A Model of Layered Architectures

Diego Marmsoler Alexander Malkis Jonas Eckhardt
Technische Universität München
Germany

Architectural styles and patterns play an important role in software engineering. One of the most known ones is the layered architecture style. However, this style is usually only stated informally, which may cause problems such as ambiguity, wrong conclusions, and difficulty when checking the conformance of a system to the style. We address these problems by providing a formal, denotational semantics of the layered architecture style. Mainly, we present a sufficiently abstract and rigorous description of layered architectures. Loosely speaking, a layered architecture consists of a hierarchy of layers, in which services communicate via ports. A layer is modeled as a relation between used and provided services, and layer composition is defined by means of relational composition. Furthermore, we provide a formal definition for the notions of syntactic and semantic dependency between the layers. We show that these dependencies are not comparable in general. Moreover, we identify sufficient conditions under which, in an intuitive sense which we make precise in our treatment, the semantic dependency implies, is implied by, or even coincides with the reflexive-transitive closure of the syntactic dependency. Our results provide a technology-independent characterization of the layered architecture style, which may be used by software architects to ensure that a system is indeed built according to that style.

1 Introduction

Lack of discipline is a substantial technical source of failures in a number of software product lines [7, 15] (while other sources as, e.g., bad management, also exist). A poor architecture can result in a disaster for the whole project [10], hence, "expanding formal relationships between architectural design decisions and quality attributes" [19] has been identified as a promising future direction to go for the field. We address the lack of discipline in architectural design [4] by providing a formal model for one of the most important architectural styles, namely, the layered architecture style, which is also known as the virtual machines style.

While this work contributes to a rigorous theory of architecture styles, we believe that it has also implications for the practicing architecture researcher and the prospective software architect. The software architecture researcher can rely on a mathematical model when working with styles, while the prospective architect is provided with a solid foundation for her/his work. A theory of styles would provide the architect with a set of properties which allows her/him to decide whether a system is actually built according to a specific style, in our case, the layered architecture style. Moreover, the outcome of the analysis would provide the architect with a set of properties she/he can rely on from a system built according to a style, for example, semantic independence of lower-level layers from upper-level layers for systems built according to the layered architecture style.

1.1 Approach

In previous work [17], we describe an approach to formalize architectural styles. Based on the insight that each style requires its own semantic domain [1], this approach roughly follows three main steps:

B. Buhnova, L. Happe, J. Kofroň (Eds.):
Formal Engineering approaches to Software
Components and Architectures (FESCA'15).
EPTCS 178, 2015, pp. 47–61, doi:10.4204/EPTCS.178.5

- Find a mathematical model which reflects the nature of the style. This is probably the most difficult part, since the model must reflect the fundamental characteristics of a style. It should be as abstract as possible to allow the results of later analyses to be applied to a broad range of systems. If for some style an adequate model already exists, this step can be skipped.
- Provide a set of axioms for the model which constrain its structure. Through the addition of new axioms it is possible to specialize a style and investigate variations thereof. For example, in the layered architecture style, a configuration is usually isomorphic to a directed acyclic graph. However, we could add an axiom which restricts configuration to a directed sequence of layers to get a description of the strict version of the style.
- Finally, we can analyze a style by means of mathematical proofs. We can state characteristic properties for a style and prove them from our model.

In the following we apply our approach to the layered architecture style as described in [9, 20, 22].

Our *major contributions* are:

- an abstract and nonetheless precise notion of a layer (for this moment, one can loosely think of a layer as a provider of services that uses some other services),
- a notion of a layered architecture configuration, which is a collection of layers connected via ports (detailed later in the paper),
- a denotational semantics of a layered architecture configuration,
- a model for updating a layer, i.e., changing its semantics,
- for a pair of layers of a layered architecture configuration, the notions of
  - a syntactic dependency and
  - a semantic dependency,
- examples on which the dependencies differ,
- the following (for now intuitively stated) link between the dependencies:
  - in any layered architecture configuration the semantic dependency implies the reflexive-transitive closure of the syntactic dependency,
  - in any so-called usable layered architecture configuration the semantic dependency is equivalent to the reflexive-transitive closure of the syntactic dependency.

2 Background and Related Work

Related work can roughly be categorized in three main areas: approaches to formalization of architectural styles, informal descriptions of the layered architecture style, and existing formal analyzes of architectural styles.

In analyzing architectural styles, our work is actually based on work regarding *approaches to formalization of architectural styles*. In our work, we follow an approach based on Abowd et al. [1]. In that work, the authors apply the general approach of denotational semantics to software architectures with the fundamental insight that each architectural style needs its own semantic model. On this basis, Allen [2] provides an architecture description language based on CSP [13] to allow the specification and analysis of architectural styles. A different, though related approach is provided by Moriconi et al. in [18]. There, the authors use first order logical theories to describe architectural styles and they suggest to use the concept of faithful interpretation mappings to relate different styles. In a third approach, Le Métayer in [16] proposes to describe architectures as graphs and architectural styles as graph grammars with the aid of analyzing architecture evolution. Finally, Bernardo et al. [5] propose the use of process algebras to formalize architectural types, which are weaker forms of architectural styles.

To build our model for layered architectures, we heavily rely on the intuition provided by *informal descriptions of the layered architecture style*. Some of the first documented descriptions of the style can be found in the work of Shaw and Garlan [20], where they identify a set of well-known styles observed in industry. Taylor et al. [22] elaborate on that work and distinguish between two kinds of layered architectures: the virtual machines style and the client-server style. Finally, there exists much literature from practicing architects documenting architectural styles and patterns. We consider [9] as one well-known representative of this kind of works.

While all such references provide the necessary background for our study, there is another line of research on *existing formal analyzes of architectural styles* which is closely related to our work. In [12], Garlan and Notkin provide a formal basis for the implicit-invocation architectural style. The signal-processing style is analyzed by Garlan and Norman in [11]. Moreover, we can find a formal description of the pipes-and-filters style in the work of Allen and Garlan [3] and in Broy's `Focus`-theory [8]. The Enterprise Java Beans architectural style is formally analyzed by Sousa and Garlan in [21]. In [18], the data-flow style is related to the pipes-and-filters style, the batch-sequential style, and the shared-memory style. The client-server style is described by Le Métayer in [16]. Finally, there are some formal analyzes of the layered architecture style. In [23], Zave and Rexford build a formal model of layered architectures and use the Alloy Analyzer [14] to analyze the style. Since their analysis concentrates on network-specific properties, it is a refinement of our model, thus, complementing our work. The work which is probably closest to our work is the one of Broy which provides a better understanding of the layered architecture style in [6].

