Abstract

Despite several (accepted) standards, core notions typically employed in information technology or system engineering architectures lack the precise and exact foundations encountered in logic, algebra, and other branches of mathematics.

In this contribution we define the syntactical aspects of the term architecture in a mathematically rigorous way. We motivate our particular choice by demonstrating (i) how commonly understood and expected properties of an architecture—as defined by various standards—can be suitably identified or derived within our formalization, (ii) how our concept is fully compatible with real life (business) architectures, and (iii) how our definition complements recent foundational work in this area [12,15,38].

We furthermore develop a rigorous notion of architectural similarity based on the notion of homomorphisms allowing the class of architectures to be regarded as a category, Arch. We demonstrate the applicability of our concepts to theory by deriving theorems on the classification of certain types of architectures. Inter alia, we derive a no go theorem proving that, in contrast to n-tier architectures, one cannot sensibly define generic architectural modularity on the syntactical level alone.
1 Introduction

Information technology architecture or systems architecture is a fundamental and widely deployed construct used when designing software- or technology-intensive systems. Several architectural standards and frameworks are known and widely accepted: ISO/IEC/IEEE [23], TOGAF [34], OMG UML [32] and SysML [33], ArchiMate [24,35], and others.

Nevertheless, the formalization attempts of the core entities of all frameworks or standards—as of to-date—remain on an essentially non-mathematical and non-exact level using ordinary (i.e., every day and prose) language.

Interestingly, this is in stark contrast to many other disciplines where similar endeavours of formalization were undertaken over 100 years ago, e.g., mathematics [37], philosophy [39], language [8,10], architecture proper [27], or even poetry [20].

This situation has recently also been recognized by the International Standards Organisation (ISO) whose JTC1/SC7/WG42 working group on System Architecture began a review of the currently adopted (prose) definitions in 2018. However, no (new) standard has yet been issued.

In order to contribute to the resolution of this unsatisfactory status quo, this paper introduces a rigorous framework capturing the essence of the mathematical syntax of architectures. Note that we deliberately focus on the syntactical side of architecture omitting any discussion and treatment of the actual semantics of an architecture.

After a recapitulation of previous related work (section 2) we proceed in a highly deductive way and firstly present (in section 3) the theoretical underpinnings of our theory. The applicability of our theory is then considerably expanded in section 4 where we show that our approach is fully compatible with and able to serve as the very foundation of classical architecture standards. We also apply our formalism to current foundational work [12,15,38] and succeed in providing a rigorous mathematical formalization of concepts employed therein. The section is complemented by several

---

1 OMG’s OCL (object constraint language) [28] comes close to a rigorous language but is still limited to operating on the otherwise undefined core notions of UML.
(mathematical) theorems on the classification of architectures demonstrating that our purely syntactical theory has a merit of its own even without any reference to architectural semantics. *Inter alia*, we derive an interesting "no go" theorem that one cannot sensibly define the concept of modularization (i.e., the segmentation of a system into a disjunct set of constituting modules) on the level of syntax alone. This is in stark contrast to the case of *n*-tier architectures where this, as we are able to show, is possible. A concise discussion in section 5 provides additional links to closely related topics and points to potential further (future) work using our approach.

## 2 Related Work

Considerable research has been expended in identifying proper ways of formalizing (software) architecture with a lot of effort going into the definition of suitable Architecture Description Languages (ADLs). In addition to providing a (typically first-order logic based) concrete and strict syntax for modelling architectures, they typically add further functionality pertaining to displaying, analyzing, or simulating architectural descriptions written in a specific ADL [17].

Even before that, Category Theory has been put forward by some as a rigorous but enormously flexible methodology for computing science [18]. This branch of mathematics has sometimes been used to supplement classical ADL-based approaches [16,36].

However, Category Theory and architectural practice do not seem to match perfectly in many circumstances as the hurdle between concrete architectural work (e.g., conceiving, describing, and communication an architecture) and the intentionally highly abstract level of Category Theory remains large. Still, one very attractive concept here is the generalized notion of *morphisms* (including *isomorphisms*), that is structure preserving maps, which we will re-use in an appropriately specialized form.

Within service-oriented architectures (SOA) [31] the vagueness of the term *service* (which is almost universally defined in prose only) has prompted some authors to introduce a more rigorous formalization of their SOAs. Asha *et al.* [3] define a (web) service as a relation over the set of observable (service) properties and functionalities (which they call “appearances”). However, this definition is not further utilized in their contribution.

Yanchuk *et al.* [40] define a (logical) *service instance* *S* in terms of its functionality *f*, its interface *I* and the set of “coordinated and interfacing” processes *P*₁, *P*₂,.*, *P*ₖ actually implementing *S*: *S* = (*P*₁, *P*₂,.*, *P*ₖ, *Λ*). A process *P*ᵢ = *P*ᵢ(*f*, *I*) implements the functionality *f* of *S* and exposes it through interface *I*. *Λ* is the so-called “network communication function” between the individual processes. Recognizing SOA’s fundamental composition principle, where finer grained services may be aggregated into larger services, they define an *application* *A* as a directed graph with edges *E* over

---

²For instance, in order to formalize the—quite simple—idea that some components consist of other (sub-)components one needs (co-)limits right at the very heart of the theory.

³This is often called “orchestration” in distinction to “choreography”. The latter term refers to the coordination (in time) of a set of distinct services without a governing entity (such as an enterprise service bus or business process engine).
a set of service instances, \( A = \{V_A = \{S_1, S_2, ..., S_k\}, E \subseteq V_A \times V_A\} \). This formalization is then used to derive and rigorously define several “application classes”, e.g. facade and satellite applications, transient applications, accumulating applications, and others.

A similar, but less refined set-theoretic approach is also found in [19]. There, an enterprise architecture \( \mathcal{E} \) is essentially defined as an otherwise unstructured 8-tuple \( \mathcal{E} = \{R, B, S, D, A, T, C, M\} \) comprising the set of requirements \( R \), business processes \( P \), business systems \( S \), data elements \( D \), applications \( A \), technologies \( T \), constraints, metadata, design rules \( C \), and the set of architectural metrics \( M \). Due to the complete lack of any further structure besides the categorization of architectural elements into 8 sorts, this definition does not convey any syntactical information.

The requirement of being able to transform architectural models into one another has been evident since the very beginnings of architecture work and lies at the heart of OMG’s model driven development (MDD) and the succession of models contained in this approach. A formalization of this transformation, though, is not provided within MDD itself. Attempts to do so recognize that, in order to be able to formally (later then also automatically) transform one model into another, one has to specify the vocabulary of the two models, and the syntax they possess [26]. In this reference, the transformation of a source model into a target model is specified as a so-called “weaving model” capturing the “correspondence” between the different model elements. In our own contribution, we will rigidly formalize this notion.

From a decidedly logical point of view, the concept of a signature of a (formal) language \( L \) has been used to “define” the syntax of architectures [7][11]. In pure model theory [13][22], a signature \( \sigma = \{R, F, C\} \) defines the set of non-logical symbols of \( L \), i.e., its vocabulary consisting of the set \( R \) of relation symbols, \( F \) of function symbols, and constants \( C \). Translated to computer science, a signatures specifies the name space of an architecture; the set of constants \( C \) is then equated with (or replaced by) the set of sorts \( S \).

However, without any additional structure or further information, a signature alone does not contain any syntactical information at all because \( \sigma \) does not restrict the way how terms of a given language \( L \) are formed. Or to put it otherwise: A (such understood) signature of an architecture does not prescribe any syntax at all. Hence, the claim of having defined the “syntax of an architecture” by solely specifying \( \sigma \) that is providing the names of the architecture elements and just the names of the relations they potentially have to each other fails. Recognizing this, de Boer et al. [11] introduce a partial order on the set of (primitive) sorts \( S \) and on the set of relation symbols \( R \). Thereby they are able to capture certain “ontological” aspects of an architecture like generalization, aggregation, or containment.

Even though endowing architectures (e.g., diagrams or other representations) with precise meaning has always been a key concern of architecture research and practice alike [17], it is important to recall that this contribution focuses on the syntactical parts as opposed to the semantic aspects of an architecture. While this distinction between syntax and semantics is often recognized (e.g., [1][7][11][12]), the focus very often lies on the semantic elements with only a rudimentary or narrow formalization of the underlying
syntax. In this sense, our work is complementary to the semantic “thread” within the architecture domain.

3 Theory

3.1 Basic Definitions

Let us start directly with the definition of an architecture.

Definition 1 (architecture). An architecture is a complex \( \mathcal{A} \), written \( (A, R, F) \), with

1. \( A \), a (possibly empty) set of elements of the architecture called the universe of \( \mathcal{A} \),
2. \( R = \{ R_i = (a_1^{(i)}, a_2^{(i)}, ..., a_k^{(i)}) \subseteq A^k \} \), a countable set of relations \( R_i \), \( i \in \mathbb{N} \) of arbitrary arity \( k \in \mathbb{N} \) over the universe \( A \), and
3. \( F = \{ f_j : A_j^* \subseteq A^m \rightarrow A, j, m \in \mathbb{N} \} \), a countable \( (j \in \mathbb{N}) \) set of functions \( f_j : A_j^* \rightarrow A \). In order to avoid partial functions, we associate to each function \( f_j \) its domain, \( \text{dom}(f_j) = A_j^* \).

We write \( R^A \) or \( f^A \) when we want to highlight that relation \( R \) or function \( f \) belongs to architecture \( \mathcal{A} \) (and not to \( \mathcal{B} \)).

We also abbreviate \( R(a_1, a_2, ..., a_n) \) for \((a_1, a_2, ..., a_n) \in R \in R \).

We write \( \alpha(R) = k \) or \( \alpha(f) = k \) to denote the arity of a relation \( R \) or a function \( f \) over \( A^k \); that is \( R = (a_1, a_2, ..., a_{\alpha(R)}) \subseteq A^{\alpha(R)} \) or \( f = f(a_1, a_2, ..., a_{\alpha(f)}) \).

