during the architectural design process. Terms referring to architectural constraint concepts encompass ''architectural style'', ''architectural pattern'', ''architectural metaphor'', and the like. The relationship between terms and concepts is ambiguous in this context: often several terms are used interchangeably even by the same authors, while other authors seem to implicitly distinguish multiple concepts, without providing a sufficiently rigorous definition. The lack of a definition causes confusion in communication, inappropriate generalisations, and hampers progress in this knowledge domain. Szyperski [30, p. 36] concludes: ''As precision and richness of the vocabulary decrease, so does the richness of expressible and distinguishable, yet concise, thoughts.'' By establishing a coherent conceptual framework, we aim to overcome these problems.

In this paper, we identify two main classes of architectural constraints:

- Pattern-based concepts on the one hand, including the work by Gamma et al. and Buschmann et al.
- Style-based concepts that include the work of Perry and Wolf, Allen and Garlan, and Medvidovic and Taylor.

1.1. Architectural constraints in architectural design

The quality of the architectural design process and of architectural descriptions is strongly influenced by

* Corresponding author. Tel.: +49 441 798 2991; fax: +49 441 798 2196.
E-mail addresses: giesecke@informatik.uni-oldenburg.de (S. Giesecke), hasselbring@informatik.uni-oldenburg.de (W. Hasselbring), matthias.riebisch@tu-ilmenau.de (M. Riebisch).

1474-0346/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.aei.2006.11.002

architectural constraints. Their impact on architectural quality encompasses properties concerning the quality of the software system under construction as well as the design process, including its efficiency and its perspectives.

We focus on *codified* constraints [27] on software architecture descriptions, i.e. such constraints which have some explicit description separable from the software architecture of a specific system under consideration. Additionally, we require that a constraint does not exist in isolation but conforms to some generic concept. A generic concept allows the specification of alternative constraints, which are *commensurable* with each other. For example, every software architecture conforms to some style. Therefore, since every software system has an architecture, every software system has a style; and styles must have existed since the first software system was developed. However, without a codification process, developers of other systems cannot efficiently exploit the properties of a style. For example, the UNIX operating system introduced a variant of the pipe-and-filter style for the runtime architecture of processes, which was later documented to be used by system developers.

Codification of constraints involves a choice in the degree of formal rigour, and the mode of inclusion of the context. These choices determine the ability to reason about properties of different constraints when used as the basis for architecting a software system. Apart from serving as documentation for a constraint imposed by a chosen implementation platform (operating system, middleware, virtual machine, etc.) and ensuring interoperability, deliberate decisions for a constraint may be made. This goal imposes additional requirements on the representation of constraints. Architectural constraints can be viewed as capturing many fine-grained design decisions, so their role in a design process may be as follows: instead of considering a large number of fine-grained design decisions which generate a huge design space, in which many points may be excluded due to conflicts of the corresponding alternatives of different decisions, a single decision is made between a smaller number of well-known architectural constraints. Such a process requires a certain degree of commensurability of the alternative constraints, while it is not always necessary to have a formal-mathematical specification of the constraints available. This single decision may be made more economically in a more sophisticated way than the many fine-grained decisions it substitutes.

Formalisation of the solution's content and representation of the problem context are two complementary aspects of architectural constraints (see also Riehle and Züllighoven [27]). While the syntactical and semantic correctness is addressed by the formalisation of the content, for an efficient and adequate use of the constraint, understanding the pragmatics is equally important. In fact, a trade-off between understandability and the degree of formalisation of the content may be required.

When an architectural constraint is selected, it restricts the design space that needs to be further considered by capturing a set of related architectural design decisions. Additionally they pre-structure the decision process of dependent decisions, which may be regarded as options enabled by the decision for a constraint. While these restrictions are useful for improving the design process in general, they particularly help less experienced designers in reusing approved design knowledge for architecting high-quality software systems.

Knowledge on properties of single architectural constraints is not sufficient to be helpful in an architecture-based process of software design. To make a decision on which constraint(s) to use for a certain software system, a catalogue of architectural constraints must be available. Different architectural constraints address different architectural concerns. Therefore, such a catalogue should provide information on the relationships between constraints and concerns. A catalogue of architectural constraints should not be built entirely from scratch but should incorporate the existing body of knowledge.

1.2. Characteristics of architectural constraints

The concepts underlying codified architectural constraints vary. These concepts may be clustered, e.g., by distinguishing the properties of their instances: the *scope* is distinguished into *global* or *local*, and their *level of obligation* into *obligatory* or *tentative*. For example, global and obligatory constraints are often referred to as ''architectural styles''. In order to effectively establish a catalogue, constraints conforming to the same cluster of concepts should be represented within a single meta-model to ensure their commensurability. A thorough understanding of the relationships among the concepts within each cluster is a prerequisite for establishing a useful meta-model, which does not prevail yet. Furthermore, an understanding of the relationships between different clusters is important for enabling architectural decisions. Our work provides a significant contribution to this goal.

1.3. Research context

The work presented here is embedded in a research project that develops and validates the MIDARCH method for migrating software systems which exploits a special kind of architectural styles [15]. We understand architectural styles in the way that is elaborated in this paper and see the need to distinguish them from architectural patterns in a precise way. The special type of architectural styles we consider are *MINT Styles* (Middleware INTegration Styles), which are architectural styles induced or endorsed by middleware platforms. The study reported on in this paper is accompanied by a study creating a taxonomy of the different usage types of architectural styles as proposed in the literature. These two taxonomies provide the conceptual foundation for the use of styles within the MIDARCH method. The MIDARCH method improves the reuse of architectural design knowledge in

software migration projects. Candidate target architectures are based on explicitly defined MINT Styles, instead of an ad-hoc development of candidate architectures. The knowledge gained in one project by evaluating candidate architectures can be linked to appropriate architectural styles. By making this kind of architectural design knowledge more tangible, the quality of the developed software is improved in the end as well.