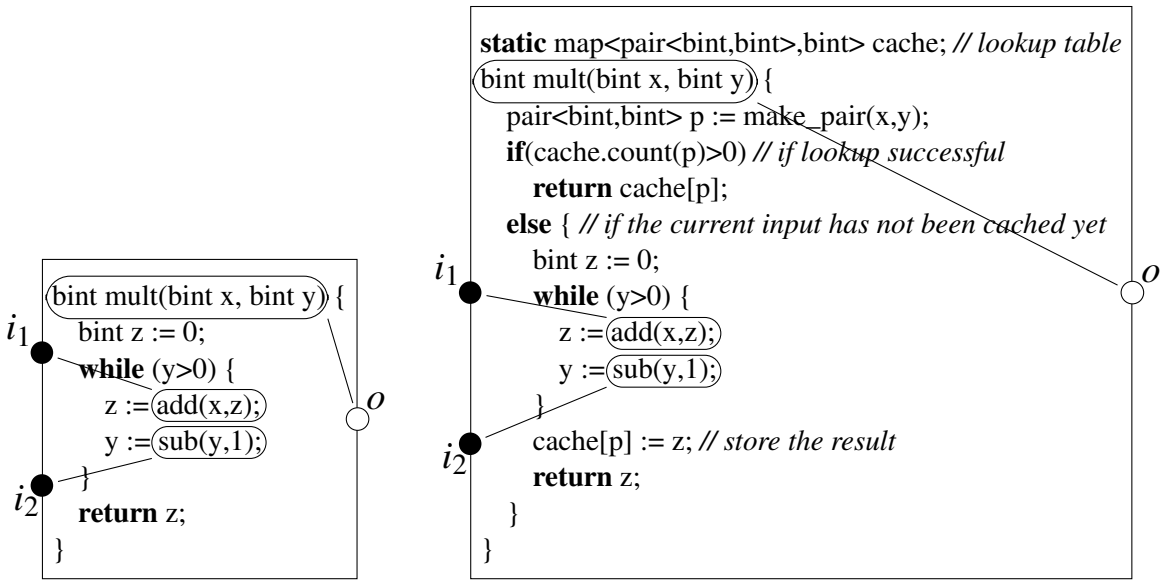
In [6], Broy provides a model of services and of layered architectures based on the `Focus` theory [8]. In that model, a layer is a component with an import and an export interface and a layered architecture is a stack of several layers. Although that model is an important contribution towards a better understanding of layered architectures, the model represents computations explicitly using streams. Our model abstracts further away from such details of computations, concentrating on the major characteristics of the style, thus making the results applicable to several, different representations of computations. In fact, our model is based on an abstract notion of a service, and streams are just one possible realization thereof as shown in Ex. 3.2. Other realizations include stateless services as shown in Ex. 3.1 and more complex interactions as shown in Ex. 3.3.

3 A Model for Layered Architectures

In the following section we provide a model of layered architectures based on ports and services. With this model we want to provide the basis for a rigorous analysis of the style. Therefore, the model should be as abstract as possible and capture the intuitive understanding of the style to allow formulation of characteristic properties of the style.

3.1 Ports and Services

For our model of layered architectures, we assume the existence of sets `PORT` and `SERVICE` which contain all ports and services, respectively. Thereby, our notion of service is rather abstract; A service can be anything, from a simple method to a complex web-service consisting of a series of interactions. A port is a placeholder for a set of related services; one can think of the method's signature or of the address of the web service. Thus, we assume the existence of a function $type\colon \texttt{PORT} \to \wp(\texttt{SERVICE})$, (where $\wp(X)$ is the power set of a set X) which assigns a type to each port. That is, the type of a port is simply a



(a) Stateless (b) Stateful

Figure 1: Stateful and stateless layers. The programming language type bint represents large integers.

set of services. We require that each port is classified either as an input port or as an output port, but not as both. We let $\mathscr{I}$ be the set of input ports and $\mathscr{O}$ be the set of output ports:

$$\mathscr{I} \cup \mathscr{O} = \texttt{PORT} \quad \text{and} \quad \mathscr{I} \cap \mathscr{O} = \emptyset.$$

Ports and services constitute the parameters of our theory. By saying what port and services are, our theory can be applied to different contexts.

In the following, let $\mathbb{N}^+$ be the set of positive integers, $\mathbb{Z}$ the set of all integers.

Example 3.1 (A model of stateless services)**.** Consider the code depicted in Fig. 1a, where we write bint for bigint, a programming language type of large but fixed-size integers. In this example, we define the set of input ports as $\mathscr{I} = \{i_1, i_2\}$, the set of output ports as $\mathscr{O} = \{o\}$, where the ports are signatures, i.e., simplifying, strings:

$i_1 =$ "`bint add(bint,bint)`", $i_2 =$ "`bint sub(bint,bint)`", $o =$ "`bint mult(bint,bint)`".

Here, a service at a port will be a (set-theoretic) map whose signature is given by the port. For the sake of the example, let us fix some $M \in \mathbb{N}^+$ and use $bint = [-2^M, 2^M - 1]$ for the set of representatives of integers modulo 2^{M+1}, writing $\widehat{a}$ for the representative of $a \in \mathbb{Z}$. The types of the ports are sets containing certain partial or total maps from $bint \times bint$ to $bint$:

- The type of i_1 is the singleton set containing exactly the modular addition, which is defined for all arguments.
- The type of i_2 is the set containing partial and total maps s whose result coincides with that of modular subtraction whenever their first argument is positive and the second is 1.
- The type of o is the set of total maps m that multiply the arguments x, y modulo 2^{M+1} whenever y is nonnegative. Such m must return some result also for negative y.

In this example, the types abstract away some details about termination and outcome and all details about the way the computations are performed. □

The above example does not use any global state. In a more complex model, a layer may also have an encapsulated state, as it is the case for object-oriented programming languages. We can easily encode stateful models by changing the notion of service to relate *streams* of concrete values of the input parameters to *streams* of concrete return values, as we will see in Ex. 3.2.

In the following, let $\mathbb{N}_0$ be the set of nonnegative integers and $\mathrm{dom}\, f$ the domain of a (partial) map f.

Example 3.2 (A model of stateful services)**.** The code from Ex. 3.1 is slow. For the sake of the example, let us assume that some calls to mult are often repeated with the same arguments so that caching would help reducing the running time, and let us cache every input-output pair in a simple way as in Fig. 1b. In the worst case the cache grows until the memory is exhausted, after which we assume that cache insertion and all later events may block or have an arbitrary behavior.