While the universe \( A \) is fairly easy to motivate, one might question the requirement of \( R \) being a set of relations \( R_i \) instead of a multiset. As relations are unnamed (i.e., just defined by their extension), this might prevent our formalization from being able to represent certain architectural structures. Consider an architecture \( \mathcal{A} \) with \( A = \{a, b\} \) and the two “relations” \( R_{\text{calls}}(a, b) \) and \( R_{\text{protects}}(a, b) \). As dealing with multisets is cumbersome at times, we rather propose to introduce two additional elements \( n_1 \) and \( n_2 \) to the universe \( A \) and include these explicitly as additional argument in the two relations, like \( R_{\text{calls}}(n_1, a, b) \) and \( R_{\text{protects}}(n_2, a, b) \). If, for whatever reasons, we want to preserve the arity of \( R \) we can create “dummy” relations like \( R_{\text{calls}}(n_1, n_1) \) and \( R_{\text{protects}}(n_2, n_2) \) to distinguish them.

The functions of set \( F \) may serve several purposes:

1. Functions may be used to specify the attributes of an element of the architecture \[11\] (even though relations do suffice for this, using functions for this might be more efficient or economical).
2. Functions may express architectural design constraints on (suitable subsets of) architectural artefacts \[17\] or analysis rules \[2\] like connectedness, reachability, self-sufficiency, latency etc.

3. Communications between components (e.g., processes) may be represented by suitable “network communication functions” \[3, 40\].

4. In general, functions may also be used to symbolically specify the dynamics of an architecture, e.g., (i) in the form of actions changing the state of the systems, or (ii) for encoding data transformations \[11\].

Because we want to focus on the bare minimum of mathematical syntax, we have deliberately not introduced any internal structure of the universe \(A\) like differentiating between nodes, software, applications, services, interfaces, requirements, stakeholders, concerns, names, properties, or any other (minuscule or major) element of the architecture.

Users of an architecture (including architects themselves) rarely look at the full architecture comprising all its elements, relations, and properties, but typically refer to and use so-called views. In general, a view may be (colloquially) defined as a part of an architecture (description) that addresses a set of related concerns and is addressed to a set of stakeholders.

To formulate this more exactly, we introduce two notions:

- **structured views** which are (even though of smaller size) architectures in their own right, and
- **unstructured views** or just views representing arbitrary slices through an architecture.

**Definition 2** (sub-architecture, structured view). Let \(\mathcal{A} = \langle A, R^A, F^A \rangle\) and \(\mathcal{B} = \langle B, R^B, F^B \rangle\) be two architectures.

Then \(\mathcal{A}\) is called a sub-architecture or structured view of \(\mathcal{B}\), notation \(\mathcal{A} \subset \mathcal{B}\), iff

\[
\begin{align*}
A & \subset B, \\
R^A & \subseteq \{ R^A | R^A = R^B \cap A^{\alpha(R)} \ \forall R^B \in R^B \}, \\
F^A & \subseteq \{ f^A | f^A = f^B|_A \ \forall f^B \in F^B \}.
\end{align*}
\]

If \(\mathcal{A} \subset \mathcal{B}\) we also say that \(\mathcal{B}\) is a super-architecture of \(\mathcal{A}\) and write \(\mathcal{B} \supset \mathcal{A}\).

The notation \(f^A = f|_A\) means that \(f^A\) is identical to \(f\) restricted to the domain \(A\). Provisions (2) and (3) in the above definition serve to restrict the relations and functions to the subdomain \(A \subset B\).

Dropping the requirements that a view of an architecture needs to be a (sub-) architecture of its own, we define an (unstructured) view as follows:

**Definition 3** ((unstructured) view). Let \(\mathcal{A} = \langle A, R, F \rangle\) be an architecture. Then the complex \(V = \langle A_V, R_V, F_V \rangle\) is called an (unstructured) view of the architecture \(\mathcal{A}\) iff

\[
\begin{align*}
A_V & \subseteq A, \\
R_V & \subseteq R, \\
F_V & \subseteq F.
\end{align*}
\]

6
This definition deliberately accepts the possibility that one may construct a view \( V \) comprising a set \( A_V \subset A \) and a relation \( R_V \in R_V \) which are incompatible, i.e., that \( \neg R_V(a_1, a_2) \forall a_1, a_2 \in A_V \). One encounters this in situations where just naming relation \( R_V \) in a view is already conveying important information to one of the stakeholders.

Methodologically, a view is specified by means of a viewpoint, which prescribes the concepts, models, analysis techniques, and visualizations that are provided by the view. Simply put, a view is what you see and a viewpoint is where you are looking from [35]. This is used in the following definition:

**Definition 4** (viewpoint). Let \( \mathcal{A} = (A, R, F) \) be an architecture and

\[
V^A = \{ V \mid V \text{ is a (structured or unstructured) view of } \mathcal{A} \}
\]

be the set of all views of \( \mathcal{A} \). Then an injective map

\[
W_I : I \to V^A \\
i \mapsto V^A_i \in V^A
\]

is called a viewpoint.

In our definition, a viewpoint may comprise many arbitrary views (selected by the index set \( I \)) of both kinds, structured or unstructured ones. This corresponds to cases where one wants to use several different architectural models for describing some aspects of a system. For economic reasons we rather want to have \( i \neq j \Rightarrow V^A_i \neq V^A_j \), hence \( W_I \) has to be injective.

### 3.2 n-tier Architectures

Let us now link the abstract concepts and definitions more closely to architectural practice. We first observe that the majority of architectural diagrams consists of a collection of ”boxes” or other closed shapes (typically representing components or other architectural artefacts) with suitable ”connectors”, i.e., lines drawn in the space between the boxes connecting two boxes (representing the association between the two entities) [6]. These “boxes & connectors” architectures can be formally defined as follows.

**Definition 5** (boxes & connectors (B&C) architecture). An architecture \( \mathcal{A} = (A, R \subseteq \mathcal{P}(A^2), \emptyset) \) is called a boxes & connectors (B&C) architecture.

We shall often write \( \mathcal{A} = (A, R) \) for a B&C architecture in the following.

In passing, we note that not all \( n \)-ary relations may be represented by binary relations without loss. The expressiveness of B&C architectures, thus, is limited.

Tiered architectures paradigmatically capture the fundamental (not only software) engineering design principle of separation of concerns. A tier thereby hides the implementation or execution complexity of one adjacent
tier while providing services to the other adjacent tier. This not only enor-
mously aids architects and developers alike, but may also be used for other
tasks, e.g., optimization [9].

In full abstraction, n-tier architectures can then be rigorously defined as
follows.

**Definition 6** (n-tier architecture). Let $\mathcal{A} = \langle A, R, F \rangle$ be an architecture. Let $C = \{C_1, C_2, \ldots, C_n\} \subset \mathcal{P}(A)$ be a partition of $A$ into $n$ disjunct sets\footnote{I.e., $\bigcup_{i=1}^{n} C_i = A$ and $C_i \cap C_j = \emptyset \quad \forall i \neq j$.}.

Let $k = \alpha(R)$ denote the arity of relation $R \in R$.

Then $\mathcal{A}$ is an n-tier architecture iff $\forall R \in R$ and $\forall a_1, \ldots, a_k \in A$

$$R(a_1, \ldots, a_j, \ldots, a_k) \Rightarrow \exists l, 1 \leq l < n : a_j \in C_l \lor a_j \in C_{l+1} \forall j = 1, \ldots, k.$$ 

In this case, the classes $C_i$ are called the tiers of the architecture. An
element $a \in C_i$ is said to belong to tier $i$. Tiers $i, j$ with $|i - j| = 1, i \neq j$
are called adjacent.

Informally, in an n-tier architecture any element $a$ in tier $i$ (i.e., $a \in C_i$),
has only relations with either another element in the same tier or in an
adjacent tier $i - 1$ or $i + 1$. Furthermore, relations never span more than
two tiers; they always contain either elements of a single tier only or just of
two adjacent tiers. We note that both peer-to-peer (P2P) and client/server
(C/S) architectures are 2-tier architectures in our sense. This is expected in
and consistent with a purely syntactical approach that (by design) cannot
distinguish between the “roles” of the two entities in a 2-tier setting. Only
semantics (e.g., of the relations connecting the two tiers) is able to provide
this discrimination.

We note in passing that every n-tier architecture may be turned into
an $(n - 1)$-tier architecture by simply merging two adjacent tiers $C_i$ and
$C_{i+1}$ into a new tier $C_i^*$. While this might be regarded as annoying from
a methodological (or foundational) point of view, it actually is encountered
frequently in (real, corporate) architectural work where architecture (dia-
agrams) is (are) often simplified in this way. One option to partially remedy
this situation is to define a maximal n-tier architecture, which is an n-tier
architecture that cannot be arranged into an $(n + 1)$-tier architecture.

We will now introduce some special architectures needed later. The
empty architecture $\mathcal{T}_0$ is introduced for the sake of completeness and closure
and may become important in the later development of this line of thinking.
The symbol $\mathcal{T}$ will be motivated later.

**Definition 7** (empty architecture). The architecture

$$\mathcal{T}_0 = \langle \emptyset, \emptyset, \emptyset \rangle$$

is called the empty architecture.

We call the next more complex architecture the trivial architecture $\mathcal{T}_1$.
Admittedly, it does not bring too much structure with it, but exhibits a
slight metamathematical twist, though:
Definition 8 (trivial architecture). Let $\text{id}_A : A \to A, a \mapsto \text{id}_A(a) = a$, be the identity function on $A$. Then the architecture
\[
\mathcal{T}_1 = \langle A = \{a\}, \{(a,a)\}, \text{id}_A \rangle
\]
is called the trivial architecture.

As the universe $A$ of $\mathcal{T}_1$ only contains a single element, no attributes or properties of the architecture are indicated at all. Consequently, we also do not get to know any structural properties of this architecture other then identity ($a = a$) and idempotence ($\text{id}_A$). Note in passing that $\mathcal{T}_0 \subset \mathcal{T}_1$.

We have introduced the generic $n$-tier architectures above. As we shall later see, they are closely related to an elementary form $\mathcal{T}_n$, defined as follows.

Definition 9 (elementary $n$-tier architecture). Let $T_n = \{1, 2, \ldots, n\}$ and define the symmetric binary relation $T_L$ ("linked to") over $T_n \times T_n$, i.e., $T_L(i, j) \forall i, j \in T_n$ as follows:
\[
\begin{align*}
|i - j| & \leq 1 \Rightarrow T_L(i, j), \\
|i - j| & > 1 \Rightarrow \neg T_L(i, j).
\end{align*}
\]
Then the B&C architecture
\[
\mathcal{T}_n = \langle T_n, \{T_L\} \rangle
\]
is called the elementary $n$-tier architecture.