1.4. Overview

The remainder of the paper is structured as follows: in Section 2, fundamental terms are introduced and the employed research method is described. Section 3 discusses the relation of architectural constraints and architectural quality. Afterwards the general definition of an ''architectural constraint concept'' is introduced in Section 4. Software architecture concepts are discussed in Section 5 and the application to engineering processes and artefacts in general in Section 6. Finally, related work is discussed in Section 7. The paper concludes with Section 8 and indicates future work in Section 9.

2. Foundations

First, our view of software architecture is briefly introduced in Section 2.1, since still no common view of software architecture has established. A central notion within this study is that of a (architectural constraint) concept. The notion of a ''concept'' and related notions are explained in Section 2.2. The specific research method used is presented in Section 2.3.

The object of this study are research results from research on Software Architecture. In this sense, our study is a meta-study. It is empirical in the sense that it is based upon research results that are assumed to have been empirically validated, but theoretical in the sense that its propositions are obtained deductively. Our research also has a creative aspect [21].

2.1. Software architecture

The literature on architectural constraint concepts refers to a variety of definitions of software architecture. Several of these definitions may be viewed as mutually incompatible, and concrete architectures modelled with reference to diverse definitions of software architecture will probably be incommensurable. Therefore, we decided to select the definition of software architecture in the IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [16,20] as an approved reference definition:

> Software Architecture: ''The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution''. [16]

Architectural decisions are reflected in this definition through the reference to underlying ''principles'' which capture architectural decisions. What is meant by ''principles'' comes close to our understanding of architectural constraints, while we additionally require that constraints have an explicit representation. Within the decomposition of the conceptual model defined in the standard (see Fig. 1), such principles are found in the rationale of an architectural description.

Two distinctions made in the conceptual model are highly relevant here: the (software) system is distinct from the architecture. Additionally, the architecture of a system is distinguished from its architectural description. Every system has an architecture, but it is not necessarily documented explicitly in an architectural description. If there is an architectural description, it may be incomplete, outdated or simply incorrect with respect to the actual architecture.

The architecture itself is intangible, and may only be modified by changing the system. Modifications to the architectural description may be regarded as a mandate to adapting the system to reestablish consistency of architectural description and the actual architecture, and vice versa.

A *viewpoint* represents related concerns that are determined by stakeholder interests. A viewpoint also determines modelling techniques and notations to use in views that correspond to the viewpoint. A *view* describes a (concrete) system from a certain (abstract) viewpoint. Views structure the overall *architectural description*. The actual information is contained in *models* that are related to views in a *m:n* relationship: A view refers to information that may be distributed over multiple models, and a model may contain information for multiple views.

2.2. Fundamental terms

A *Concept*, as defined in ISO standard 1087 [17], is a cognitive construct that represents the properties of a group of individual objects (note that ''Object'' is to be understood in a general sense here, and may refer to intangible, abstract objects such as elements of software architecture descriptions). Employing a stricter notion than that of general terminology theory, we use the term ''concept'' as a shorthand for ''architectural constraints concept'' in the remainder of the present paper. A *Concept instance* then is a particular architectural constraint, e.g., one specific architectural style. Every *Concept* and every *Concept instance* is denominated by a *Term*. *Terms* are distinguished here into *Generic terms* and *Names*.

For example, ''architectural style'' is a generic term which denotes a concept, and ''pipe-and-filter style'' is a name of an instance of that concept.

A *concept* is never conceived in isolation, but in relation to other *concepts*, therefore forming a *system of concepts*; the relationships within such a system may be separated into explicit and implicit relationships. Previously, architectural constraint concepts were only implicitly related within a system of concepts that structures the universe of discourse