We assume that the cache operations are purely internal and that the cache can hold at least N input-output pairs. We define the set of input ports as $\mathscr{I} = \{i_1, i_2\}$ and the set of output ports as $\mathscr{O} = \{o\}$ again. Let $sbint = (\text{stream } bint) = bint^* \cup bint^\omega$ be the set of streams over $bint$, i.e., the set of finite and countably infinite sequences over $bint$, where we index the elements of a stream by the corresponding downward-closed subset of $\mathbb{N}_0$. We lift the previous types of i_1 and i_2 pointwise to streams as usual; e.g., $type(i_1) = \{a \in (sbint \times sbint \to sbint) \mid \forall r,s,t \in sbint\colon a(r,s) = t \Rightarrow (\mathrm{dom}\, t = (\mathrm{dom}\, r) \cap (\mathrm{dom}\, s) \wedge \forall i \in \mathrm{dom}\, t\colon t(i) = \widehat{r(i)+s(i)})\}$. We define $type(o)$ to be the set of all maps $m \in (sbint \times sbint \to sbint)$ such that whenever $m(r,s) = t$ for streams $r,s,t \in sbint$ and $i \in (\mathrm{dom}\, r) \cap (\mathrm{dom}\, s)$ is such that the number of cached entries $|\{(r(j),s(j)) \in bint^2 \mid j \leq i \wedge j \in (\mathrm{dom}\, r) \cap (\mathrm{dom}\, s)\}|$ is below N, then $i \in \mathrm{dom}\, t$ and we have $(\widehat{s(i)} \geq 0 \;\Rightarrow\; t(i) = \widehat{r(i)s(i)})$.

Loosely speaking, types containing functions over streams have just helped specifying stateful abstractions of stateful services without actually referring to their state spaces. □

Example 3.3 (A model of complex services)**.** In Ex. 3.2, a service is still realized by a simple method which depends on a global state. However, we could also think about models in which a service is actually realized by a series of method calls, coordinated by some kind of protocol. By adjusting the concrete notion of service, our theory can also be applied to those kind of models. Here, the behavior of the services relates streams of concrete values for all the input parameters of *all* the methods in the series with streams of output values of *all* the return values of the series. *Ports* are then a set of method signatures equipped with an expected order of execution. Again, an *output port* specifies methods which can be called within the layers implementation while an *input port* specifies those methods realized by a layer. □

3.2 Valuations

For a set of ports $P \subseteq \texttt{PORT}$, a valuation is a function from the set P to the set of services that respects the types of the ports. By $\overline{P}$ we denote the set of all valuations for P, formally,

$$\overline{P} = \prod_{p \in P} type(p)\,.$$

Sometimes, we shall use $[p_0, \ldots, p_n \mapsto S_0, \ldots, S_n]$ to denote a valuation of ports $p_0, \ldots, p_n$ with services $S_0, \ldots, S_n$, respectively. Formally,

$$[p_0, \ldots, p_n \mapsto S_0, \ldots, S_n] \quad = \quad \lambda p \in \{p_i \mid i \in \mathbb{N}_0 \wedge i \leq n\}. \begin{cases} S_0 & \text{if } p = p_0, \\ \vdots & \\ S_n & \text{if } p = p_n. \end{cases}$$

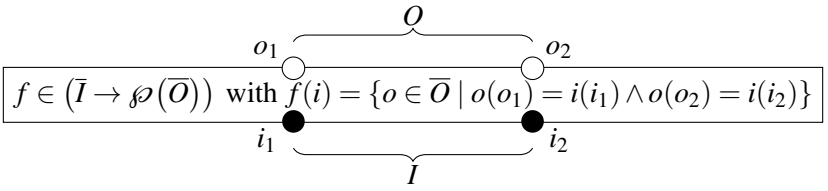


Figure 2: A layer with input-ports I, output-ports O and behavior function f.

Example 3.4 (Valuations for services)**.** In the models described in Ex. 3.1, 3.2, and 3.3, a *port valuation* just associates a services behavior with the corresponding method signature. □

3.3 Layers

Informally speaking, a layer consists of input ports, output ports, and some behavior that generates services at output ports from services at input ports. The behavior may be nondeterministic, so we represent it by a map that assigns a set of output-port valuations to every input-port valuation.

Definition 3.5. A *layer* is a triple (I,O,f), where $I \subseteq \mathscr{I}$, $O \subseteq \mathscr{O}$, and $f\colon \bar{I} \to \wp(\overline{O})$.

For a layer $l = (I,O,f)$, we denote by $l.in$ its input-ports I, by $l.out$ its output-ports O, and by $l.fun$ its behavior function f. We denote the set of all layers by $\mathscr{L}$.

Example 3.6 (A simple layer)**.** Consider, for example, the layer depicted in Fig. 2, which just copies i_j to o_j for $j \in \{1,2\}$. In our model, such a layer is represented as a triple (I,O,f) with input-ports $I = \{i_1,i_2\}$, output-ports $O = \{o_1,o_2\}$ and behavior function $f \in (\bar{I} \to \wp(\overline{O}))$ with $f(i) = \{o \in \overline{O} \mid o(o_1) = i(i_1) \wedge o(o_2) = i(i_2)\}$. □

3.4 Layered Architecture Configuration

A layered architecture configuration consists of a set of layers and an attachment describing the connections between the layers. Thus, a layered architecture configuration is modeled as a pair of a set of layers and a so-called attachment relation describing which output ports of which layers convey services to which input ports of which layers.

In the following, we denote by

$$X \dashrightarrow Y \quad = \quad \{f \subseteq X \times Y \mid \forall x,y_1,y_2\colon ((x,y_1) \in f \wedge (x,y_2) \in f) \Rightarrow y_1 = y_2\}$$

the set of partial maps from a set X to a set Y.

Definition 3.7. A *layered architecture configuration* is a pair (L,A), where $L \subseteq \mathscr{L}$ and $A \in ((\bigcup_{l\in L} l.in) \dashrightarrow (\bigcup_{l\in L} l.out))$, called the *attachment*, are such that the following constraints hold.

- Different layers do not share any ports, formally:

$$\forall\, k,l \in L\colon \quad k = l \;\vee\; (k.in \cup k.out) \cap (l.in \cup l.out) = \emptyset.$$

- If a service is provided at an output port that is connected to an input port, the layer owning the input port must be able to employ the service, i.e. the port types are compatible. Formally:

$$\forall\, (p_i,p_o) \in A\colon \quad type(p_o) \subseteq type(p_i).$$

For a layered architecture configuration $c = (L,A)$, we denote the set of layers L by $c.l$ and the attachment relation A by $c.conf$.

The domain of the attachment is a subset of the occurring input-ports, and the range is a subset of the occurring output-ports, signifying that the input ports are connected to the output ports. The attachment is a partial map, since not necessarily all input ports are internally connected, but whenever an input port is connected, it accepts services only from one output port.

Example 3.8 (A simple layered architecture configuration)**.** Fig. 3 shows a layered architecture configuration $c = (L,A)$. The first component of the layered architecture configuration describes the layers, i.e., their input and output-ports and their behavior function. In this example $L = \{l_0, \ldots, l_n\}$, where $l_k = (I_k, O_k, f_k)$ with $I_k = \{i_{0,k}, i_{1,k}, i_{2,k}\}$ for $0 < k \leq n$ and $I_0 = \{i_{0,0}\}$, $O_k = \{o_{0,k}, o_{1,k}, o_{2,k}\}$ for $0 \leq k < n$ and $O_n = \{o_{0,n}\}$, and $f_k \in \left(\overline{I_k} \to \wp\left(\overline{O_k}\right)\right)$ for $0 \leq k \leq n$.