Obviously, when we want to construct an elementary $n$-tier architecture we cannot go below or beyond $n$ elements in the universe $A$ of the architecture. And relation $T_L$ on the set of the first $n$ integers is a very primitive relation on $T_n \times T_n$.\footnote{We admit a slight, albeit irrelevant, notational imprecision at this point: In definition \textfeature{} we have denoted the single element of the universe $a$, while here we simply equate the $n$ elements with the first $n$ integers. As we later shall see (in definition \textfeature{}) these conventions are isomorphic and, hence, irrelevant when it comes to discussing the structural properties of our architectures. They do, though, simplify our proofs.}

3.3 Architecture Homomorphisms

Having defined the syntax of architectures we now turn to formalizing concepts for comparing different architectures in order to determine whether they are similar or even equal to each other. This task is typically accomplished by defining suitable structure-preserving maps from one architecture to another, so-called (homo-)morphisms.

Definition 10 (homomorphism, isomorphism). Let $\mathcal{A} = \langle A, R^A, F^A \rangle$ and $\mathcal{B} = \langle B, R^B, F^B \rangle$ be architectures. Then a (generalized) map $h : \mathcal{A} \to \mathcal{B}$, consisting of three individual maps, $h = \langle h_0, h_R, h_F \rangle$, is called a homomorphism between $\mathcal{A}$ and $\mathcal{B}$, and $\mathcal{A}$ is said to be homomorphic to $\mathcal{B}$.
iff
\[
h_0 : A \to B, \\
ah \mapsto h_0(a) \in B.
\]
\[
h_R : R^A \to R^B, \\
R^A \mapsto h_R(R^A) \quad \text{with } \alpha(R^A) = \alpha(R^B).
\]
\[
h_F : F^A \to F^B, \\
F^A \mapsto h_F(F^A) \quad \text{with } \alpha(F^A) = \alpha(F^B).
\]
(with $\alpha(\cdot)$ denoting the arity of a relation $R$ or function $f$. See definition 7) and
\[
R^A(a_1, a_2, \ldots, a_m) \Rightarrow h_R(R^A)(h_0(a_1), h_0(a_2), \ldots, h_0(a_m)), \quad (8)
\]
\[
h_0(f^A(a_1, a_2, \ldots, a_k)) = h_F(f^A)(h_0(a_1), h_0(a_2), \ldots, h_0(a_k)). \quad (9)
\]
$h$ is called injective, (surjective, bijective) if all three maps $h_0$, $h_R$, and $h_F$ are injective (surjective, bijective) on their respective (co-)domains.

In case $h$ is bijective and \([8]\) is valid in both directions, i.e.,
\[
R^A(a_1, a_2, \ldots, a_m) \Leftrightarrow h_R(R^A)(h_0(a_1), h_0(a_2), \ldots, h_0(a_m)) \quad (10)
\]
h is called an isomorphism between $\mathcal{A}$ and $\mathcal{B}$.

Condition \([10]\) is required to render our definition of architecture isomorphism equivalent to the (generalized) notion well known from category theory \([5, 25]\). There a morphism $f : A \to B$ is an isomorphism iff there exists a morphism $g : B \to A$ such that $f \circ g = \text{id}_B$ and $g \circ f = \text{id}_A$.

The following corollary finds that a set of architectures endowed with homomorphisms as per def. \([10]\) constitute a category (which we shall call Arch) \([25]\).

**Corollary 1.** Let $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots\}$ be a set of architectures and $\mathcal{H}_A = \text{Hom}_A$ be the set of homomorphisms over $\mathcal{A}$.

Then the pair $\langle \mathcal{A}, \mathcal{H}_A \rangle$ forms a category (which we shall call Arch).

**Proof.** We need to show that the “arrows” $f \in \mathcal{H}_A$, that is, the architecture homomorphisms as defined above, allow (i) an associative composition $(f \circ g)$ and (ii) have an identity element $\text{id}_A$ for all $\mathcal{A} \in \mathcal{A}$. Let’s define $h = g \circ f$ as follows:
\[
h_0 = g_0 \circ f_0, \\
h_R = g_R \circ f_R, \\
h_F = g_F \circ f_F.
\]
It is clear, that this definition automatically fulfils associativity but we still need to show that our $h$ fulfils the additional constraints \([8]\) and \([9]\). For the relational part $h_R$ of $h$ we find for any $R \in \mathcal{A}$
\[
R^A(a_1, \ldots) \Rightarrow f_R(R^A)(f_0(a_1), \ldots) \\
\Rightarrow g_R(f_R(R^A))(g_0(f_0(a_1)), \ldots) \\
= (g_R \circ f_R)(R^A)((g_0 \circ f_0)(a_1), \ldots) \\
= h_R(R^A)(h_0(a_1), \ldots).
\]
\(^{14}\) shorthand for $R \in R^A$ with $\mathcal{A} = \langle A, R^A, F^A \rangle$. 

10
For the functional part $h_F$ of $h$ we find for any $\phi^A \in \mathcal{A}$:

$$h_0(\phi^A(a_1, ...)) = (g_0 \circ f_0)(\phi^A(a_1), ...)
= g_0(f_0(\phi^A(a_1), ...))
= g_0(f_F(\phi^A))(f_0(a_1), ...)
= g_F(f_F(\phi^A))(g_0(f_0(a_1)), ...)
= (g_F \circ f_F)(\phi^A)((g_0 \circ f_0)(a_1), ...)
= h_F(\phi^A)(h_0(a_1), ...).$$

The identity arrow $\text{id}_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ for every object $\mathcal{A} \in \mathcal{A}$ may be easily defined as

$$\text{id}_{\mathcal{A}} = \langle \text{id}_A, \text{id}_{R^A}, \text{id}_{F^A} \rangle,$$
that is the identity map on the respective three domains $A$, $R^A$, and $F^A$.

This automatically ensures the required identity law, that $\forall f : \mathcal{A} \to \mathcal{B}$ we have $f \circ \text{id}_{\mathcal{A}} = f = \text{id}_{\mathcal{B}} \circ f$, which concludes the proof that the pair $\langle A, \mathcal{A}_A \rangle$ indeed forms a category, $\text{Arch}$.

We have defined the empty architecture $\mathcal{T}_0$ and the trivial architecture $\mathcal{T}_1$ earlier. In categorical terms, these special elements are initial and terminal objects as the following corollary shows.

**Corollary 2 (Initial & terminal objects of Arch).** The empty architecture $\mathcal{T}_0$ is the initial object in category $\text{Arch}$.

Let $\text{ArchBC}$ be the subcategory of $\text{Arch}$ where all architectures $\mathcal{A} \in \text{ArchBC}$ are restricted to be B&C architectures. Then the trivial architecture $\mathcal{T}_1$ is the terminal object in subcategory $\text{ArchBC}$.

**Proof.** Recall that an object $\mathcal{A} \in \text{Arch}$ is called an initial object, if for every element $\mathcal{A} \in \text{Arch}$ there exists a unique homomorphism $h : \mathcal{A} \to \mathcal{A}$. An element $\mathcal{A}$ is called terminal object, if for every $\mathcal{B} \in \text{Arch}$ there exists a unique homomorphism $g : \mathcal{B} \to \mathcal{T}$.

The first claim that $\mathcal{T}_0 = \langle \emptyset, \emptyset, \emptyset \rangle$ is initial in $\text{Arch}$ is trivial. Because all initial object are isomorphic to each other, we may speak of the initial object (and also vice versa of the terminal object).

Regarding the second claim that $\mathcal{T}_1 = \langle T_1 = \{a\}, \{\{a, a\}\}, \{\text{id}_{T_1}\} \rangle$ we need to observe that as per our definition homomorphisms $f : \mathcal{B} \to \mathcal{T}$ do not change the arity of any relation $R^B \in R^B$ or function $g : T^a \to T \in F^B$.

As our (proposed) terminal object $\mathcal{T}_1$ only contains a single binary relation, $\Delta_a^2 = \{\{a, a\}\}$, and a single unary function, $\text{id}_{T_1} : T_1 \to T_1, a \mapsto a$, we (i) either have to restrict the set of allowed architectures for which $\mathcal{T}_1$ should act as terminal, or (ii) we have to generalize the definition of $\mathcal{T}_1$, or (iii) we change our definition of homomorphisms on $\text{Arch}$.

If we simply restrict the domain of $g$ to B&C architectures $B = \langle B, R^B \subseteq \mathcal{P}(A \times A), \emptyset \rangle$ then it is easy to explicitly write down the unique $g : B \to T_1$:

$$g_0 : B \to T_1 = \{a\},$$

$$b \mapsto a \quad \forall b \in B,$$

$$g_R : R^B \to \Delta_a^2$$

$$R^B \mapsto \Delta_a^2$$
The uniqueness of \( g \) (i.e., its elements \( g_0 \) and \( g_R \)) is easy to observe because there simple is only a single way to map every object of \( \text{dom}(g) \) to a single element in the codomain \( \text{cod}(g) \). We also do not need to specify a \( g_F : \mathcal{F}^B \rightarrow \{ \text{id}_{T_1} \} \) because B&C architectures, by definition, do not contain any functions. Hence, \( \mathcal{T} \) is terminal in \( \text{ArchBC} \).