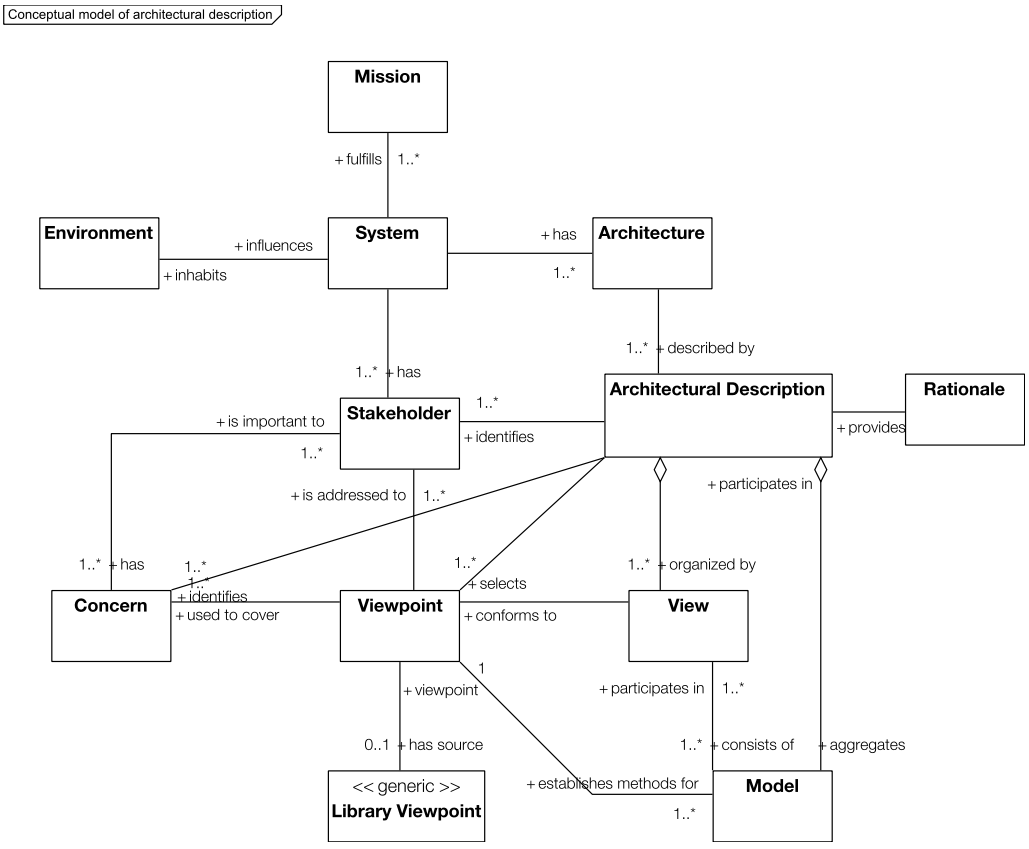

Fig. 1. Conceptual framework for architectural description [16].

of software architecture. The goal of our research is to define explicit relationships for these concepts.

2.3. Research method

To start with, we explicitly cover characteristics of architectural constraint concepts in a general definition of an "architectural constraint concept" consisting of several elements that we synthesised from our implicit knowledge of architectural constraint concepts. Then, we check for each individual architectural constraint concept whether it matches the general definition. In this case, we describe it as a specialisation of the general definition, i.e. we describe where each element of the definition is specialised and where additional definition elements must be added. If it does not match the general definition, the respective concept is not considered an architectural constraint concept.

This approach is non-trivial due to the fact that for most individual architectural constraint concepts, no explicit definition is given, and even if one is given, it does not relate directly to the structure of our definition. Consequently, it may not even be possible to determine in all cases whether the general definition fits at all, since the characterisation given by the respective authors is too fuzzy. However, this problem is part of the motivation for conducting this study, and this study will help in solving the problem, even if it cannot resolve all ambiguities.

As a result of our study, relationships between the examined architectural constraint concepts established by their respective contribution to the characterisation within the elements of the general definition are obtained. Based on these results, the adequacy of the original general definition can be determined, and the definition may be revised.

Additionally, a classification with respect to different properties of the instances is imposed on the constraint concepts:

Level of rigour. Architectural descriptions may exhibit different levels of formality, ranging from *unstructured* or *structured* natural-language descriptions and *informal* box-and-line diagrams, over *semi-formal* diagrams, such as the UML, to *formal* notations, which may even provide executable specifications. Each of these notations may be adequate for its intended purpose.

Level of obligation. Architectural constraints may differ in the level of obligation they demand from the affected software artifacts. This may range from *suggestions* (loose) over *recommendations* to *restrictions* (strict). Whether it is possible to check/enforce them or provide tool support for the semi-automatic generation of compliant architectural descriptions, depends both on the level of rigour of the constraint and of the architectural description.

The obligation does not extend to revisions of an architectural decision, i.e. we assume that—regardless of the

level of obligation—any decision for a constraint may later be revoked, which also removes the obligation.
Scope. The scope of the constraint can be local or global within the architectural description it is applied to. Local constraints constrain only an explicitly specified set of description elements, while global constraints apply to all current and future elements, possibly restricted by some application condition. The characterisation depends on the regarded system scope: any constraint imposed globally on a subsystem may be regarded as local to the subsystem in a system-level view (see also Gamma et al. [9, p. 3]). In any case, the scope is expressed relative to the *system under review*.

3. Architectural quality and quality assessment

The use of architectural constraints in architectural design is an important contribution to achieving architectural quality [7]. Adequate properties of the software architecture enable the resulting software system to meet the requirements on behaviour, quality and life cycle [4]. The use of architectural constraints is meant to contribute to all these aspects of architectural quality.

The support for architectural quality in the development life cycle of a software system is central to this contribution. During the analysis and design phase, a software architecture helps to make appropriate decisions and to reduce risks related to a specific set of requirements. Architectural constraints enhance these effects by providing well-established solutions. Furthermore, they reduce the complexity of a solution by improving the *conceptual integrity* and *consistency* of an architecture [5, p. 95]. Concerning validation and verification, architectural quality encompasses both the internal consistency of the architectural description *(architectural consistency)* and the external compliance of a system's implementation with its architectural description *(architectural compliance)*. The use of architectural constraints contributes to these aspects of architectural quality as well. It reduces the effort necessary to create and maintain an architectural description, and to maintain architectural compliance. Architectural compliance is attained through *traceability*. Traceability includes *requirements traceability*, i.e. the possibility to relate software artefacts (at any level) to the requirements they contribute to, as well as *architectural traceability*, i.e. the possibility to relate implementation artefacts to architectural artefacts. A proper level of architectural compliance is a prerequisite for useful architecture-based predictions of *system quality* attributes [18], concerning the internal and external software quality. Tool support for these activities is desirable.

3.1. Architectural constraints in software design processes

Additionally, architectural constraints have a strong impact on the decision-making process: they help to reduce the number of alternatives for design decisions and therefore, they improve the efficiency of the design process.

First, they ease design decisions. For a single decision, architectural constraint concepts reduce the set of available alternatives by excluding the ones that are not applicable. In this way, they reduce the complexity of the decision and thus, they ease the work of the developer. Furthermore, architectural constraints introduce well-established solutions. Such a solution has already proved its trade-off; its benefits and properties are known. The particular design decision can be based on an analysis; it can be performed methodically. In this way, the architectural constraint helps to reduce redundant revision work and to improve the efficiency of the design process.