The second component of the layered architecture configuration describes the attachment relation A which relates $i_{1,k}$ with $o_{1,k+1}$ and $i_{2,k}$ with $o_{2,k+1}$: $A = \{(i_{j,k}, o_{j,k-1}) \mid j \in \{1,2\} \wedge k \in \{1,\ldots,n\}\}$. □

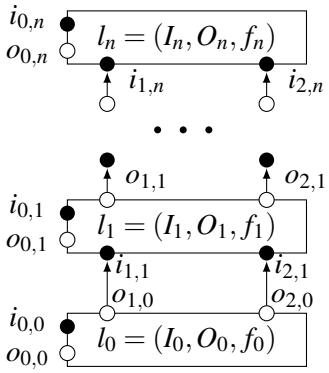


Figure 3: A layered architecture configuration with $n+1$ layer instances.

3.4.1 Selection and Projection

To facilitate reasoning about layered architecture configurations, in the following we introduce two kind of operators: selection and projection operators.

A selection operator allows to access ports belonging to a layered architecture configuration.

Definition 3.9. For a layered architecture configuration c, we define port selection as follows:

$$\Pi_i(c) = \bigcup_{l \in c.l} l.in \text{ and } \Pi_o(c) = \bigcup_{l \in c.l} l.out.$$

To select all ports of a layered architecture configuration, we just write

$$\Pi(c) = \Pi_i(c) \cup \Pi_o(c). \tag{D 1}$$

To select only the *open input-ports* (input ports which are not attached) of a layered architecture configuration c, we write

$$\Pi_{in}(c) \;:=\; \Pi_i(c) \setminus \mathrm{dom}(c.conf). \tag{D 2}$$

A projection operator, on the other hand, allows to access layers of a layered architecture configuration based on their ports.

Definition 3.10. Given a layered architecture configuration c and a port $p \in l.in \cup l.out$ for some $l \in c.l$, we define the layer projection

$$\sigma_p(c) = l.$$

By Def. 3.7, the layer possessing a given port is unique, so $\sigma_\cdot(.)$ is well-defined.

3.5 Semantics

Now we are going to define the computational meaning of a layered architecture configuration.

In the following, for a map $f: X \to Y$, we write $f|_Z$ for the restriction of f to the domain $X \cap Z$.

Definition 3.11. For a layered architecture configuration c, the *attachment-closure* $l.out^*$ of the output ports of a layer l is

$$\begin{aligned} l.out^* = \bigcap \big\{ & P \subseteq \Pi(c) \mid l.out \subseteq P && (1)\\ & \wedge (\forall (i,o) \in c.conf : i \in P \Rightarrow o \in P) && (2)\\ & \wedge (\forall o \in \Pi_o(c) : o \in P \Rightarrow \sigma_o(c).in \subseteq P) \big\}. && (3) \end{aligned}$$

The *configuration semantics* of a layer $l \in c.l$ is a function $[\![l]\!]_c : \overline{\Pi_{in}(c)} \to \wp\big(\overline{l.out}\big)$, with

$$\begin{aligned} [\![l]\!]_c(\mu) \;=\; \big\{ & \nu|_{l.out} \mid \nu \in \overline{l.out^*} && (4)\\ & \wedge\ \mu|_{l.out^*} = \nu|_{\Pi_{in}(c)} && (5)\\ & \wedge\ (\forall i \in \Pi_i(c) \cap l.out^* : (\nu(i) = \nu(c.conf(i)))) && (6)\\ & \wedge\ (\forall o \in \Pi_o(c) \cap l.out^* : \\ & \quad \exists \xi \in \sigma_o(c).fun(\nu|_{\sigma_o(c).in}) : \xi|_{l.out^*} = \nu|_{\sigma_o(c).out}) \big\} && (7) \end{aligned}$$

In (4), we would not like to use all the $\Pi(c)$ instead of $l.out^*$, since, informally speaking, there might be no consistent valuation of all the ports, but there may be a consistent valuation of a subset of ports that is sufficient to define the output of the layer. Instead we use the minimal set of ports including $l.out$ and closed under the attachment relation.

Each element of the semantics $[\![l]\!]_c(\mu)$ is created by constructing a valuation ν of the ports $l.out^*$ of the configuration that are needed for getting the value of the output ports of l and projecting ν to these output ports. In fact, line (4) says that ν provides a valuation of all needed ports. Line (5) says that the valuation of an open input-port must be taken into account if and only if we need this port. Line (6) says that if we require the value of a connected input-port, then we use the value of the corresponding output-port. Line (7) says that if we need a service provided by a layer, then the computation proceeds according to the layer's behavior function.

Example 3.12 (Calculating a layer's configuration semantics)**.** Consider, for example, the layered architecture configuration $c = (\{l_f, l_g\}, A)$ in Fig. 4a.
Here, $l_f = (\{i_0, i_0'\}, \{o_0, o_0'\}, f)$ and $l_g = (\{i_1, i_1'\}, \{o_1, o_1'\}, g)$ with $\{i_0, i_0', i_1, i_1'\} \subseteq \mathscr{I}$, $\{o_0, o_0', o_1, o_1'\} \subseteq \mathscr{O}$, and $A = \{(i_0', o_1'), (i_1', o_0')\}$.

For the sake of this example, let's assume that $\{B, C, D, F, X, Y\} \subseteq \texttt{SERVICE}$ and $type(i_0) = \{B\}$, $type(i_0') = type(o_1') = \{X, Y\}$, $type(i_1) = \{D\}$, $type(i_1') = type(o_0') = \{X, Y\}$, and $type(o_0) = type(o_1) = \{C, F\}$. Here, we use symbols B, C, D, F for services at externally visible ports (i_0, o_0, i_1, o_1) and X, Y for services at internal ports (i_0', o_0', i_1', o_1').

(a) Before update. (b) After update.

Figure 4: Layered architecture configuration consisting of two layers l_f and l_g.