It is somewhat enlightening to investigate how we would have to expand the definition of \( \mathcal{T} \) if we want to make this new \( \mathcal{T}^*_1 \) terminal in the full category \( \text{Arch} \). Our strategy is based on the observation above that we need to endow \( \mathcal{T}^*_1 \) with suitable relations and functions over \( T_1 \) of arbitrary arity which may serve as the targets for the maps \( g : \mathcal{B} \rightarrow \mathcal{T}^*_1 \). Let’s fix the notation first and write

\[
\Delta^n_a = \left\{ (a, a, ..., a) \right\},
\]

for the diagonal relation of arity \( n \) over \( T_1 \) and also abbreviate a (countable) set of generalized identity functions \( f^{(k)}_a \) of increasing arity

\[
f^{(1)}_a : T_1 \rightarrow T_1, \quad a \mapsto f^{(1)}_a(a) = a
\]

\[
f^{(2)}_a : T^2_1 \rightarrow T_1, \quad (a, a) \mapsto f^{(2)}_a(a, a) = a
\]

\[
f^{(3)}_a : T^3_1 \rightarrow T_1, \quad (a, a, a) \mapsto f^{(3)}_a(a, a, a) = a
\]

\[\vdots\]

\[
f^{(k)}_a : T^k_1 \rightarrow T_1, \quad (a, a, ..., a) \mapsto f^{(k)}_a(a, a, ..., a) = a.
\]

With these generalizations it is straightforward to write down our terminal object \( \mathcal{T}^*_1 \) of \( \text{Arch} \).

\[
\mathcal{T}^*_1 = \langle T_1, \{ \Delta^2_a, \Delta^3_a, \Delta^4_a, ..., \Delta^n_a, ... \}, \{ f^{(1)}_a, f^{(2)}_a, f^{(3)}_a, ..., f^{(k)}_a, ... \} \rangle. \tag{11}
\]

The unique \( g : \mathcal{B} \rightarrow \mathcal{T}^*_1 \) then may be defined as follows:

\[
g_0 : B \rightarrow T_1, \quad b \mapsto a \quad \forall b \in B,
\]

\[
g_R : \mathcal{R}^B \rightarrow \{ \Delta^2_a, \Delta^3_a, ..., \Delta^n_a, ... \},
\]

\[
R^B \mapsto \Delta^n_a \quad \text{if } \alpha(R^B) = n,
\]

\[
g_F : \mathcal{F}^B \rightarrow \{ f^{(1)}_a, f^{(2)}_a, ..., f^{(k)}_a, ... \},
\]

\[
f^B \mapsto f^{(k)}_a \quad \text{if } \alpha(f^B) = k.
\]
3.4 Modules and Modularity

We have introduced the concept of \( n \)-tier architectures as one way to realize architectural separation of concerns earlier. In systems engineering and software engineering in particular and in information technology in general, we regularly encounter the segmentation of larger systems into distinct modules—called modularization—to achieve this on a finer scale \[15\].

Contrary to the approach in section 3.2 we will start the formalization with the elementary form of a modular architecture by suitably generalizing our notion of an elementary \( n \)-tier architecture (cf. definition \[9\]).

**Definition 11** (Elementary modular architecture of \( N \) modules). Let \( M_N = \{m_1, m_2, \ldots, m_N\} \) be a finite set and \( \mathcal{M}_N = (M_N, \{D^M\}, \emptyset) \) a B\&C architecture with a single relation \( D^M \subseteq M_N \times M_N \) (with intended semantics of depends on).

Then \( \mathcal{M}_N \) is called an **elementary modular architecture of \( N \) modules**.

The elements \( m_i \in M_N \) are called **modules** of \( \mathcal{M}_N \).

The following corollary \[13\] provides proof that, indeed, definition \[11\] generalizes the concept of an \( n \)-tier architecture

**Corollary 3.** Every elementary \( n \)-tier architecture \( T_n \) is an elementary modular architecture \( \mathcal{M}_n \) with \( n \) modules.

**Proof.** This immediately follows by comparing the definitions of an elementary \( n \)-tier architecture \( T_n = \langle T_n = \{1, 2, \ldots, n\}, T^D \subseteq T_n \times T_n \rangle \) with the definition of an elementary modular architecture with \( n \) modules, \( \mathcal{M}_n = (M_n = \{m_1, m_2, \ldots, m_n\}, D^M \subseteq M_n \times M_n) \).

Instead of trying to define a generic modular architecture (as we have done in definition \[6\] above) we will invoke our notion of architecture homomorphism as per definition \[10\] to define an arbitrary modular architecture comprising \( n \) elements (= modules).

**Definition 12** (Modular architecture and modules). Let \( \mathcal{M}_n = \langle M = \{m_1, m_2, \ldots, m_n\}, D^M \rangle \) be the elementary modular architecture with \( n \) modules.

An (arbitrary) B\&C architecture \( \mathcal{A} = \langle A, R^A \rangle \) is called a **modular architecture with \( n \) modules** iff there exists a surjective homomorphism \( h : \mathcal{A} \rightarrow \mathcal{M}_n \) from \( \mathcal{A} \) to \( \mathcal{M}_n \).

Denote the two maps comprising \( h \) as follows: \( h = \langle h_0, h_R \rangle \) with

\[
\begin{align*}
 h_0 : A &\rightarrow M_n, \\
 a &\mapsto m \in M_n \quad \forall a \in A,
\end{align*}
\]

\[
\begin{align*}
 h_R : R^A &\rightarrow \{D^M\}, \\
 R^A &\mapsto D^M \quad \forall R^A \in R^A,
\end{align*}
\]

and define the pullback \( h_0^\leftarrow (m) \) of an element \( m \in M_n \) as

\[
h_0^\leftarrow (m) = \{a \in A \mid h_0(a) = m \in M_n\}.
\]

\[\text{As we shall see later in section 4.3.3 we do not really have another option.}\]
Then the $n$ pullbacks $A_i = h^{-1}_{0}(m_i) \subseteq A$ are called the modules of $\mathcal{A}$ with regard to the modularization $\mathcal{M}$.

Note that the $A_i$'s form a partition of $A$ into $n$ disjunct (and non-empty) subsets as well. The additional characterization of the modularization of $\mathcal{A}$ “with regard to $\mathcal{M}$” is required because there may be other ways how to segment $\mathcal{A}$ into $n$ distinct modules with a certain dependency structure (as mediated by the relation $D^M$) between the modules (see section 4.3.3 for more details).

4 Application

4.1 Classical architectural standards and practice

In this section we present evidence why we believe that the definitions given above make sense. We do this by “mapping” concepts of various architectural metamodels to our architecture definition.

Let us first turn to the ISO/IEC/IEEE Standard 42010:2011 [23]. Then we can easily identify the elements of the ISO metamodel depicted in Figure 1 which are explicitly formalized by this work. This correspondence is depicted in Table 1.

![Figure 1: ISO/IEC/IEEE 42010: Architecture metamodel [23]](image-url)
| **CONSTRUCT** | **RECOGNITION IN THIS THEORY** |
|---------------|--------------------------------|
| Architecture  | The theory is capable of expressing architectures for the most complex systems thinkable because the theoretical constructs available basically make up most of our mathematical and logical system. |
| Architecture Description | Our complex $\mathcal{A} = \langle A, R, F \rangle$ essentially conforms to an "architecture description". Evidently, it can contain an arbitrary number of elements, relations, and functions with essentially unrestricted semantics. |
| Stakeholder   | Stakeholders may be identified by a suitable subset $S \subseteq A$. Relations over $S$ and supersets $B \supseteq S$ or functions including elements from $S$ may then be used to reason about or include stakeholders in the architecture. |
| Concern       | Concerns may be identified by a suitable subset $C \subseteq A$. Relations over $C$, $S$ and supersets $B' \supseteq S \cup C$ or functions including elements from $B'$ may then be used to reason about or include concerns and their stakeholders in the architecture. |
| Architecture Viewpoint | Viewpoints $\mathcal{V}$ are an intrinsic element of the theory. |
| Architecture View | Views $V^A$ are an intrinsic element of the theory. |
| Model Kind    | A "model kind" specifies conventions for a certain type of (architectural) modelling. This may be reflected easily in our theory by identifying suitable subsets $K_i \subseteq (R \cup F)$ selecting the types of functions and relations to be used in a specific "model kind". |
| Architecture Model | An architecture view consists of multiple models, each following one model kind. Therefore, we can identify an "architecture model" $\mathcal{M}$ by a suitable subset $M \subseteq A$ and a particular model kind $K_j$. |

Table 1: Comparison of ISO/IEC/IEEE 42010 and the current theory
Figure 2 shows the metamodel of the ArchiMate framework. The classes ELEMENT and RELATIONSHIP directly map to our own constructs of def. 1. The class RELATIONSHIP CONNECTOR contains two special elements, AND JUNCTION and OR JUNCTION, which are used to join two or more relations of the same type with the respective semantics of ‘∧’ and ‘∨’. The case that AND joins two relations, e.g. $R(a, b)$ and $R(a, c)$, is easily realized in our formalism as a (new) relation $R(a, b, c)$. In the case of an OR-join, we simply introduce a special element $\omega$ in our universe and can realize the same structure by $R(a, \omega)$, $R(\omega, b)$, and $R(\omega, c)$.

![Diagram of ArchiMate meta-model](image)

Finally, figure 3 shows a typical boxes & connectors architecture prevalent in today’s architectural representations and diagrams [21]. This particular architecture recognizes a single (binary and symmetric) relation $R(\cdot, \cdot) \subset A^2$ over the universe $A$ of capabilities with the semantic of data exchange or data flow between the capabilities.

The architecture (diagram) furthermore differentiates between several vertical layers like Mediation & Publishing or Analytics. There are also two horizontal sections shown in the diagram, Software as a service and On premise / edge.

All these constructs may be easily implemented in our rigorous architecture model by specifying certain subsets $L_i \subset A$ (conforming to the various layers and tiers) and defining suitable characteristic functions $f_{L_i} : A \rightarrow \mathbb{B} = \{0, 1\}$, as follows:

$$f_{L_i}(a) = \begin{cases} 0 & \text{if } a \in L_i, \\ 1 & \text{if } a \notin L_i. \end{cases}$$

4.2 Recent foundational work

4.2.1 Foundation of the term architecture

Recently we have seen renewed interest in and attempts at putting concepts, definitions, and terminology in general systems theory (including information technology and engineering) — such as architecture, structure, or even

\[16\] but not in the strict sense of an n-tier architecture according to def. 6.
the term system itself — on a firmer grounding (cf. [12, 14, 38]). Here we concentrate on one approach ([12, 38]) and demonstrate how our syntactical apparatus is fully capable of formalizing the prose definitions provided in these references.

In this approach, architecture is defined as a primary concept and not just one (important) property of the system under study. This definition is, inter alia, based on the observation that any architecture invariably includes a description of the relationships of the objects of a system, and that these relationships are further grounded in certain properties that some sets of objects may possess collectively or individually. This is called the structure (of a system) and defined as follows:

**Definition 13.** Structure is junction and separation of the objects of a collection defined by a property of the collection or its objects.