Second, by repeatedly considering (selecting some and rejecting others) architectural constraints in different projects, the body of knowledge associated with these constraints grows, and knowledge on the relationship between constraints and the quality characteristics of the resulting software systems can be collected. Software design methods such as the MIDARCH method [15] explicitly exploit this aspect of architectural constraints to improve reuse of architectural design knowledge.

Third, the efficiency of the communication between the stakeholders is improved by the usage of architectural constraints as parts of a solution. Their application leads to an increased degree of formality and to an improved degree of conceptual integrity of the design description. Thus, the communication effort as well as the probability of misunderstandings is reduced.

4. Architectural constraint concepts

We start with the following definition of an *architectural constraint concept*:

Definition 1 (*Architectural constraint concept*). An *architectural constraint concept* is a concept, which refers to the universe of discourse of software architecture, representing instances (called *architectural constraints*) which

(1) refer to a meta-model for architectural models that is defined by some viewpoint,
(2) contribute to the architectural rationale by capturing a set of meaningful design decisions,
(3) should define a vocabulary for constituents of an architectural model,
(4) do not apply only to isolated constituents of a model, and
(5) should have an explicit representation.

The first two criteria establish the link to the conceptual framework of software architecture as defined by the IEEE 1471-2000 standard: An architectural constraint is part of the architectural rationale and refers to a meta-model, which are defined—by convention of the standard—within

a viewpoint. The instances of such a meta-model are architectural models. The intention is to enable reuse of architectural constraints for multiple systems, i.e. an architectural constraint may be imported into the rationale of a specific software architecture description from some catalogue of reusable architectural constraints.

The first criterion particularly excludes constraints that define a meta-model themselves (e.g., "architectural design styles" [28]). The second criterion excludes purely syntactical or semantic constructs. The third criterion is particularly important for aspects concerning the use of architectural constraints for stakeholder communication. The fourth criterion excludes concepts such as component and connector types (see, e.g., Allen and Garlan [3]). These may be defined by or referred to by an architectural constraint, but do not form an architectural constraint as such.

The fifth criterion excludes architectural constraints that are merely implied or assumed by specific approaches to architectural description, by specific component and connector types, etc. Nonetheless we regard the analysis of such implicit assumptions with the goal of representing them explicitly worthwhile, but outside the scope of our current work. Analogous to the distinction of architecture and architectural description in the IEEE Standard 1471, we distinguish a *constraint* and its *constraint description*.

5. Taxonomy of architectural constraints

Important specialised architectural constraint concepts defined in the Software Architecture literature will be characterised as specialisations of the general definition given in Section 4.

Architectural constraints can be formalised to a varying extent, which is not completely determined by the concept definitions as discussed in the following, i.e. the instances of each concept may be described in a more formal or a less formal way. The reason for this variance is that we do not discuss the architectural constraint concepts at a technical, but at a conceptual level; we do not explicitly consider the language in which constraints should be expressed. Still, some concepts' instances are more apt to formalisation than others.

However, as Riehle and Züllighoven [27] emphasise concerning "patterns", the "form" describing the pattern can be formalised, but its "context" may not. This statement applies to architectural constraints in general. In our terms, a constraint description comprises both form and context, and can thus not be completely formalised.

In the following, first pattern-based concepts are discussed, followed by style-based concepts.

5.1. Pattern-based concepts

5.1.1. "Gang of Four" patterns

The idea of "design patterns" as expressed by the "Gang of Four" [9, Chapter 1.1] is close to Alexander's idea of a pattern [2]. A pattern is a "solution to a problem in a context" [9, p. 3], and consists of a name and the description of the problem, solution and consequences.

Although Gamma et al. are not concerned with architecture-level patterns, but with "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [9, p. 3], many of their patterns can also be interpreted from an architectural point of view, e.g., Composite, Mediator, or Observer. The presentation in Gamma et al. [9] provides code examples and relies on certain properties of the object-oriented programming languages used (C++ and Smalltalk). On the implementation-level it is a prevalent question which language features are available and *how* they are used. When regarding the same patterns from an architectural point of view, only the question whether the design pattern may be implemented using some implementation technology (a question to which the answer will in virtually any case be yes due to the inherent flexibility of software) and how far the effort of implementation is. The nature of the architectural perspective essentially is the abstraction from the details of the mapping of architectural artifacts to implementation artifacts.

With respect to the general architectural constraint concept definition, we can infer that a design pattern from an architectural perspective

(1) refers to some meta-model, specifically an object-oriented class or object hierarchy,
(2) contributes to the architectural rationale, which is embodied both in its name and the problem description,
(3) defines a vocabulary in the form of a solution "template" describing "the elements that make up the design, their relationships, responsibilities, and collaborations",
(4) does therefore apply to multiple elements of a model,
(5) has an explicit representation, which is organised in a structured pattern description.

As a consequence, we conclude that design patterns in their architectural interpretation can be regarded as an architectural constraint concept.

Classification: The patterns described by Gamma et al. are clearly local in scope, since they specify an explicitly enumerated set of design elements.

The level of rigour is semi-formal, since natural language combined with OMT diagrams (essentially the precursor of UML class and object diagrams) is used for documentation. The level of obligation is only moderate because of two aspects: First, many variations are allowed for each pattern which are not exhaustively enumerated in the pattern description. Second, a pattern implementation is often intermixed with other implementation fragments. However, an obligation is expected to follow the commitment to a pattern.

5.1.2. Buschmann et al. patterns

Buschmann et al. [6] define the concept of *architectural patterns* as their central architectural constraint concept:

> An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. [6, section 1.3]

By now, there exists a series of books describing patterns in this sense. In the original book, [6] consider patterns such as Broker, Model-View-Controller and Presentation-Abstraction-Control.