The behavior functions are as follows:

$$\begin{array}{lll} f\colon \overline{\{i_0, i'_0\}} \to \wp\left(\overline{\{o_0, o'_0\}}\right), & [i_0, i'_0 \mapsto B, X] & \mapsto \{[o_0, o'_0 \mapsto C, X]\}, \\ & [i_0, i'_0 \mapsto B, Y] & \mapsto \{[o_0, o'_0 \mapsto F, X]\}; \\ g\colon \overline{\{i_1, i'_1\}} \to \wp\left(\overline{\{o_1, o'_1\}}\right), & [i_1, i'_1 \mapsto D, X] & \mapsto \{[o_1, o'_1 \mapsto C, X]\}, \\ & [i_1, i'_1 \mapsto D, Y] & \mapsto \{[o_1, o'_1 \mapsto F, Y]\}. \end{array}$$

Let us now apply Def. 3.11 to calculate $[\![l_f]\!]_c(\mu) \subseteq \overline{\{o_0, o'_0\}}$ and $[\![l_g]\!]_c(\mu) \subseteq \overline{\{o_1, o'_1\}}$ for $\mu = [i_0, i_1 \mapsto B, D]$. Therefore, we first calculate all elements $\nu \in \overline{\Pi(c)}$. For our simple system, ν satisfies

$$\begin{array}{ll} \nu(i_0) = \mu(i_0) = B, & \nu(i_1) = \mu(i_1) = D, \\ \nu(i'_0) = \nu(o'_1) = X, & \nu(i'_1) = \nu(o'_0) = X, \\ \nu(o_0) = C, & \nu(o_1) = C. \end{array}$$

Note that ν is the only element of $\overline{\Pi(c)}$ that satisfies the constraints that should hold for each element of $[\![.]\!]_c(\mu)$ according to Def. 3.11. Thus,

$$\begin{array}{ll} [\![l_f]\!]_c(\mu) & = \{\nu|_{\{o_0, o'_0\}}\} = \{[o_0, o'_0 \mapsto C, X]\} \text{ and} \\ [\![l_g]\!]_c(\mu) & = \{\nu|_{\{o_1, o'_1\}}\} = \{[o_1, o'_1 \mapsto C, X]\}. \end{array}$$

□

3.6 Semantic Change

A key concept in developing a piece of software is changing the semantics of a layer. We model such a change of the semantics of a layer through an update function.

Definition 3.13. For a layer l and a map $f\colon \overline{l.in} \to \wp\left(\overline{l.out}\right)$, a *semantic update* $[l \mapsto f]$ is the layer $(l.in, l.out, f)$.

Note that a semantic update is indeed a layer according to Def. 3.5.

The notion of semantic update easily generalizes to sets of layers $L \subseteq \mathscr{L}$:

$$L[l \mapsto f] = (L \setminus l) \cup \{[l \mapsto f]\}. \tag{D 3}$$

Finally, it also generalizes to layered architecture configurations:

$$c[l \mapsto f] = (c.l[l \mapsto f], c.conf). \tag{D 4}$$

Example 3.14 (A semantic update for a layered architecture configuration)**.** Consider, for example, the layered architecture configuration $c = (L,A)$ depicted in Fig. 4a and described in Ex. 3.12.

If we change the behavior of layer l_f to

$$f' \colon \overline{\{i_0, i_0'\}} \to \wp\left(\overline{\{o_0, o_0'\}}\right), \quad \begin{array}{l}[i_0, i_0' \mapsto B, X] \mapsto \{[o_0, o_0' \mapsto C, X]\},\\ [i_0, i_0' \mapsto B, Y] \mapsto \{[o_0, o_0' \mapsto F, Y]\},\end{array}$$

we get a new layered architecture configuration $c\,[l_f \mapsto f']$ where layer l_f has changed to (I_f, O_f, f') (see Fig 4b). Applying Def. 3.11 to calculate $[\![l_f]\!]_{c[l_f \mapsto f']}(\mu) \subseteq \overline{\{o_0, o_0'\}}$ and $[\![l_g]\!]_{c[l_f \mapsto f']}(\mu) \subseteq \overline{\{o_1, o_1'\}}$ for $\mu = [i_0, i_1 \mapsto B, D]$, produces, in addition to ν, a new valuation ν', which satisfies

$$\begin{array}{ll}\nu'(i_0) = \mu(i_0) = B\,, & \nu'(i_1) = \mu(i_1) = D\,,\\ \nu'(i_0') = \nu'(o_1') = Y\,, & \nu'(i_1') = \nu'(o_0') = Y\,,\\ \nu'(o_0) = F\,, & \nu'(o_1) = F\,.\end{array}$$

Note that ν' satisfies the constraints that should hold for each element of $[\![.]\!]_{c[l_f \mapsto f']}(\mu)$ according to Def. 3.11 and that ν, ν' are now the only elements of $\overline{\Pi(c)}$ which do so. Thus,

$$\begin{array}{ll}[\![l_f]\!]_{c[l_f \mapsto f']}(\mu) & = \{\nu|_{\{o_0, o_0'\}}, \nu'|_{\{o_0, o_0'\}}\} = \{[o_0, o_0' \mapsto C, X], [o_0, o_0' \mapsto F, Y]\} \text{ and}\\ [\![l_g]\!]_{c[l_f \mapsto f']}(\mu) & = \{\nu|_{\{o_1, o_1'\}}, \nu'|_{\{o_1, o_1'\}}\} = \{[o_1, o_1' \mapsto C, X], [o_1, o_1' \mapsto F, Y]\}\,.\end{array}$$

□

In the above example as well as in general, a semantic update of a layered architecture configuration changes neither the input/output-ports nor the attachment, thus producing a layered architecture configuration again:

Proposition 3.15. *For a layered architecture configuration c, layer $l \in c.l$, and a map $f \colon \overline{l.in} \to \wp\left(\overline{l.out}\right)$, the layered architecture configuration update $c\,[l \mapsto f]$ is a layered architecture configuration.*

Thus, all properties and notation introduced so far for layered architecture configurations are also valid for layered architecture configuration updates.

3.7 Syntactic Dependency

In a layered architecture configuration, the attachment relation induces a dependency relation between layers. We say that a layer l' *syntactically* depends on another layer l, if an input port of l' is connected to an output port of l.

Definition 3.16. Syntactic dependency for a layered architecture configuration c is a relation $\prec_c \subseteq c.l \times c.l$ defined by

$$l \prec_c l' \quad \overset{\text{def}}{\Leftrightarrow} \quad \exists\, o \in l.out,\ i \in l'.in \colon \quad o = c.conf\,(i)\,.$$

Example 3.17 (Syntactic dependency)**.** In the layered architecture configuration depicted in Fig. 3, we have $l_i \prec_c l_{i+1}$ for $i \in \{0, \ldots, n-1\}$ and no other syntactic dependencies. □

For a layered architecture configuration c, we denote by $\prec_c^+$ the *transitive* closure of $\prec_c$ and by $\prec_c^*$ the *reflexive-transitive* closure of $\prec_c$. Moreover, we denote by $\prec_c \_ \colon c.l \to \wp(c.l)$, defined via

$$\prec_c m = \{l \in c.l \mid l \prec_c m\} \qquad \text{for } m \in c.l\,, \tag{D 5}$$

all layers l that a given layer m syntactically depends on ($\prec_c^+ \_$, $\prec_c^* \_$ for [reflexive-] transitive dependency, respectively).