On a meta-level, the fact that a system $S$ possesses a certain structure may then be regarded as a (higher level) property of the system. At this (higher) level we can extend our thinking along the lines that also other systems, say $S'$ or $S''$, may possess this property. Such a generalization of a property is called a type\(^{17}\). This then leads to the definition of architecture in terms of the property “having some defined structure”:

**Definition 14.** An architecture is a structural type in conjunction with consistent properties that can be implemented in a structure of that type.

The first example given in [38] is a “system” of two inhomogeneous linear equations in 2 variables, $x_1$ and $x_2$, the architecture of which is given as the following matrix equation

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (12)$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \quad (13)$$

\(^{17}\)In software engineering, the term classifier is also frequently used.
Here the structural type consists of (i) a separation of all symbols present in eqn. 13 into three groups, \( A \), \( b \), and \( x \), and (ii) two “junctions”\(^{18}\) the juxtaposition of \( A \) and \( x \) and the linking of \( x \) and \( b \) through the equality sign ‘\( = \)’.

This can be easily formalized in our mathematical syntax in the following architecture ‘\( C \)’:

\[
C = \langle \ C = \{ A, x, b \}, \{ R_\ast, R_= \}, \emptyset \rangle. \tag{14}
\]

with relations \( R_\ast \) and \( R_= \) defined as follows:

\[
R_\ast = \{ (A, x) \}, \quad R_= = \{ (x, b) \}. \tag{15}
\]

The consistent properties of an architecture (cf. definition 14) then are the three equalities given in eqn. 13. We note that these properties are not part of the architecture formalization ‘\( C \)’. This feature is also present in the original example in \[38\] and, in our view, a sign of an incomplete formalization. A (syntactically) exhaustive architecture would also have to include these properties which may be easily achieved by extending ‘\( C \)’ into a super-architecture ‘\( C^\ast \):

\[
C^\ast = \langle \ C^\ast = \{ A, x, b \}, \{ R^\ast_\ast, R^\ast_\ast \}, \{ f^\ast \} \rangle \tag{16}
\]

with \( C^\ast_E = \{ a_{11}, a_{12}, a_{21}, a_{22}, x_1, x_2, b_1, b_2 \} \), \( R^\ast_\ast = R_\ast \), and \( R^\ast_\ast = R_\ast \). The (new) function \( f^\ast(\cdot) \) then captures the property how the individual variables and constants present in eqn. 13 relate to the structural part of the architecture as given in eqn. 12:

\[
f^\ast : C^\ast_E \to C \text{ with } \begin{cases} a_{ij} \to A & i, j = 1, 2; \\
x_i \to x & i = 1, 2; \\
b_i \to b & i = 1, 2. \end{cases} \tag{17}
\]

It is easy to verify that the architecture ‘\( C^\ast \)’ defined in eqn. 16 is indeed a super-architecture of ‘\( C \)’, i.e. ‘\( C^\ast \supset C \)’, in the sense of def. 2. Alternatively we may formulate that the matrix equation 12 without properties of eqn. 13 only represents a sub-architecture or structured view of the full architecture. This also perfectly mirrors the verbal definition of architecture in def. 14 which explicitly requires a “conjunction” of the structural type with the consistent properties.

In a second example, Wilkinson et al.\[38\] derive the architecture of generic torch-like lighting systems (i.e. torches using burning wax or batteries) based on the framework of Conceptual Structures\[30\]. The final architectural structure is given in figure 4 with the boxes representing the fundamental “concepts” of the architecture and the arrows signifying the relations between them.

It is plainly evident that this architecture is a simple B&C architecture\[19\] with a universe containing 7 elements and 7 binary relations.

Tangential to our main points we observe that the relation is transferred by between the two concepts Energy Store and Energy Transfer Mechanism does not seem to be correct: Not the Energy Store is transferred but the

\(^{18}\)cf. definition 13 above.

\(^{19}\)It also happens to be a maximal 6-tier architecture.
energy carrier contained in the **Energy Store** (e.g., electricity in a battery, wax in a Hessian role). The same imprecision applies to relation is consumed by. This seems to be a consequence of the fact that the architecture is missing the important additional concept of **Energy** or **Energy Carrier**.

This purely relational approach is extended in [12] where architectures are described using first order predicate calculus restricted to the domain (or universe in our terminology) of architectural concepts. As our definition [1] is sufficiently powerful to be interpreted as a “model” (in a model theoretic sense [13][22]) of such a “theory”, the formalization given in this contribution is logically equivalent (by Gödel’s Completeness Theorem) to the approach of [12].

### 4.2.2 Foundation of the terms **module** and **modularity**

In the context of system engineering, Efatmaneshnik et al. [15] discuss and define the notions **module**, **modularity**, and **modular**. They define a system \( S \) as a set \( S = \{S_P, S_F, S_{NF}\} \) comprising

- the set of (physical) subsystems \( S_P \) with \( S_P = \{s_1, s_2, ..., s_{n_P}\} \),
- the set of functional requirements \( S_F \) with \( S_F = \{f_1, f_2, ..., f_{n_F}\} \), and
- the set of non-functional requirements \( S_{NF} \) with \( S_{NF} = \{r_1, r_2, ..., r_{n_{NF}}\} \).

Each of the \( n_P \) physical subsystems \( s_i \) then consists of several (physical) components, \( c_{i_j} \), viz. \( s_i = \{c_{i_1}^1, c_{i_1}^2, ..., c_{i_1}^{n_i}\} \).

Sadly, without any mathematical definition, they introduce the core notion **realizes** between physical components and the functional and non-functional requirements postulating that physical subsystems realize both, functional and non-functional requirements and write this as follows:

\[
S_P \xrightarrow{\text{realizes}} S_F, \\
S_P \xrightarrow{\text{realizes}} S_{NF}.
\]
The notion realizes is further extended to the level of subsystems as follows:

\[ s_i \in S_P \xrightarrow{\text{realizes}} \{f_{i_1}, f_{i_2}, \ldots, f_{i_m}\} \subseteq S_F, \]

with the meaning that “a set of main system functions are [sic!] satisfied/delivered/realized by a subsystem”.

Let us first fix the imprecise notation from above with the following definition of the “arrow” notation before turning to their propositions and claims.

**Definition 15** (arrow notation). Let \( A \) and \( B \) be sets. Let \( R \subseteq A \times B \) denote a binary relation over \( A \times B \). Then we write

\[ a \xrightarrow{R} b \iff (a, b) \in R, \]

\[ a \xrightarrow{R} \{b_1, b_2, \ldots, b_k\} \subseteq B \iff \forall b_i, i = 1, \ldots, k : (a, b_i) \in R. \]

Then we can define system in the sense of [15] precisely as follows.

**Definition 16** (Efatmaneshnik system). A system \( S \) is a quadruple \( S = \langle S_P, S_F, S_{NF}, R \rangle \) comprising the set of (physical) subsystems \( S_P \), and the sets of functional \( (S_F) \) and non-functional \( (S_{NF}) \) requirements,

\[ S_P = \{s_1, s_2, \ldots, s_{n_P}\}, \quad (18) \]

\[ S_F = \{f_1, f_2, \ldots, f_{n_F}\}, \quad (19) \]

\[ S_{NF} = \{r_1, r_2, \ldots, f_{n_{NF}}\}, \quad (20) \]

and a binary relation \( R \subseteq \mathcal{P}(S_P) \times (S_F \cup S_{NF}) \) (with the intended semantics of realization), such that

\[ S_P \xrightarrow{R} S_F, \quad (21) \]

\[ S_P \xrightarrow{R} S_{NF}. \quad (22) \]

Each of the \( n_P \) physical subsystems \( s_i \in S_P \) may then consist of several (physical) components, \( c^i_j \),

\[ s_i = \{c^i_1, c^i_2, \ldots, c^i_{n_i}\}, \quad (23) \]

fulfilling

\[ \{s_i\} \xrightarrow{R} \{f_{i_1}, f_{i_2}, \ldots, f_{i_m}\} \subseteq S_F, \quad (24) \]

for some \( f_{i_m} \).

Note that the left argument of \( R(\cdot, \cdot) \) needs to be an element of the powerset of \( S_P \), \( \mathcal{P}(S_P) \), i.e. itself is a set. We need this to be able to formalize the fact that (at times) two subsystems \( s_1 \) and \( s_2 \) collectively realize a functional \( f \) or nonfunctional requirement \( r \) without any single one of them being able to achieve this. For instance, let \( s_i, i = 1, 2 \), denote two servers and let \( r_{HA} \) stand for the nonfunctional requirement of high availability. Then the fact that an active/active configuration of the two servers realizes \( r_{HA} \) may be written as

\[ \{s_1, s_2\} \xrightarrow{R} r_{HA}. \]

20
but

\[ \neg \{ \{ s_1 \} \xrightarrow{R} r_{HA} \}, \]

\[ \neg \{ \{ s_2 \} \xrightarrow{R} r_{HA} \}. \]

The authors then define \textit{architectural modularity} \(^{20}\) iff \( m \equiv 1 \ \forall s_i \) in eqn. 24, that is, if it reads

\[ \forall i = 1, ..., n_F : \{ s_i \} \xrightarrow{R} f_i, \tag{25} \]

They then claim that condition 25 means “there is a one-to-one correspondence between subsystems [i.e., the \( s_i \)’s, CFS] and main functional requirements [i.e., the \( f_i \)’s, CFS]”.

Obviously, this claim is unfounded as the simple system

\[ S^* = \langle \{ s_1, s_2, s_3 \}, \{ f_1, f_2 \}, \emptyset, R^* \rangle \]

with relation \( R^* \)

\[ \{ s_1 \} \xrightarrow{R^*} f_1, \tag{26} \]

\[ \{ s_2 \} \xrightarrow{R^*} f_1, \tag{27} \]

\[ \{ s_3 \} \xrightarrow{R^*} f_2, \tag{28} \]

provides a counterexample where we certainly do not have a 1:1 correspondence between the \( s_i \)’s and the \( f_i \)’s (see also case A in figure 5).