With respect to the general architectural constraint concept definition, we can infer:

(1) an architectural pattern refers to some meta-model, which is not specified explicitly (''expresses a fundamental structural organisation schema''),
(2) contributes to the architectural rationale implicitly, in that it represents codified architectural design knowledge,
(3) defines a vocabulary (''predefined subsystems'', ''their responsibilities'', ''rules and guidelines''),
(4) usually applies to multiple elements of a model (''a set of . . .''),
(5) has an explicit representation (which is what the book Buschmann et al. [6] is about).

Buschmann et al. also recognise the prevalence of a different architectural constraint concept, *architectural styles*. However, they refer to colloquial uses of the term, and thus the distinction remains fuzzy. However, they assume that architectural styles are a special case of architecture patterns, but ''every architectural style can be described as an architectural pattern'' [6, Section 6.2]. They allege that styles lack interdependencies with other styles and that the viewpoint of architectural styles is not problem-oriented, but ''independent of an actual design situation''.

Classification: Due to the fact that Buschmann et al. subsume styles under their pattern concept, some of their patterns apply to a global scope, but essentially Buschmann patterns are local in scope since they make no statement about the system outside the scope targeted by the explicit elements of the pattern. Still, Buschmann patterns are more global than the Gang-of-Four patterns since Gamma et al. [9, p. 3] explicitly exclude patterns that determine the overall architectural organisation of a software system.

Since the POSA series of books provides a large number of instances, statements regarding the typical level of rigour and obligation can be made with a certain confidence. The level of rigour that is employed is semi-formal throughout the presented patterns. The level of obligation can only be described on a less formal basis: On the one hand, the coarse structure of a pattern should be followed, but usually many variations to a pattern exist, and, in practise, even after choosing a variant of a pattern, deviations may be made in implementing a pattern as long as the general idea of the pattern is still obeyed.

5.2. Style-based concepts

5.2.1. Perry/Wolf styles

The seminal paper by Perry and Wolf [25] already mentions ''style'' as one important justification for introducing the ''architecture'' metaphor. Besides the use of multiple views, style is also used to create the major analogies to building architecture: These are related to the descriptive and prescriptive effects of architectural style. The interactions of the style with engineering principles and material properties are transferred to software architecture. As examples, they name a ''distributed'' or a ''multi-process'' style.

Perry and Wolf assume a *uniformity of architecture and architectural style*: ''We have a continuum in which one person's architecture may be another's architectural style'' [25, Section 3.2]. This is due to the fact that they define both concepts only vaguely as constraints for the lower-level realisation of a system.

Due to the vague definition, ''style'' is seen primarily as an organisational rather than a technical instrument: ''An emphasis on style as a constraint on the architecture provides a visibility of those aspects and insensitivity to them will be more obvious.'' As a consequence, we do not consider Perry/Wolf's ''architectural style'' as an architectural constraint concept. However, their work significantly inspired several more rigorous architectural constraint concepts, such as the SEI styles (see Section 5.2.2).

5.2.2. SEI styles (Allen, Garlan)

Allen and Garlan define architectural styles in several papers, notably in Abowd et al. [1] and in Garlan [12]. The underlying ideas were already indicated in Garlan and Shaw [11]. Examples of the structuring principles they describe as styles include Pipes and Filters, Data Abstraction, Event-based, Layered and other rather generic architectural styles.

Garlan [12] discusses several different approaches to the definition and use of architectural styles, but assumes several common properties of any view of *architectural style*:

(a) The provision of a vocabulary of design elements, which are component and connector types.
(b) The definition of a set of configuration rules.
(c) The definition of a semantic interpretation, which gives some well-defined meaning to all configurations of design elements that satisfy the configuration rules.
(d) The definition of analyses for configurations of that style. Examples include schedulability analysis, deadlock analysis, code generation, and conformance checking.

With respect to the general architectural constraint concept definition, we can infer that an architectural style:

(1) refers to some meta-model, in which the vocabulary is defined (see (a)),
(2) contributes to the architectural rationale in that it represents codified architectural design knowledge, which is given by the semantic interpretation (see (c)),
(3) defines a vocabulary (see (a)),
(4) applies to multiple elements of a model, which is implicitly expressed in the idea of a configuration (see (b)),
(5) has an explicit representation, which is implicitly assumed by the definition.

The idea of analyses, which are style-specific and can be applied to configurations (allegedly) conforming to a style, is an issue that is notable in addition to the items contained in the general definition. This idea builds upon the assumption that a style is applied globally to the system under review.

Classification: The scope of the SEI styles is clearly global, the authors refer explicitly to whole architectures that are expected to follow a style.

Based on the SEI approach, varying degrees of rigour are conceivable and have been proposed. Sometime, they employ a rigorous formal approach in Abowd et al. [1]. In the Aesop system [10], which is used to generate style-based architectural design environments, a less formal approach is used. Here the style is not explicitly specified, but implicitly implemented in code extensions to the Aesop system. The expected level of commitment is high in both cases.

5.2.3. Medvidovic/Taylor styles

While Taylor et al. [31] only describe a single style, the Chiron-2 or C2 style, an underlying generic concept may be inferred from their description, which, in principle, could be used to specify other styles as well.

With respect to the general architectural constraint concept definition, we can infer that an architectural style

(1) refers to some meta-model: the authors explicitly refer to a decomposition of a system into components and connectors,
(2) contributes to the architectural rationale (they are "key design idioms", "provide a rationale for the desired properties of the components and connectors, as well as for the choice of principles" [31]),
(3) defines a vocabulary: C2-specific components and connectors are proposed,
(4) does apply to multiple elements of a model ("a network of concurrent components", "configuration" [31]),
(5) has an explicit representation (which is, in the case of the C2 style, published in Medvidovic [22]).