Lemma 3.18. *For a layered architecture configuration c, and layer $l \in c.l$, the attachment closure $l.out^*$ contains only ports of layers on which layer l reflexively-transitively syntactically depends on. Formally, $\forall p \in l.out^* \colon \sigma_p(c) \prec_c^* l$.*

Proof. Let $f \colon \wp(\Pi(c)) \to \wp(\Pi(c))$,
$P \mapsto l.out \cup \{o \mid \exists i \in P \colon (i,o) \in c.conf\} \cup \bigcup \{r.in \mid r \in c.l \wedge P \cap r.out \neq \emptyset\}$.
Using the fixed point theorem of Tarski one can show that $l.out^* = \bigcup_{n \in \mathbb{N}_0} f^n(\emptyset)$. Fix a layered architecture configuration c and one of its layers $l \in c.l$. We show that $\forall n \in \mathbb{N}_0 \; \forall p \in f^n(\emptyset) \colon \sigma_p(c) \prec_c^* l$ by induction on i.
"$\forall p \in f^0(\emptyset) \colon \sigma_p(c) \prec_c^* l$": Since $f^0(\emptyset) = \emptyset$, the statement is vacuously true.
"$\forall p \in f^n(\emptyset) \colon \sigma_p(c) \prec_c^* l$ implies $\forall p \in f^{n+1}(\emptyset) \colon \sigma_p(c) \prec_c^* l$": Fix $p \in f^{n+1}(\emptyset)$. At least of the following cases is true.

Case $p \in l.out$: By Def. 3.10, $\sigma_p(c) = l$ and by reflexivity, $l \prec_c^* l$. Thus, $\sigma_p(c) \prec_c^* l$.

Case $p \in \{o \mid \exists i \in f^n(\emptyset) \colon (i,o) \in c.conf\}$: Then there is an $i \in f^n(\emptyset)$ such that $(i,p) \in c.conf$. By Def. 3.10 and 3.7 we have $i \in \sigma_i(c).in$ and $p \in \sigma_p(c).out$. From $(i,p) \in c.conf$ we obtain $\sigma_p(c) \prec_c \sigma_i(c)$ by Def. 3.16. Since $i \in f^n(\emptyset)$, we have $\sigma_i(c) \prec_c^* l$ by induction hypothesis. By transitivity, $\sigma_p(c) \prec_c^* l$.

Case $p \in \bigcup \{r.in \mid r \in c.l \wedge f^n(\emptyset) \cap r.out \neq \emptyset\}$: Then there is an $r \in c.l$ such that $f^n(\emptyset) \cap r.out \neq \emptyset$ and $p \in r.in$. Since $f^n(\emptyset) \cap r.out \neq \emptyset$, we have $r \prec_c^* l$ by induction hypothesis. Since $p \in r.in$, we have $\sigma_p(c) = r$ by Def. 3.10. Thus, we conclude $\sigma_p(c) \prec_c^* l$. □

Note that the syntactic dependency relation is not transitive in general: just because a layer L_1 depends on another layer L_2 which depends on a third layer L_3, this does not necessarily mean that layer L_1 depends on layer L_3.

3.8 Semantic Dependency

Besides the syntactic dependency relation between layers of a layered architecture configuration we also have a semantic dependency relation between those layers. A layer l' semantically depends on a layer l if updating l may influence the configuration semantics of l'.

Definition 3.19. Semantic dependency for a layered architecture configuration c is a relation $\ll_c \subseteq c.l \times c.l$ defined by

$$l \ll_c l \overset{\text{def}}{\Leftrightarrow} \exists f \in \left(\overline{l.in} \to \wp\left(\overline{l.out}\right)\right) : [\![l]\!]_c \neq [\![[l \mapsto f]]\!]_{c[l \mapsto f]} \qquad \text{for all } l \text{ in } c.l \text{ and}$$
$$l \ll_c l' \overset{\text{def}}{\Leftrightarrow} \exists f \in \left(\overline{l.in} \to \wp\left(\overline{l.out}\right)\right) : [\![l']\!]_c \neq [\![l']\!]_{c[l \mapsto f]} \qquad \text{for all } l \neq l' \text{ in } c.l.$$

Now we provide simple examples of semantic dependency and independence.

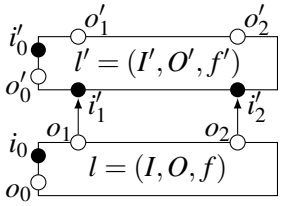


Figure 5: A simple layered architecture configuration with 2 layers.

Example 3.20 (Semantic dependency)**.** As an example, consider the simple layered architecture configuration depicted in Fig. 5 where changing the behavior of layer l does indeed influence the configuration semantics of layer l'.

In order to see this, we first need to formally define the behavior functions $f \in \left(\overline{I} \to \wp\left(\overline{O}\right)\right)$ and $f' \in \left(\overline{I'} \to \wp\left(\overline{O'}\right)\right)$. Let us assume that $\{A,B,C,D,E,F,X,Y\} \subseteq \texttt{SERVICE}$ and $type(i_0) = \{A\}$, $type(o_0) = \{B\}$, $type(o_1) = type(i'_1) = \{X\}$, $type(o_2) = type(i'_2) = \{Y,Z\}$, $type(i'_0) = \{C\}$, $type(o'_0) = \{D\}$, $type(o'_1) = \{E\}$, and $type(o'_2) = \{F,G\}$. Here, we use symbols A,B,C,D,E,F for services occurring at externally visible ports $(i_0,o_0,i'_0,o'_0,o'_1,o'_2)$ and X,Y for services occurring at internal ports (o_1,o_2,i'_1,i'_2).

Let $f : \overline{\{i_0\}} \to \wp\left(\overline{\{o_0,o_1,o_2\}}\right)$ and $f' : \overline{\{i'_0,i'_1,i'_2\}} \to \wp\left(\overline{\{o'_0,o'_1,o'_2\}}\right)$ be defined by

$$\begin{aligned} f\left([i_0 \mapsto A]\right) &= \{[o_0,o_1,o_2 \mapsto B,X,Y]\}\,, \\ f'\left([i'_0,i'_1,i'_2 \mapsto C,X,Y]\right) &= \{[o'_0,o'_1,o'_2 \mapsto D,E,F]\}\,, \\ f'\left([i'_0,i'_1,i'_2 \mapsto C,X,Z]\right) &= \{[o'_0,o'_1,o'_2 \mapsto D,E,G]\}\,. \end{aligned}$$

Now we calculate $[\![l]\!]_c : \overline{\{i_0,i'_0\}} \to \wp\left(\overline{\{o_0,o_1,o_2\}}\right)$ and $[\![l']\!]_c : \overline{\{i_0,i'_0\}} \to \wp\left(\overline{\{o'_0,o'_1,o'_2\}}\right)$ by Def. 3.11:

$$\begin{aligned} [\![l]\!]_c\left([i_0,i'_0 \mapsto A,C]\right) &= \{[o_0,o_1,o_2 \mapsto B,X,Y]\}\,, \\ [\![l']\!]_c\left([i_0,i'_0 \mapsto A,C]\right) &= \{[o'_0,o'_1,o'_2 \mapsto D,E,F]\}\,. \end{aligned}$$