Figure 5: Functional modularization

However, the original claim of “modularity” may be salvaged if we identify the \textit{modules} \( m_k \) not (mistakenly) simply by putting \( m_k \equiv s_k \), but by

\[ m_k = \{ s_i | \{ s_i \} \xrightarrow{R^*} f_k \} \quad k = 1, ..., n_F. \tag{29} \]

\(^{20}\)As we will argue later, the name is misleading. This particular form of modularity should be better characterized as “functional modularity”, not the least to demarcate this notion of \textit{modularity} from the “non-functional modularity” the authors introduce and treat in the rest of the paper.
The authors furthermore characterize the situation $m > 1$ in condition 24 as an “architecturally nonmodular” system where “function sharing” (according to Ref. [1]) takes place.21

Again, this proposition is not valid universally. Consider the system

$$ S^* = \langle \{s_1, s_2\}, \{f_1, f_2, f_3, f_4\}, \emptyset, R^* \rangle $$

with relation $R^*$

$$ \{s_1\} \xrightarrow{R^*} \{f_1, f_2\}, \quad (30) $$

$$ \{s_2\} \xrightarrow{R^*} \{f_3, f_4\}. \quad (31) $$

This system $S^*$ clearly (i.e., for all practical definitions of “architectural modularity”, in our opinion) consists of two modules, $m_1 = s_1$ and $m_2 = s_2$ (cf. case B in figure 5).

In passing we note that figure 2 in ref. [15] is incorrect because the depicted arrows, which are the graphical representation of the relation $\xrightarrow{\text{realizes}}$ according to the running text, wrongly extend from the “logical domain” (i.e., $S_F$) to the “physical domain” (i.e., $S_P$). The figure, thus, is either mislabelled or the direction of all (!) arrows needs to be reversed.

Nevertheless, we may salvage the whole notion of (architectural) modularity by generalizing the observation that in both cases A and B it was possible to identify certain partitions of the set $S_P$ and $S_F$ such that there is a 1:1 relationship $\xrightarrow{R^-}$ between elements of both partitions. Because this segmentation is solely based on the functional requirements $S_F$ we would rather call this a functional modularization of $S$ instead of the misleading “architectural modularization” as put forward in [15].

This immediately leads to the following universal definition.

**Definition 17** (functional modularization). Let $S = \langle S_P, S_F, S_{NF}, R \rangle$ be a system. Let $F = \{F_1, F_2, ..., F_i, ..., F_N\}$ be a partition of $S_F$ into $N$ disjunct subsets $F_i$ and define $R^-(\cdot)$ as

$$ R^-(F_i) = \{s \in S_P \mid \exists f \in F_i : \{s\} \xrightarrow{R} f\}. \quad (32) $$

In case the following condition holds for $F$:

$$ R^-(F_i) \cap R^-(F_j) = \emptyset \quad \forall i \neq j, \ i, j = 1, ..., N, \quad (33) $$

we call the set $M$

$$ M = \{m_1, m_2, ..., m_N\}, \quad m_i = R^-(F_i), \ i = 1, ..., N, \quad (34) $$

a functional modularization of the system $S$. The $N$ elements $m_i \in M$ are called (functional) modules of $S$ (remember $m_i \subseteq S_P$).

21Ref. [1] does not support the notion of “function sharing” because it is solely concerned with the internal modularization of a product in terms of its constituting components, i.e., of $S_F$ in our syntax, using elements of graph theory and the Design Strategy Matrix method. Neither requirements nor the relation of components or subsystems to them play any role in the core part of that work.

22The article mistakenly writes $i_m > 1$ instead of the intended $m > 1$. 
Evidently, the (implicit) partitions of $S_F$ in our previous examples (cf. equations 29–31) all fulfil condition 33. We have depicted a non-trivial example of functional modularization as case C in figure 5.

Let us now turn to the central definition of modularity as put forward in [15].

**Definition 18** (Efatmaneshnik modularization). A module is a group of some of a system’s elements (components, subsystems, etc.), with a physical or notional boundary, and is detachable, either physically or notionally from the system that, by this detachability alone, has a nonfunctional utility for one or more system lifecycle stages or stakeholders.

A system is modular if it either has modules or has certain qualities that ease or make possible the process of modularization.

A unit is modular if it is either a module itself or is a unit with the potential to be a module.

Modularity is an attribute of a system or a unit (of that system) designating (i) the quality of being modular or (ii) the potential to become modular.

For both, a system or a unit of a system, modularization is the process of becoming modular.

At its heart the definition of module says that any subset of elements of the system satisfying (to any degree) one or more nonfunctional requirements may be regarded a module. It is completely irrelevant if this separation is physically possible at all (arg. “physically detachable”) or not (arg. “notionally”); the simple ability to discriminate the elements that form part of the module from those which are not part of it suffices.

Consequently, they formalize a module $m$ as follows:

$$ m \subseteq \bigcup_{k=1}^{n_F} s_k, $$

with subsystems $s_k \in S_F = \{s_1, s_2, \ldots, s_{n_F}\}$. Actually, eqn. 5 in [15] is imprecise in defining $m = \cup c_i^j \ [sic!]$ with the $c_i^j \in s_i$ because the (collective) union operator $\cup$ is only defined for sets and not for individual elements. The correct formulation would have been $m = \bigcup \{c_i^j\}$.

The condition in definition 18 of “having a nonfunctional utility” is modelled by relating $m$ to the set of nonfunctional requirements $S_{NF}$ of the system $S$:

$$ m \xrightarrow{\text{fully or partially realizes}} R_{NF} \subseteq S_{NF}. $$

The introduction of this relation $\xrightarrow{\text{fully or partially realizes}}$ is problematic in two ways.

First, this is not the relation $R$ we have introduced in definition 16 above but a new one as it ranges over the components $c_i^j$ of the subsystems $s_i$ (cf. equ. 23 in definition 16).

Secondly, we note that the authors weaken the original meaning of relation $R$ as “realizing” (functional requirements) when they now also apparently accept semantics of “partial fulfilment”.

---

23 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

24 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

25 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

26 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

27 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

28 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

29 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

30 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

31 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

32 which also should have been written as a big $\cup$ instead of the binary operator $\cup$. 

33 which also should have been written as a big $\cup$ instead of the binary operator $\cup$.
Ignoring the slightly changed semantics of $R$ we can, nevertheless, salvage the formalization of module in the sense of \[15\] by suitable extending the range of $R$ to a new $R^*$:

**Definition 19** (Efatmaneshnik modularization 1). Let $S$ be a system

$$S = \langle S_P = \{s_1, s_2, ..., s_{n_P}\}, S_F, S_{NF}, R^* \rangle,$$

with the $s_i = \{c_{i1}^1, c_{i2}^1, c_{i3}^1, ...\}$ its subsystems and components $c_{i}^1$ thereof. Abbreviate the set of all components of $S$ by $S_C$, that is

$$S_C = \bigcup_{k=1}^{n_P} s_k,$$

and let $R^*$ be a binary relation (with the intended meaning of realizes) as follows:

$$R^* \subseteq \mathcal{P}(S_P \cup S_C) \times (S_F \cup S_{NF}).$$

A set $m$

$$m = \{c_{\alpha}, c_{\beta}, c_{\gamma}, ...\} \subseteq S_C$$

is called a **module** iff we have

$$m \xrightarrow{R^*} R_{NF} = \{r_{i_1}, r_{i_2}, ...\} \subseteq S_{NF}.$$}

for some set $R_{NF}$. A subset $m^* \subseteq m$ is called a **submodule** of $m$ iff

$$m^* \xrightarrow{R^*} r_i \in R_{NF}.$$

Definition 19 formalizes the notion of “detachability” (cf. definition 16) of a module from the rest of the system in eqn. 39 in the form of naming only. Eqn. 40 captures the idea that any module $m$ has to have “nonfunctional utility”.

Unfortunately, the definition above is apparently limited to defining a single module in complete ignorance of any other module the system may possess. This, however, is contrary to architectural or engineering practice where one is not so much concerned with identifying a single module within a system but in sensibly segmenting the whole system into a set of (distinct) modules. The authors briefly mention this but do not regard it in any way problematic.

Secondly (and because of the previous shortcoming) the above construction allows that two or more modules arbitrarily share the very same components. The authors remark

[...] some define the usage of standard components as modularity. This means that components $c^1_i$ and $c^1_j$ are the same type of components or are exactly the same component (which can happen in software where a piece of code is called/used by different parts of the program).

Now, while their formalism certainly allows this (vid. eqn. 39 does not forbid two modules $m$ and $m^*$ sharing one or more components $c^0_\delta$), this is typically and certainly not the case in real practice. If we have physically
detachable modules $m$ and $m^*$ sharing one or more components it would lead to the considerably strange situation where one can extract module $m$ but, in doing so, destroys module $m^*$ (or at least depriving it from the shared components).

The incomplete understanding is also evidenced in the quote above where relation calling (e.g. a subroutine, a library, or another module) is confounded with the notion of being member of a module. Remarkably, none of the three examples of (physical) modularization displays this (artificial) feature.

From a principal point of view, the paper repeatedly argues that every subsystem $s_i$ is a module. but not every module $m$ necessarily is a subsystem. Formally, that translates to the claim

$$\forall s_i \in S_P : \exists R_i \subseteq S_{NF} : s_i \xrightarrow{R^*} R_i.$$  \hspace{1cm} (42)

While on an informal level we certainly could not agree more with this assertion, the claim itself cannot be (logically correctly) deduced from the formalization of module given in the paper or reconstructed here. Technically, this lies in the fact that the two definitions given, eqn.s [10] and [22], are too generic to allow a cogent deduction of eqn. [42].

The root cause, though, is the fact that one cannot formalize modularity or define modules on the level of syntax only. This is rigorously proven in our “no go” theorem below.

The only thing we need to show is that we can apply our “no go” theorem to the definitions of [13], that is, that the formalization given here in this section is compatible with our own theory (as given in section 3).

For this we first extend definition [19] to a segmentation of the system $S$ into a disjunct set of $N$ modules.