Thus, Medvidovic and Taylor's styles can be considered an architectural constraint concept.

Classification: The scope of a Medvidovic/Taylor style is clearly global. While they explicitly recognise that it might be applied in the UI subsystem of a larger system only, this subsystem constitutes the system under review.

The example of the architectural style concept provided by Medvidovic [22] employs the Z notation for specifying the C2 style, thus a rigorous formal mathematical level of rigour is employed. The level of obligation to the style is also very high, as their primary concern is the achievement of technical interoperability, which does not allow for deviations from specified properties.

5.3. Summary

Throughout the software engineering literature, the terms "style" and "pattern" are used in various contexts, in most cases in a purely informal manner. Many uses come close to architectural constraints, other uses do not. It is probably infeasible to analyse all uses of these terms in the way we did before, and we are sceptical whether such an analysis would lead to interesting results. However, there are certainly several more elaborated and more rigorous constraint concepts published, which could be included in the future. So far, we focused on the most "popular" concepts.

These concepts can be organised into a taxonomy in various ways. Essentially, a top-down and a bottom-up approach can be applied. Our *top-down* approach is based on the two classification dimensions scope and level of rigour.

To summarise: Buschmann's and the Gang of Four's Patterns and the SEI and Medvidovic/Taylor Styles can be considered architectural constraint concepts in the sense of our general definition. Not included are the Perry/Wolf Styles, which we consider too vague to be architectural constraints.

The concepts of patterns and styles differ in the type of *instantiation* of its instances within a software system. The choice of a pattern already involves the enumeration of the participating elements. Even if the pattern specification allows variants, the variation is resolved as part of the choice. Tool support for instantiation of the pattern then is concerned with the mapping to implementation-level artifacts (i.e. code generation). The choice of a style, on the other hand, merely prepares the design of the architecture itself. As part of the choice, not the concrete elements are chosen, but only their types are chosen or constrained. To give an example for this difference: when choosing the Singleton pattern [9], the choice of this pattern and the identification of the class which should be made a singleton cannot be separated but constitute a single atomic design decision. However, for a style, patterns may be defined that can be instantiated once the style has been selected (see below).

Additionally, instantiation is related to different *design activities*: For pattern-based concepts, instantiation is targeted at the implementation process, while for style-based

concepts the focus is on guiding the further design process. As a consequence, requirements on tool support for applying patterns and styles differ.

Tools should be able to guide the further selection of concrete elements based on the selected style. Concurrently to architectural design, a tool should be able to check whether the architecture conforms to the selected style. The architecture of a system cannot automatically be generated from the style alone.

To establish a technical relationship of styles and patterns, *style-specific patterns* can be defined, similarly to domain-specific patterns that have been discussed by many authors. In fact, many so-called domain-specific patterns (such as Java EE patterns) are not specific to an *application* domain, but to the style imposed by some implementation technology (e.g., the Java EE platform), which determines a *technical* domain. Then, a style contains or refers to a set of patterns.

6. Application to engineering processes and artefacts in general

For any large, complex system, the design of the overall system structure (the architecture) is a central problem. This is a recurring problem in any engineering discipline. The *architecture* of a system defines that system in terms of components and connections among those components. It is not the *design* of that system which is more detailed. The architecture shows the correspondence between the requirements and the constructed system, thereby providing some rationale for the design decisions.

In the civil engineering discipline, the engineer knows that a house includes a roof at the top and a cellar at the bottom, etc. Similarly, using the example of compiler construction, the software engineer knows that a compiler contains a pipeline including lexical analysis, parsing, semantic analysis, and code generation. Up to now, this is not the case for all areas of software and systems development: e.g., the discipline of software architecture is still an emerging discipline. For building architecture, the idea of encoding design knowledge in a pattern form has been devised by Alexander [2]. The transfer to computer-based systems is discussed in the following Section 6.1. A case study for the application of architectural styles in software system engineering is then presented in Section 6.2.

6.1. Patterns in computer-based engineering systems

In this contribution, architectural styles and patterns are discussed in a software engineering context. They are applied in a very similar way in many other engineering processes, and they act as architectural constraint concepts there by influencing design decisions. As an illustration, the architectural style Layer—as used in software architectures—is compared to similar styles in the design of automatic control solutions for large systems and in network design for building automation systems.

Automatic control of large systems is usually structured in a hierarchical way. For managing the complexity of large systems and for simplifying their models, systems are decomposed to modules or units. Hierarchical structures occur not only for system models but for control structures and control tasks as well. This way of structuring has been established as a basic style by Ogata [24]. Furthermore, hierarchies can represent different aspects [33].

Hierarchical layering is widely accepted as a typical style in the network design of building automation systems. By decomposing network parts into levels, different requirements in terms of reliability, time behaviour and performance can be fulfilled. As an example, an emergency notification system as part of a building automation system usually consists of sub-networks at different levels to manage alarms and failures effectively [23].

If we compare these examples to the field of software engineering, a uniform influence on the resulting design can be recognised: there are hierarchically structured units with specific responsibilities. The architectural style Layer is applied in the different engineering disciplines in a very similar way, since identical engineering principles apply: abstraction, encapsulation and modularisation. The uniform role of the Layer style for the engineering decisions in the different engineering disciplines is caused by similar problem solving strategies.

6.2. Styles in a software system engineering process

As an example on how architectural styles can be used in a Software System Engineering Process, we briefly sketch a case study that we conducted [14]. The goal of the case study was to evaluate a generic method for conducting migration and integration of information systems (MIDARCH Method). The method is tailored towards so-called Middleware-intensive systems, i.e. complex systems whose structure is significantly influenced by the underlying Middleware platform. The features embodying this influence form architectural constraints, which are a special kind of the architectural styles discussed in Section 5.2.2, which we refer to as MINT Styles (Migration and Integration Styles).

