A Model of Service-Oriented Architectures

Alexander Malkis and Diego Marmsoler
Technische Universität München

***Abstract*—Architectural styles and patterns play an important role in software engineering. Over the last years, a new style based on the notion of services emerged, which we call the service-oriented architecture style. However, this style is usually only stated informally, which may cause inherent problems such as ambiguity, wrong conclusions, and the difficulty of checking the conformance of a system to the style. We address these problems by providing a formal, denotational semantics of the service-oriented architecture style and two variants thereof: the layered architecture style and the strict architecture style. Loosely speaking, in our model of the service-oriented architecture style, services are a means of communication. Components exchange services between each other via ports. The layered architecture variant imposes a well-foundedness constraint on the communication structure, while the strict variant imposes an antitransitivity constraint. We analyze the notions of syntactic and semantic dependencies for service-oriented architectures and investigate their relationship. Moreover, the expected informal properties of the styles are formulated as theorems. Finally, we present a method for soundly analyzing instances of the style. Our rigorous approach enables building higher-quality architectures, for which properties can be mathematically stated and proven, by enforcing formal discipline on the inter-component scale.**

I. Introduction

Lack of discipline is one substantial technical source of failures in a number of software projects [1], [2]. A poor architecture can lead to a disaster for the whole project [3], hence, "expanding formal relationships between architectural design decisions and quality attributes" [4] has been identified as a promising future direction to go for the field of software architectures. We address the lack of discipline in architectural design [5] by providing a formal model for an important architectural style, namely, the service-oriented architecture style.

While we aim at contributing to a rigorous theory of architecture styles, we believe that developments in this theory also have 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 obtains a solid foundation for working with concrete architecture instances at hand. A theory of styles would provide the architect with a set of properties which allows one to decide whether a system is actually built according to a specific style. (In our case, such specific styles are the service-oriented architecture style, its layered variant, or its strict variant.) Moreover, the outcome of the analysis would provide the architect with a set of properties one can trust in 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 variant.

Our *major contributions* are as follows:

- an abstract model of the service-oriented architecture style and some concrete instances thereof (§ IV).
- an in-depth analysis of the crucial notions of syntactic (§ IV-B2) and semantic (§ IV-C) dependencies between the components, including their interrelations.
- a fast method for proving properties of the semantics of the components based on overapproximating the exact semantics (§ IV-D).
- a rigorous definition of two common variants of service-oriented architectures: the layered and the strict variant (§ V).
- a rigorous analysis of the layered variant (§ V-A), showing the conditions under which certain components maintain their semantics after updating the behavior of one of the components.
- a rigorous analysis of the strict variant (§ V-B), showing that for the strict variant our fast proof method is in fact precise.

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. Therefore, we directly address the call to "expand the formal relationships between architectural design decisions and quality attributes" [4] and to disciplined architectural design [5].

II. Approach

In [6], we describe an approach to formalize architectural styles. Based on the insight that each style requires its own semantic domain [7], this approach roughly follows three main steps:

1) 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.
2) Provide a set of axioms for the model that 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 variant, 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.
3) 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 service-oriented style. According to step 1 of our approach, in [8] we first provided an abstract, nonetheless precise model of the style which is able to cope with different concrete service models. In this article we mainly address steps 2 and 3 by investigating different variants of the style. (Hereby, we also improve on the details of the semantics and adopt a more mainstream terminology in comparison to our prior investigation [8].)

III. Related Work

There is a large body of work on service-oriented architectures; we discuss next only a choice of the literature which is, subjectively, most related to our work. We roughly categorize related work in three main areas: approaches to formalization of architectural styles, informal descriptions of the layered architecture style, and existing formal analyses of architectural styles.

Concerning analyzing architectural styles, our work is situated among the *approaches to formalization of architectural styles*. Broadly speaking, we follow an approach based on Abowd et al. [7]. 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 [9] provides an architecture description language based on CSP [10] to allow the specification and analysis of architectural styles. Another related approach is developed by Moriconi et al. in [11]. There, the authors use first-order logical theories to describe architectural styles and so-called faithful interpretation mappings to relate different styles. In a further approach, Le Métayer in [12] proposes to describe architectures as graphs and architectural styles as graph grammars with the aid of analyzing architecture evolution. Additionally, an interesting approach applying category theory to formalize architectural concepts is provided by Fiadeiro in [13]. Finally, Bernardo et al. [14] 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 earliest documented descriptions of the style can be found in the work of Shaw and Garlan [15], where they identify a set of well-known styles observed in industry. Taylor et al. [16] elaborate on that work and distinguish between two kinds of layered architectures: the virtual machines style and the client-server style. Finally, practicing architects often report on styles and patterns they are commonly empolying themselves. We consider [17] as one well-known book of this kind.