If we now replace f by $g : \overline{\{i_0\}} \to \wp\left(\overline{\{o_0,o_1,o_2\}}\right)$, defined as

$$g\left([i_0 \mapsto A]\right) = \{[o_0,o_1,o_2 \mapsto B,X,Z]\}\,,$$

we can see that $l \ll_c l'$, because calculating $[\![l]\!]_{c[l \mapsto g]} : \overline{\{i_0,i'_0\}} \to \wp\left(\overline{\{o_0,o_1,o_2\}}\right)$ and $[\![l']\!]_{c[l \mapsto g]} : \overline{\{i_0,i'_0\}} \to \wp\left(\overline{\{o'_0,o'_1,o'_2\}}\right)$ by Def. 3.11 results in

$$\begin{aligned} [\![l]\!]_{c[l \mapsto g]}\left([i_0,i'_0 \mapsto A,C]\right) &= \{[o_0,o_1,o_2 \mapsto B,X,Z]\}\,, \\ [\![l']\!]_{c[l \mapsto g]}\left([i_0,i'_0 \mapsto A,C]\right) &= \{[o'_0,o'_1,o'_2 \mapsto D,E,G]\}\,. \end{aligned}$$

and we see that $[\![l']\!]_{c[l \mapsto g]} \neq [\![l']\!]$. □

Example 3.21 (Semantic independence)**.** In the simple layered architecture configuration in Fig. 5 changing the behavior of layer l' does not influence the configuration semantics of layer l.

Let us assume behavior functions $f : \overline{I} \to \wp\left(\overline{O}\right)$ and $f' : \overline{I'} \to \wp\left(\overline{O'}\right)$ of Ex. 3.20. Then we can see that there is no behavior function $g : \overline{I'} \to \wp\left(\overline{O'}\right)$ such that $[\![l]\!]_{c[l' \mapsto g]} \neq [\![l]\!]$. This is the case, because the semantics of l does not depend on any inputs from l'. Thus, we have $l' \not\ll_c l$. □

3.9 Relating Syntactic and Semantic Dependencies

Having a formal model of layered architecture configurations allows us to analyze the relationship between syntactic and semantic dependencies.

An interesting property is that if layers are syntactically dependent, this does not necessarily mean that they are also semantically dependent.

Example 3.22 (Syntactic dependency does not necessarily imply semantic dependency)**.** Consider a single layer with just one input and just one output port that are typed by the empty set of services and attached to each other. According to Def. 3.16, the layer depends on itself syntactically. However, it

is not possible to change the layers configuration semantics at all, since the layer's behavior function is the only map from the (empty) set of valuations of the input port to the (empty) set of valuations of the output port. Thus, according to Def. 3.19, the layer does not depend on itself semantically. In general, if $\texttt{SERVICE} = \emptyset$, any configuration with a nonempty attachment will have a pair of layers with this property. □

However, under certain circumstances, syntactic dependency does indeed imply semantic dependency.

Definition 3.23. A layered architecture configuration c is *usable* iff there is at least one valuation of open input-ports such that the configuration semantics of every layer produces at least one output valuation on this input. Formally:
c usable $\overset{\text{def}}{\iff} \exists\, \mu \in \overline{\Pi_{in}(c)}\ \forall\, l \in c.l\colon [\![l]\!]_c(\mu) \neq \emptyset$.

Theorem 3.24. *For a usable layered architecture configuration c the reflexive-transitive closure of syntactic dependency implies semantic dependency. Formally: c usable* $\Rightarrow \prec_c^* \subseteq \ll_c$.

Proof. Let c be usable and $l \prec_c^* l'$. So there is some $\mu \in \overline{\Pi_{in}(c)}$ such that $[\![l']\!]_c(\mu) \neq \emptyset$. Let $g : \overline{l.in} \to \wp\big(\overline{l.out}\big)$, $i \mapsto \emptyset$. If $l = l'$, then $[\![[l' \mapsto g]]\!]_{c[l' \mapsto g]}(\mu) = \emptyset$. If $l \neq l'$, we inductively follow that all the layers $r \neq l$ such that $l.out \cap r.out^* \neq \emptyset$ satisfy $[\![r]\!]_{c[l \mapsto g]}(\mu) = \emptyset$. In particular, $[\![l']\!]_{c[l \mapsto g]}(\mu) = \emptyset$. □

Vice versa, if layers are semantically dependent, they are not necessarily (directly) syntactically dependent.

Example 3.25 (Semantic dependency does not necessarily imply syntactic dependency)**.** Consider a single layer with one output port that is typed by two services and no other ports. According to Def. 3.19, it depends on itself semantically. However, according to Def. 3.16, it does not depend on itself syntactically. Indeed, it does not have any syntactic dependency at all.

A less trivial example is demonstrated in Fig. 6, where $\texttt{SERVICE} = \{A, B\}$, all ports are typed by SERVICE, $l.fun = \lambda\, \nu \in \overline{\emptyset}.\, \{[o \mapsto A]\}$, $l'.fun = \lambda\, \nu \in \overline{\{i'\}}.\, \{[o' \mapsto \nu(i')]\}$, and $l''.fun = \lambda\, \nu \in \overline{\{i''\}}.\, \{[o'' \mapsto \nu(i'')]\}$. We have $l \ll_c l''$, but $l \not\prec_c l''$. □

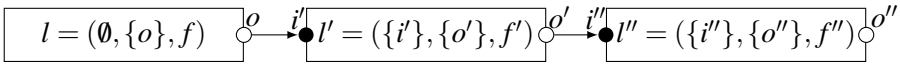


Figure 6: A three-layered configuration with $c.l = \{l, l', l''\}$ and $c.conf$ as shown.

As we see, changing the behavior of a single layer may impact not only the configuration semantics of directly depending layers, but also of layers which transitively depend on the modified layer.

This property of layered architecture configurations implies that a test after a change of a layers behavior should include tests of the behavior of all semantically dependent layers. As we will see in a moment, there is a bound on how many layers one should test.

Theorem 3.26. *Semantic dependency implies the reflexive-transitive closure of syntactic dependency. Formally:* $\ll_c \subseteq \prec_c^*$.

Proof. Fix a layered architecture configuration c, its layers $l, l' \in c.l$ such that $l \not\prec_c^* l'$; we will show $l \not\ll_c l'$.
Notice that $l \neq l'$. Fix arbitrary $f\colon \overline{l.in} \to \wp\big(\overline{l.out}\big)$ and $\mu \in \overline{\Pi_{in}(c)}$. We are going to show that $[\![l']\!]_c(\mu) = [\![l']\!]_{c[l \mapsto f]}(\mu)$.