**Definition 20** (Efatmaneshnik modularization 2). Let $S$ be a system, $S = \langle S_P = \{s_1, s_2, \ldots, s_{n_P}\}, S_F, S_{NF}, R \rangle$. Write $S_C = \bigcup_{i=1}^{n_P} c_i^1$ for the set of all components of $S$ and let $M = \{m_1, m_2, \ldots, m_N\} \subseteq P(S_C)$ be a partition of $S_C$ into $N$ distinct subsets $m_i \subseteq S_C$ such that

$$\forall i = 1, \ldots, N : \exists R_i \subseteq S_{NF} : m_i \xrightarrow{R_i} R_i.$$  \hspace{1cm} (43)

Then $M$ is called a nonfunctional modularization of $S$ and the $N$ $m_i$’s are called modules of $S$.

If we now collect all elements of $S$ in a single (flat) set $S^* = S_P \cup S_C \cup S_F \cup S_{NF} \cup M$ and remember that $m_i \xrightarrow{R_i} R_i$ is just another way to write $R(m_i, r_{i\alpha}) \forall r_{i\alpha} \in R_i$, then we can write down the B&C architecture $\mathcal{J}$ of the system $S$ as $\mathcal{J} = \langle S^*, R \rangle$ with a suitable $R \subseteq S^* \times S^*$.

### 4.3 Mathematical Reasoning

Here we demonstrate that our theory of architectures also provides a mathematical framework for reasoning about (and proving) facts about architectures in general.
4.3.1 Graphs and Architectures

**Theorem 1.** Every Graph $G = (V, E)$ with $V = \{V_1, V_2, ..., V_n\}$ the set of vertices and $E \subseteq V^2$ the set of (directed) edges of $G$, is isomorphic (in a model theoretic sense) to a B&C architecture $\mathcal{G}$.

**Proof.** Define a B&C architecture $\mathcal{G}$ as follows: $\mathcal{G} = \langle V^G = V, R^G = \{E\}, F^G = \emptyset \rangle$. Then, in a naive sense, we already can see the isomorphism. On an exact level, though, we first have to specify a framework within which we may relate the two objects $G$ and $\mathcal{G}$ to each other. We will use model theory here [13, 22]. Then graph $G$ and architecture $\mathcal{G}$ may be regarded as $L$-structures of the (simple) signature $L = \{E\}$ containing just the symbol for the edge relation. In this case, two $L$-structures are isomorphic iff there exists a bijective homomorphism $h : V \rightarrow V^G$ satisfying the following condition on $E$

$$(a, b) \in E \Leftrightarrow ((h(a), h(b)) \in E \in V^G).$$

The following surjective embedding $h$

$$h : V \rightarrow V^G$$

$$v \mapsto v \in V^G$$

provides the (trivial) structure preserving bijection between the two domains $E$ and $V^G$.

4.3.2 Characterization of n-tier architectures

Many architects use $n$-tier architectures due to their clear (and easy to follow) structure. The following lemma and theorem provide a rigorous characterization in terms of elementary $n$-tier architectures $\mathcal{T}_n$.

**Lemma 1.** A $n$-tier B&C architecture $\mathcal{A}$ is homomorphic to the elementary $n$-tier architecture $\mathcal{T}_n$, that is there exists a homomorphism $h : \mathcal{A} \rightarrow \mathcal{T}_n$.

**Proof.** Let $\mathcal{A} = \langle A, R \rangle$ be an $n$-tier B&C architecture. Then, by definition, every element $a \in A$ belongs to exactly one tier $C_i$ with $1 \leq i \leq n$. We also have $R \subseteq A^2$ because of the ”boxes & connectors” property. Then define the map $h : \mathcal{A} \rightarrow \mathcal{T}_n$ as follows:

$$h_0 : \quad A \rightarrow T_n = \{1, 2, ..., n\}$$

$$a \in C_i \mapsto i \in T_n$$

$$h_R : \quad R \rightarrow \{T^L\}$$

$$R \mapsto T^L$$

Assume $R(c_i, c_j)$ with $c_i \in C_i$ and $c_j \in C_j$. Then $h(c_i) = i$ and $h(c_j) = j$. Note that, from now on, we drop the subscripts on the three different maps $h_0$, $h_R$, and $h_F$ comprising $h$ for simplicity and readability reasons. It should be immediately evident from the respective domain $\text{dom}(h_\alpha)$ which $h_\alpha$ we are talking about. Because $\mathcal{A}$ is an $n$-tier architecture, we have

$$|i - j| \leq 1 \Rightarrow |h(c_i) - h(c_j)| \leq 1 = T^L(h(c_i), h(c_j)) = h(R)(h(c_i), h(c_j)).$$

Therefore $h$ is a homomorphism. \qed
We can strengthen the above lemma to characterize \( n \)-tier B&C architectures as follows:

**Theorem 2.** Every B&C architecture \( \mathcal{A} = \langle A, R \rangle \) is an \( n \)-tier B&C architecture iff there exists a surjective homomorphism to the elementary \( n \)-tier architecture \( \mathcal{T}_n \).

**Proof.** Lemma 1 already proves the \( \Rightarrow \) direction.
For the \( \Leftarrow \) direction we discriminate three cases and proceed in an indirect manner.

\( n = 2 \). Trivial.

\( n = 3 \). Let \( \mathcal{A} \) be an architecture which is not an \( n \)-tier architecture and assume, contrary to the theorem, that there exists a homomorphism \( h : \mathcal{A} \to \mathcal{T}_n \). Then let \( C_i = h^{-1}(i) \) be the pullback of \( h \). Because \( h \) is surjective, non-empty pullbacks \( C_i \) exist for \( i = 1, \ldots, 3 \). Because \( \mathcal{A} \) is not a 3-tier architecture, there exist elements \( e_i^{(j)} \in C_i \) with \( h(e_i^{(j)}) = i \) and relations \( R_{\alpha} \in R \) with

\[
R_{\mu}(e_1^{(1)}, e_2^{(1)}), R_{\nu}(e_2^{(2)}, e_3^{(1)}) \text{ and } R_{\sigma}(e_1^{(2)}, e_3^{(2)}).
\]

As \( h \) is a homomorphism we have

\[
R_{\sigma}(e_1^{(2)}, e_3^{(2)}) \Rightarrow h(R_{\sigma})(h(e_1^{(2)}), h(e_3^{(2)})) = T^{L}(1, 3),
\]

which is a contradiction.

\( n \geq 4 \). As in the case \( n = 3 \) but note that in addition to the cyclic pattern given above we also might encounter the extended form of this pattern with

\[
R_{\mu}(e_{i-1}^{(1)}, e_i^{(1)}), R_{\nu}(e_i^{(2)}, e_{i+1}^{(1)}) \text{ and } R_{\sigma}(e_i^{(2)}, e_k^{(2)}) \text{ with } |i - k| > 1.
\]

As \( h \) is a homomorphism we have

\[
R_{\sigma}(e_i^{(2)}, e_k^{(2)}) \Rightarrow h(R_{\sigma})(h(e_i^{(2)}), h(e_k^{(2)})) = T^{L}(i, k) \text{ with } |i - k| > 1.
\]

which, again, is a contradiction. \( \square \)

### 4.3.3 A “No Go” Theorem on Modularity at the Syntax Level

In section 3.4 we have introduced the concept of a generic modular architecture in terms of being homomorphic to an elementary modular architecture \( \mathcal{M}_n \). Based on the existence of stand-alone definition of (arbitrary) \( n \)-tier architectures (def. 6) one might now be induced to also look for a definition of a generic modular architecture independent of the existence of a homomorphism to \( \mathcal{T}_n \).

The following theorem, sadly, proves that this search is futile.

**Theorem 3** (Modularity No Go Theorem). Every B&C architecture \( \mathcal{A} = \langle A, R^A \rangle \) is a modular architecture with \( n \) modules for arbitrary \( n \) provided \( n \leq |A| \).

**Proof.** Let \( \mathcal{A} = \langle A, R^A \rangle \) be an arbitrary B&C architecture and choose \( n \leq |A| \). Furthermore, let \( \{A_1, A_2, \ldots, A_n\} \) be an (arbitrary!) partition of \( A \) into \( n \) disjunct sets. Then write the elementary modular architecture of
n elements \( M_n = \langle M_n, \{ D^M \} \rangle \) and define the relation \( D^M \subseteq M_n \times M_n \) as follows:

\[
\forall a_i \in A_i \ \forall a_j \in A_j \ \forall R^A \in R^A : R^A(a_i, a_j) \Rightarrow D^M(m_i, m_j).
\] (44)

Then define the homomorphism \( h : A \rightarrow M_n \) (with \( h = \langle h_0, h_r \rangle \)) as

\[
h_0 : A \rightarrow M,
\]
\[
a \in A_i \mapsto m_i \ \forall a \in A_i,
\]
\[
h_R : R^A \rightarrow \{ D^M \},
\]
\[
R^A \mapsto D^M \ \forall R^A \in R^A.
\] (46)

Note that we also allow \( i = j \).

It now remains to be shown that condition 8 of definition 10 is fulfilled. Take an arbitrary \( R^A \in R^A \) and any \( a_i \in A_i \) and \( a_j \in A_j \). Then by eqn. 44 we can write:

\[
R(a_i, a_j) \Rightarrow D^M(m_i, m_j)
\]
\[
= h_R(R^A(m_i, m_j))
\]
\[
= h_R(R^A(h_0(a_i), h_0(a_j))).
\]

which is all we need.

We want to stress the consequences of this (null-)result: Theorem 3 shows that we can (theoretically correct) “slice” any (B&C) architecture into arbitrary (!) many modules consisting of any (!) collection of elements of the architecture’s universe if we do not restrict the (dependency) relation between the resulting modules. This proves that one cannot sensibly define modularity on the syntactic level alone but needs to refer to additional semantics in order to be able to do so. The result, in hindsight, partly justifies the approach in [15] who argue that one should define modules in relation to (nonfunctional) requirements the modules fulfill or satisfy. However, the restriction to nonfunctional requirements only cannot be made plausible on the level of syntax as we are equally well and easy able to define modularization with regards to functional requirements (cf. our definition 17).

Interestingly, this is in stark contrast to the situation of \( n \)-tier architectures where the syntactical restriction of the relation between the individual tiers is sufficiently strong to allow non-trivial instantiations of the concept.

5 Discussion

5.1 Applicability

We show that our formal definition of the mathematical syntax of architectures encompasses most (if not all) elements and constructs present in today’s architectural standards and architectures produced and consumed by practitioners.