While all such references provide the background for our study, there is another line of research on *existing formal analyses of architectural styles* which is closely related to our work. Below we shortly list a few representatives of such analyses. In [18], Garlan and Notkin provide a formal basis for the implicit-invocation architectural style. The signal-processing style is analyzed by Garlan and Norman in [19]. Moreover, we can find a formal description of the pipes-and-filters style in the work of Allen and Garlan [20] and in Broy's `Focus`-theory [21]. The Enterprise Java Beans architectural style is formally analyzed by Sousa and Garlan in [22]. In [11], the dataflow 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 [12]. Finally, there are some formal analyses of the layered architecture style. In [23], Zave and Rexford build a formal model of layered architectures and use the Alloy Analyzer [24] to analyze the style. In [25], Broy provides a model of services and of layered architectures based on the `Focus` theory [21].

Our work is probably closest to [25], where 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, namely 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. If fact, our model is based on an abstract notion of a service, and streams are just one possible realization thereof.

IV. A Model of Service-Oriented Architectures

In this chapter, we provide a model of service-oriented architectures based on ports and services. With this model, we offer the basis for a rigorous analysis of the style. First, in § IV-A, we provide basic definitions which lay the foundation for our model. In § IV-B these definitions are used to define architecture configurations, syntactic dependency, and the semantics of components. In § IV-C we define the notions of semantic dependency and investigate their interrelationships with the syntactic dependency. Finally, in § IV-D we provide a fast method to prove properties of a component's semantics.

A. Foundations

This section provides the foundation for our model by introducing several key concepts. We introduce the notion of ports and services in § IV-A1 and define the notion of port valuations in § IV-A2. Then we provide a rigorous definition for components in § IV-A3 and introduce selection and projection operators in § IV-A4.

1) Ports and Services

For our model of service-oriented 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 everything, from a simple procedure of a programming language to a complex web-service consisting of a series of interactions. A port is just a placeholder for a set of related services; one can think of the procedure's declaration (in the sense of a programming language) or of the address of the web service. Thus, we assume the existence of a function $\mathrm{type}\colon \texttt{PORT} \to \wp(\texttt{SERVICE})$, (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 set

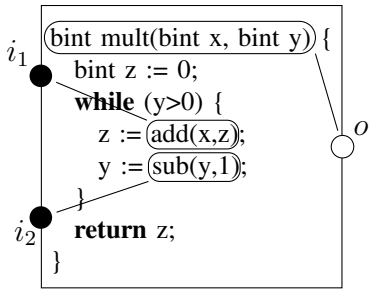


Figure 1: Stateless

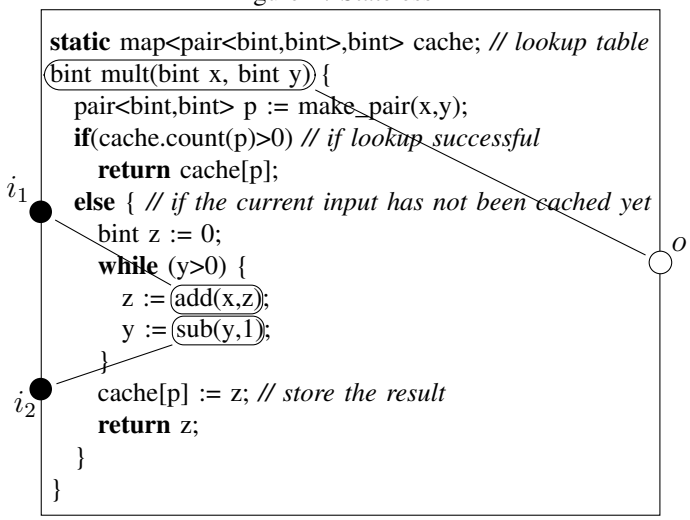


Figure 2: Stateful

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, such that:

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

Ports and services constitute the parameters of our theory. By saying what ports 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 IV.1 (Modeling stateless services)**.** Consider the code depicted in Fig. 1, where we write bint for bigint, a programming language type of large but fixed-size integers. 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 function declarations, i.e., simplifying, strings:
i_1 = "bint add(bint,bint)", i_2 = "bint sub(bint,bint)", o = "bint mult(bint,bint)".

Intuitively, a service at a port will be a (set-theoretic) map that fits the declaration given by the port. Formally, we fix some $M \in \mathbb{N}^+$ and let $bint = [-2^M, 2^M{-}1]$, which we use as the set of representatives of integers modulo 2^{M+1}. The types of the ports are sets containing certain partial or total maps from $bint{\times}bint$ to $bint$:

- $\mathrm{type}(i_1)$ is the singleton set containing exactly the modular addition, which is defined for all arguments.
- $\mathrm{type}(i_2)$ is the set containing partial and total maps whose result coincides with that of modular subtraction whenever the first argument is positive and the second is 1.
- $\mathrm{type}(o)$ 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.