"$\subseteq$": Let $\kappa \in [\![l']\!]_c(\mu)$. By (4) of Def. 3.11 there is some $\nu \in \overline{l.out^*}$ such that $\nu|_{l'.out} = \kappa$ and (5), (6), (7) hold for κ and l'. Since l' does not reflexively-transitively syntactically depend on l, by Lemma 3.18 no ports of l are in $l'.out^*$ and we readily conclude that (4), (5), (6), (7) still hold for κ and l' if c is replaced by $c[l \mapsto f]$. Thus $\kappa \in [\![l']\!]_{c[l\mapsto f]}(\mu)$.

"$\supseteq$": Analogously. □

Informally speaking, this property allows us now to restrict testing after a modification to only those layers which (reflexively-)transitively depend on the modified layer.

Corollary 3.27. *For usable layered architecture configurations, the semantic dependency and the reflexive-transitive closure of the syntactic dependency are the same.*

4 Conclusion

With this work we provided an abstract model for the layered architecture style. Our model is based on the notion of services and ports which can supply services. A layer consists of input and output ports and is modeled as a function from input-port valuations to output-port valuations. A layered architecture configuration consists then of some layer instances and an attachment describing the connections between layers' input and output ports.

We have given a formal definition of syntactic and semantic dependency between layers. Though syntactic and semantic dependencies do not necessarily imply one another, we have shown that the semantic dependency implies the reflexive-transitive closure of the syntactic dependency, and the reverse also holds for usable configurations.

Having developed a formal model of layered architectures, the model can now be used for a rigorous analysis of the style. Thus, future work arises in two main areas: (i) First of all, different variants of the style should be identified and defined through constraints over our model. For example, a "basic" variant of the style would impose a *well-foundedness* constraint on the attachment relation and a "strict" variant would further constrain the attachment relation to be *antitransitive*. (ii) Then, for each variant, a set of properties should be formulated and proved from the constraints. For example, in the "basic" variant, we may want to provide conditions that ensure that the configuration is usable. Moreover, the configuration semantics of lower level layers may be strictly independent of the behavior of upper level layers and under certain circumstances, the configuration semantics of upper level layers may also be independent of the behavior of lower level layers. In the "strict" version, changing a layers behavior may have even less impact on the configuration semantics of other layers within the architecture configuration.

Our work aims to contribute to a rigorous theory of architectural styles to provide a better understanding of architectural styles and the formal relationships between architectural design decisions and quality attributes. Thus, two further directions for future work arise: (i) The approach used in this article should be applied to other architectural styles as well. (ii) Then, a general theory of architectural styles should be developed to investigate relationships between the different styles.

5 Acknowledgments

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grants "Software Campus project RE4SoS, 01IS12057", and "ARAMiS project, 01IS11035".

We would like to thank Manfred Broy, Wolfgang Boehm, Maximilian Irlbeck, Maximilian Junker, Andreas Vogelsang, Vasileios Koutsoumpas, Veronika Bauer, and Daniel Méndez Fernández for their comments and helpful suggestions.

References

[1] Gregory D. Abowd, Robert Allen & David Garlan (1995): *Formalizing Style to Understand Descriptions of Software Architecture*. ACM Transactions on Software Engineering and Methodology, doi:10.1145/226241.226244.

[2] Robert Allen (1997): *A Formal Approach to Software Architecture*. Ph.D. thesis, Carnegie Mellon, School of Computer Science.

[3] Robert Allen & David Garlan (1992): *A Formal Approach to Software Architectures*. Proceedings of the IFIP 12th World Computer Congress.

[4] Len Bass, Paul Clements & Rick Kazman (2012): *Software Architecture In Practice*, 3rd edition. Pearson Education, Inc.

[5] Marco Bernardo, Paolo Ciancarini & Lorenzo Donatiello (2000): *On the formalization of architectural types with process algebras*. ACM SIGSOFT Software Engineering Notes, doi:10.1145/357474.355064.

[6] Manfred Broy (2005): *Service-Oriented Systems Engineering: Specification and Design of Services and Layered Architectures*. In: Eng. Theories of Software Intensive Systems, doi:10.1007/1-4020-3532-2_2.

[7] Manfred Broy (2011): *Can practitioners neglect theory and theoreticians neglect practice?* IEEE Computer, doi:10.1109/MC.2011.305.

[8] Manfred Broy & Ketil Stølen (2001): *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, doi:10.1007/978-1-4613-0091-5.

[9] Frank Buschmann, Kelvin Henney & Douglas Schimdt (2007): *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. Wiley.

[10] David Garlan (2000): *Software architecture: a roadmap*. In: ICSE 2000, doi:10.1145/336512.336537.

[11] David Garlan & Norman Delisle (1990): *Formal Specifications as Reusable Frameworks*. In: Proceedings of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development, doi:10.1007/3-540-52513-0_9.

[12] David Garlan & David Notkin (1991): *Formalizing Design Spaces: Implicit Invocation Mechanisms*. In: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development.

[13] Charles Antony Richard Hoare (1985): *Communicating sequential processes*. Prentice Hall.

[14] Daniel Jackson (2012): *Software Abstractions: logic, language, and analysis*. MIT Press.

[15] Pontus Johnson, Mathias Ekstedt & Ivar Jacobson (2012): *Where's the Theory for Software Engineering?* IEEE software, doi:10.1109/MS.2012.127.

[16] Daniel Le Métayer (1998): *Describing Software Architecture Styles using Graph Grammars*. IEEE Transactions on Software Engineering, doi:10.1109/32.708567.

[17] D. Marmsoler (2014): *Towards a Theory of Architectural Styles*. In: 22th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-22), pp. 823–825, doi:10.1145/2635868.2661683.

[18] Mark Moriconi, Xiaolei Qian & Robert A. Riemenschneider (1995): *Correct Architecture Refinement*. IEEE Transactions on Software Engineering, doi:10.1109/32.385972.

[19] Mary Shaw & Paul Clements (2006): *The golden age of software architecture*. Software, IEEE, doi:10.1109/MS.2006.58.

[20] Mary Shaw & David Garlan (1996): *Software architecture: perspectives on an emerging discipline*.

[21] Joao Pedro Sousa & David Garlan (2001): *Formal Modeling of the Enterprise JavaBeans Component Integration Framework*. Information and Software Technology, doi:10.1016/S0950-5849(00)00157-9.

[22] Richard N. Taylor, Nenad Medvidovic & Eric M. Dashofy (2010): *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons.

[23] Pamela Zave & Jennifer Rexford (2013): *Compositional Network Mobility*. In: Verified Software: Theories, Tools, Experiments - 5th International Conference, pp. 68–87, doi:10.1007/978-3-642-54108-7_4.