The subject system in the case study was a web-based regional trade information system (REGIS-Online) developed by regio Institut GmbH, Oldenburg. Prior to the migration, the architecture of the system consisted of two distinct subsystems for querying and updating data, which both used the Apache Cocoon middleware, however in incoherent and differing ways. The goal of the migration was to create a more coherent architecture which eases future maintenance of the system.

Fig. 2 shows the activities involved in the migration process, and the dependencies between the activities. The focus with respect to architectural styles in the form of MINT Styles is in the third column. MINT Styles are modelled for two variants of using Apache Cocoon. Based on these MINT Style descriptions, target architectures are modelled

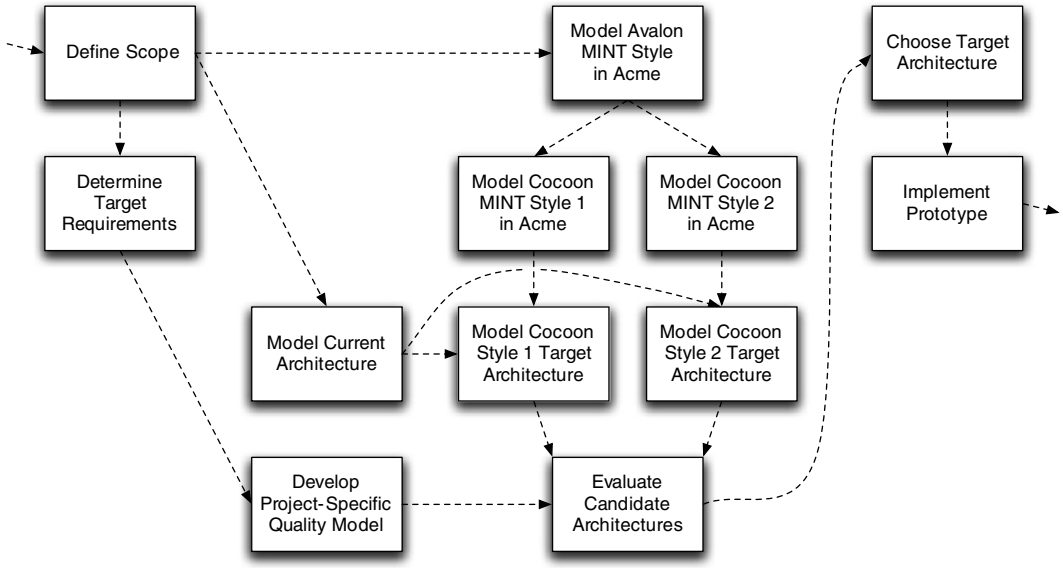

Fig. 2. Case study: migration process.

and evaluated. Since there is an explicit link between the architecture and the style (the architecture is an instance of the style), the results of the architectural evaluation can also be linked to the MINT Style and this design knowledge may be reused in further migration and integration projects that apply the MIDARCH method.

7. Related work

Related work includes Garzas and Piattini [32,13]. Both approaches aim at capturing design knowledge, but their approaches and the scope are different: they refer to more implementation-oriented, low-level design knowledge, while we refer to higher-level architectural design knowledge. They relate different contexts to each other by implicitly considering their characteristics, but we explicitly capture these characteristics within a basic definition.

Szyperski [30, Chapter 9] discusses "design-level reuse" based on artifacts such as programme code, libraries, interfaces, protocols, patterns, frameworks, and system architectures. These artifacts are grossly classified with respect to programming-in-the-large and programming-in-the-small. We did not include this dimension into our taxo-nomy, since we only take the perspective of software architecture, which belongs to programming-in-the-large. However, several of the artifacts considered by Szyperski also play a role in our taxonomy. Szyperski places (design) patterns between message protocols and frameworks. In fact, what Szyperski describes as frameworks comes close to the style concepts discussed above when considering their relationship to patterns: "A framework integrates and concretizes a number of patterns to the degree required to ensure proper interleaving and interaction of the various patterns' participants. Indeed, a framework can be explained in terms of the patterns it uses." However, the focus set by Szyperski tends to neglect an important difference between styles and frameworks: frameworks are an implementation-level artifact and consist of binary or source code, while styles are architectural-level artifacts. However, a style can be inferred from a framework, and a framework can be developed to support the realisation of a style.

8. Conclusion

In this paper, we analysed several software architecture concepts and put them into the context of our definition of architectural constraint concepts. We provide a taxonomy of the architectural constraint concepts. The taxonomy is based on the ANSI/IEEE Standard 1471 (IEEE Recommended Practice for Architectural Description of Software-Intensive Systems). While much of the content of architectural constraints will be found in the architectural rationale, a refinement of the elements defined in the standard would allow a better mapping of the elements of architectural constraints. For the considered software architecture concepts, the working definition has proved adequate.

The approach may be applied to additional architectural constraint concepts. However, some evaluation of their significance has to be applied, otherwise it is possible that the general definition is becoming too general to be meaningful.

As emphasised in the introduction, a well-founded understanding of the differences between architectural constraint concepts is necessary to rigorously define individual constraints and provide methods for analysing constraints and exploiting them in order to improve the design of high-quality software systems. We will exploit the insights gained in the study presented here, together with an accompanying study on different usages of architectural styles, to develop a method for selecting appropriate middleware platforms

in Enterprise Application Integration projects based on a taxonomy of the styles endorsed by available middleware platforms.