Furthermore, we demonstrate that our approach is an excellent complement to recent foundational work [12, 38], which focuses more on the semantics of architectures than on its syntax. Ref. 38 implicitly uses the
syntax of conceptual structures on a superficial level while ref. uses first order predicate calculus. Our approach is then used to improve and rigorously reconstruct on the syntactical (mathematical) level a theory on modularization as put forward in .

Capitalizing on our notion of architecture isomorphism, we then use our formalism to rigorously prove certain relations between architectures. On the formal side, we establish that the naive view that every “boxes and connectors” diagram of an architecture (represented rigorously as a graph) is fully warranted on the syntactical level. On the categorical side, we prove a classification theorem on general $n$-tier architectures, namely that a boxes and connectors architecture is an $n$-tier architecture if and only if it is homomorphic to an elementary $n$-tier architecture.

However, we are clearly aware that this paper can only capture the very beginning of what we believe could be a completely novel and fruitful way of reasoning about architectures in a mathematically rigorous way.

5.2 Relation to Model Theory

The relation of our definition of an architecture to the model theoretic concepts of an $(L)$-model or an $(L)$-structure over a signature $L$ is evident (cf. [13,22]). However, we note the following differences:

- We cannot refer to a common signature $L$ shared between (sets of) architectures.
- Contrary to an $(L)$-structure, we do not have singled out some elements of our (architectural) universe as constants.
- Our definition of (architecture) homomorphisms is generalizing the notion used in standard model theory. Because we cannot rely on the fact that a common signature $L$ implicitly ”fixes” relations and functions in two different architectures, we need to provide this mapping explicitly through the functions $h_R$ and $h_F$ (cf. def. [10]).

5.3 Further Work

One possible way to extend the work of this paper is to define refinements of architectures (like $\mathcal{A} \succ \mathcal{B}$) as another (order) relation besides the sub-architecture property ($\mathcal{A} \subset \mathcal{B}$). We could, in principle, also differentiate between a ”refinement” of architectural elements (artefacts) $a \in A$ and of relations ($R \in \mathcal{R}$) [7,29]. Thereby we could give a precise meaning to the various forms of “hierarchicalization” in architectural work.

We could also give a precise meaning of the semantics of a given architecture by linking our constructs $\mathcal{A}$ to $L$-structures and $L$-models over a given signature $L$. The signature $L$ could, for instance, then be related to some architectural description languages (ADL).

One can also further develop the properties of the category Arch and transfer results from Category Theory to the architecture domain. Finally, the explicit recognition and inclusion of names in our formalism (e.g. for relations) can be explored in more detail to increase the theory’s relevance to the field.
Acknowledgements

The author gratefully acknowledges the support of Harald Schöning, Software AG’s Vice President Research and the intensive (mathematical) discussions especially on the very first version of this paper. Additionally, I would like to thank Burkhard Hilchenbach, Lead Architect Hybrid in Software AG’s CTO Office, for his comments and pointers to additional topics of high relevance to (true) architects. Finally, I am indebted to my wife, Susi, for repeatedly proof-reading the various versions of this work. Being a biologist, she could only improve the English language, though, so all the conceptual and mathematically mistakes are solely mine ;-).

References

[1] Agrawal, A.: Product networks, component modularity and sourcing. Journal of Technology Management & Innovation 4(1), 59–81 (May 2009). https://www.jotmi.org/index.php/GT/article/view/art105

[2] Altoyan, N., Perry, D.E.: Towards a well-formed software architecture analysis. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, p. 173–179. ECSA ’17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3129790.3129813

[3] Asha, H., Shantharam, N., Annamma, A.: Formalization of SOA concepts with mathematical foundation. Int. J. Elec. & Comp. Eng. 10(4), 3883–3888 (2020). https://doi.org/10.11591/ijece.v10i4.pp3883-3888

[4] Astesiano, E., Reggio, G., Cerioli, M.: From formal techniques to well-founded software development methods. In: Aichernig, B.K., Maibaum, T. (eds.) Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers, pp. 132–150. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40007-3_9

[5] Barr, M., Wells, C.: Category Theory for Computer Science. Prentice-Hall (1989)

[6] Bass, L., Clements, P., Kazmann, R.: Software Architecture in Practice. SEI Series in Software Engineering, Pearson Education, Upper Saddle River: NJ, 3rd edn. (2013)

[7] Broy, M.: Mathematical system models as a basis of software engineering. In: van Leeuwen J. (ed.) Computer Science Today, vol. 1000, pp. 292–306. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/BFb0015250

[8] Carnap, R.: Die Logische Syntax der Sprache. Springer (1934)
[9] Chiang, M., Low, S.H., Calderbank, A.R., Doyle, J.C.: Layering as optimization decomposition: A mathematical theory of network architectures. Proceedings of the IEEE 95(1), 255–312 (2007). https://doi.org/10.1109/JPROC.2006.887322

[10] Chomsky, N.: The Logical Syntax of Linguistic Theory. Berlin, Springer (1975)

[11] de Boer, F.S., Bonsangue, M.M., Jacob, J., Stam, A., van der Torre, L.: A logical viewpoint on architectures. In: Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. pp. 73–83 (2004)

[12] Dickerson, C.E., Wilkinson, M., Hunsicker, E., Ji, S., Li, M., Bernard, Y., Bleakley, G., Denno, P.: Architecture definition in complex system design using model theory. IEEE Systems Journal pp. 1–14 (2020). https://doi.org/10.1109/JSYST.2020.2975073

[13] Doets, K.: Basic model theory. Studies in logic, language and information, CSLI Publications, Stanford, MA (1996)

[14] Dori, D., Sillitto, H., Griego, R.M., McKinney, D., Arnold, E.P., Godfrey, P., Martin, J., Jackson, S., Krob, D.: System definition, system worldviews, and systemness characteristics. IEEE Systems Journal pp. 1–11 (2019). https://doi.org/10.1109/JSYST.2019.2904116

[15] Efatmaneshnik, M., Shoval, S., Qiao, L.: A standard description of the terms module and modularity for systems engineering. IEEE Transactions on Engineering Management 67(2), 365–375 (2020). https://doi.org/10.1109/TEM.2018.2878589

[16] Fiadeiro, J.L., Maibaum, T.: A mathematical toolbox for the software architect. In: Proceedings of the 8th International Workshop on Software Specification and Design. p. 46. IWSSD ’96, IEEE Computer Society, USA (1996). https://doi.org/10.1109/IWSSD.1996.501146

[17] Garland, D.: Formal modeling and analysis of software architecture: Components, connectors, and events. In: Bernardo, M., Inverardi, P. (eds.) Formal Methods for Software Architectures: Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003. Advanced Lectures, pp. 1–24. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39800-4_1

[18] Goguen, J.A.: A categorical manifesto. Technical Manifesto PRG-72, Oxford University Computing Laboratory, Programming Research Group (March 1989)

[19] Goikoetxea, A.: A mathematical framework for enterprise architecture representation and design. Int. J. Inf. Technol. Decis. Mak. 3, 5–32 (2004)

[20] Hamburger, K.: Die Logik der Dichtung. Klett, Stuttgart (1987)
[21] Hilchenbach, B., Strnadl, C.F.: Software AG reference architecture - capabilities view. https://www.softwareag.cloud/site/reference-architecture/software-ag-reference-architecture.html (2020), last accessed April, 6th, 2020

[22] Hodges, W.: Model Theory. Encyclopedia of Mathematics and its Applications, Cambridge University Press, Cambridge, MA (1993)

[23] ISO: ISO/IEC/IEEE 42010:2011: Systems and software engineering — architecture description. Standard, International Standards Organization, Geneva, CH (2011)

[24] Lankhorst et al., M.: Enterprise Architecture at Work. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-53933-0

[25] Leinster, T.: Basic Category Theory., Cambridge Studies in Advanced Mathematics, vol. 143. Cambridge University Press, Cambridge, UK (2014)

[26] López-Sanz, M., Vara, J.M., Marcos, E., Cuesta, C.E.: A model-driven approach to weave architectural styles into service-oriented architectures. In: Proceedings of the First International Workshop on Model Driven Service Engineering and Data Quality and Security. p. 53–60. MoSE+DQS ’09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1651415.1651426

[27] Mitchell, W.J.: The Logic of Architecture. The MIT Press, Cambridge, MA (1990)

[28] OMG: OMG Object Constraint Language. Standard, The Object Management Group (2014), http://www.omg.org/spec/OCL/2.4

[29] Peng, J., Abdi, S., Gajski, D.: Automatic model refinement for fast architecture exploration [soc design]. In: Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15th International Conference on VLSI Design. pp. 332–337. IEEE (2002). https://doi.org/10.1109/ASPDAC.2002.994944

[30] Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, Reading, MA (1984)

[31] Strnadl, C.F.: Aligning business and IT: The process-driven architecture model. IS Management 23(4), 67–77 (2006)

[32] The Object Management Group: OMG Unified Modeling Language. Standard, The Object Management Group (2017), https://www.omg.org/spec/UML/

[33] The Object Management Group: OMG Systems Modeling Language. Standard 1.6, The Object Management Group (2019), https://www.omg.org/spec/SysML/1.6/

[34] The Open Group: The TOGAF Standard Version 9.2. Standard, The Open Group (2018)
[35] The Open Group: Archimate 3.1 Specification. Standard 3.1, The Open Group (2019), https://pubs.opengroup.org/architecture/archimate3-doc/

[36] Wermelinger, M., Fiadeiro, J.L.: Algebraic software architecture re-configuration. In: Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 393–409. ESEC/FSE-7, Springer-Verlag, Berlin, Heidelberg (1999)

[37] Whitehead, A.N., Russell, B.: Principia Mathematica. Cambridge University Press, Cambridge, UK (1910-1913)

[38] Wilkinson, M.K., Dickerson, C.E., Ji, S.: Concepts of architecture, structure and system (2018), https://arxiv.org/abs/1810.12265

[39] Wittgenstein, L.: Logisch-philosophische Abhandlung. Wilhelm Ostwalds Annalen der Naturphilosophie 4(3-4) (1921)

[40] Yanchuk, A., Alexander, I., Maurizio, M.: Towards a mathematical foundation for service-oriented applications design. J. Software 1(1), 32–39 (2006). https://doi.org/10.4304/jsw.1.1.32-39