9. Future work

Formal ontologies are used for modelling the universe of discourse of information systems. In future work, our taxonomy of architectural constraint concepts could evolve into a formal ontology (e.g., in the OWL). So far, the focus of our work is not targeted towards automation of the processes around architectural constraints. While tool support for decision processes with respect to architectural constraints is an important goal of our work, it is not yet clear whether the techniques for ontological engineering are helpful in this context.

As a certain degree of commensurability is established by the taxonomy, in a next step the taxonomy of the individual instances of each concept cluster may be refined. On the level of design patterns, in some cases the question of equivalence of design patterns formulated in different ways has proved to be difficult to be resolved, e.g., the controversy about the equivalence of the Multicast and Observer patterns, which could only be resolved by using the LePuS formalism [8].

So far, we focused on constraint concepts that are found in the literature and that have been defined and used in practise. It would be interesting to explore additional constraint concepts that allow new applications. This could include, for example, constraints governing multiple models while ensuring their consistency.

References

[1] G.D. Abowd, R. Allen, D. Garlan, Formalizing style to understand descriptions of software architecture, ACM Transactions on Software Engineering and Methodology 4 (4) (1995) 319–364.
[2] C. Alexander, S. Ishikawa, M. Silverstein, A pattern language : towns, buildings, constructionCenter for Environmental Structure, vol. 2, Oxford University Press, New York, 1977.
[3] R. Allen, D. Garlan, A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology 6 (3) (1997) 213–249.
[4] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003.
[5] F.P. Brooks, The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary ed., Addison-Wesley, 1995.
[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.
[7] J.C. Dueñas, W.L. de Oliveira, J.A. de la Puente, A software architecture evaluation model, in: Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, Springer-Verlag, London, UK, 1998, pp. 148–157.
[8] A.H. Eden, Y. Hirshfeld, A. Yehudai. Multicast – observer ≠ typed message. C++ Report 10 (9), 1998.
[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., 1995.
[10] D. Garlan, R. Allen, J. Ockerbloom, Exploiting style in architectural design environments, in: Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, 1994, pp. 175–188.
[11] D. Garlan, M. Shaw, An introduction to software architecture, in: V. Ambriola, G. Tortora (Eds.), Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, Singapore, 1993, pp. 1–39.
[12] D. Garlan, What is style? in: D. Garlan (Ed.), Software Architectures, Dagstuhl-Seminar-Report, Proceedings of the Dagstuhl Workshop on Software Architecture, Saarbrücken, Germany, vol. 106, February 1995.
[13] J. Garzas, M. Piattini, An ontology for microarchitectural design knowledge, IEEE Software 22 (2) (2005) 28–33.
[14] S. Giesecke, J. Bornhold, Style-based architectural analysis for migrating a web-based regional trade information system, in: Trentini, A., Marchetto, A., Bellettini, C. (Eds.), First International Workshop on Web Maintenance and Reengineering (WMR 2006) in conj. with CSMR 2006, Bari, Italy, CEUR Workshop Proceedings, vol. 193, 2006, pp. 15–23.
[15] S. Giesecke, Middleware-induced styles for enterprise application integration, in: Proc. 10th European Conference on Software Maintenance and Reengineering (CSMR06), IEEE Comp. Soc., 2006, pp. 334–340.
[16] IEEE, Recommended Practice for Architectural Description of Software-Intensive Systems, ANSI/IEEE Standard 1471-2000, 2000.
[17] ISO, Terminology work – Vocabulary – Part 1: Theory and Application, ISO Standard 1087-1:2000, 2000.
[18] H.-W. Jung, S.-G. Kim, C.-S. Chung, Measuring software product quality: a survey of ISO/IEC 9126, IEEE Software 21 (5) (2004) 88–92.
[19] M.S. Mahoney, Finding a history for software engineering, IEEE Annals of the History of Computing 26 (1) (2004) 8–19.
[20] M.W. Maier, D. Emery, R. Hilliard, Software architecture: introducing IEEE Standard 1471, Computer 34 (4) (2001) 107–109.
[21] E. Marcos, Software engineering research versus software development, SIGSOFT Software Engineering Notes 30 (4) (2005) 1–7.
[22] N. Medvidovic, Formal definition of the Chiron-2 architectural style, Tech. Rep. UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, November 1995.
[23] G. Neugschwandtner, W. Kastner, B. Erb, Fire safety alarm transmission in networked building automation systems, in: 6th IEEE Intl. Workshop on Factory Communication Systems (WFCS'06), IEEE, 2006, pp. 79–82.
[24] K. Ogata, Modern Control Engineering, fourth ed., Prentice-Hall, 2002.
[25] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.
[26] B. Randell, J. Buxton (Eds.), Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969, Brussels, NATO Scientific Affairs Division, Brussels, 1970.
[27] D. Riehle, H. Züllighoven, Understanding and using patterns in software development, Theory and Practice of Object Systems 2 (1) (1996) 3–13.
[28] M. Shaw, Comparing architectural design styles, IEEE Software 12 (6) (1995) 27–41.
[29] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, Inc., 1996.
[30] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press/Addison-Wesley Publishing Co., 1998.
[31] R.N. Taylor, N. Medvidovic, K.M. Anderson, E. James, J. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, D.L. Dubrow, A component- and message-based architectural style for GUI software, IEEE Transactions on Software Engineering 22 (6) (1996) 390–406.
[32] W.M. Tepfenhart, J.J. Cusick, A unified object topology, IEEE Software 14 (1) (1997) 31–35.
[33] P. Varaiya, A question about hierarchical systems, in: T.E. Djaferis, I.C. Schick (Eds.), System Theory: Modeling, Analysis and Control, The International Series in Engineering and Computer Science, vol. 518, Kluwer, 1999 (Chapter 23).