If even more abstraction would be wished, we would redefine the above types as, e.g., the set of all partial and total maps from $bint{\times}bint$ to $bint$. In this alternative situation, the correctness of a service provided by a component (here mult) would depend on the correctness of the services required by a component (here add and sub). That is, if these services would not work properly, the service provided by a component would not work as expected. □

The above example does not use any global state. In a more complex model, a component may also have an encapsulated state, e.g. in class instances in object-oriented programming languages. We can easily encode stateful models by changing the notion of a service to relate *streams* of concrete values of the input parameters to *streams* of concrete return values, as we will see in Ex. IV.2.

In the following, we write $\mathbb{N}_0$ for the set of natural numbers (including zero) and $\mathrm{dom}\, f$ for the domain of a (partial) map f. Given a set X, the set of finite sequences over X will be written as X^*, the set of infinite sequences will be written as X^ω (where the set of indices is the set of natural numbers), and the set of *streams* over X is defined as the set of sequences over X where the indices form a downward-closed subset of natural numbers: $X\,\mathrm{stream} = X^* \cup X^\omega$.

Example IV.2 (Modeling stateful services)**.** Continuing example IV.1, 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. 2. In the worst case the cache grows until the memory is exhausted, after which the behavior is undefined.

We assume that the cache operations are purely internal and that the cache can hold at least $N{\geq}1$ 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 $sbint = bint\,\mathrm{stream}$. We lift the previous (correct) types of i_1 and i_2 pointwise to streams as usual. For example, $\mathrm{type}(i_1) = \{a \in (sbint{\times}sbint \to sbint) \mid \forall r,s,t \in sbint\colon a(r,s) = t \Rightarrow (\mathrm{dom}\, t = (\mathrm{dom}\, r){\cap}(\mathrm{dom}\, s){\wedge}\forall i \in \mathrm{dom}\, t\colon t(i) = \widehat{r(i)+s(i)})\}$, and similar for i_2. (Here, we write $\widehat{a}$ for the representative of $a{\in}\mathbb{Z}$ in $bint$ modulo 2^{M+1}.) We define $\mathrm{type}(o)$ 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 $|\{(r(j),s(j)) \in bint^2 \mid j{\leq}i \wedge j \in (\mathrm{dom}\, r){\cap}(\mathrm{dom}\, s)\}|$ does not exceed N, then $i \in \mathrm{dom}\, t$ and we have $(\widehat{s(i)} \geq 0 \Rightarrow t(i) = \widehat{r(i)s(i)})$.

As we see, types containing functions over streams allow representing stateful abstractions of stateful services without actually referring to their state spaces. □

2) Valuations

For a set of ports $P \subseteq \mathtt{PORT}$, 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,

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

For pairwise different ports p_i ($i{\in}\mathbb{N}_0$, $i{\leq}n$) and any services S_i ($i{\in}\mathbb{N}_0$, $i{\leq}n$) we write

$$[p_0,\ldots,p_n \mapsto S_0,\ldots,S_n] \quad = \quad \{(p_i,S_i) \mid i\in\mathbb{N}_0 \,\wedge\, i{\leq}n\}$$

to denote the valuation that maps each port p_i to the corresponding service S_i ($i{\in}\mathbb{N}_0$, $i{\leq}n$).

3) Components

Informally speaking, a component 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 IV.3. A *component* is a triple (I,O,f) where $I{\subseteq}\mathscr{I}$, $O{\subseteq}\mathscr{O}$, and $f\colon \overline{I}{\to}\wp\left(\overline{O}\right)$. □

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

4) Selection and Projection

To facilitate reasoning about a set of components, we introduce now the so-called selection and projection operators.

A *selection* operator allows to access ports belonging to a set of components.

Definition IV.4. The *input* and *output* ports of a set of components C are respectively defined as

$$\Pi_{\mathrm{i}}(C) \;=\; \bigcup_{c\in C} c.in \quad\text{and}\quad \Pi_{\mathrm{o}}(C) = \bigcup_{c\in C} c.out.$$

To select all ports of a set of components C, we write

$$\Pi\,(C) \quad = \quad \Pi_{\mathrm{i}}(C)\cup\Pi_{\mathrm{o}}(C).$$ □

A *projection* operator, on the other hand, allows to access specific components based on their ports.

Definition IV.5. For a set of components C such that no port is common to more than one component of C and a port $p\in\Pi\,(C)$ we write

$$\sigma_p\,(C)$$

for the unique component $c\in C$ such that $p\in c.in{\cup}c.out$. □

B. Architecture Configuration

We first define the notion of an architecture configuration, then introduce examples, and after that proceed with the syntactic dependency and the semantics.

1) Definition and Examples

An architecture configuration, informally, consists of a set of components and a so-called attachment describing the connections between the components. (From now on we say simply "configuration" for "architecture configuration".)

In the following, we denote by

$$X \dashrightarrow Y = \left\{ f\subseteq X{\times}Y \,\middle|\, \forall\, x,y_1,y_2\colon \begin{pmatrix}(x,y_1),(x,y_2)\in f\\ \Rightarrow\;\; y_1=y_2\end{pmatrix} \right\}$$

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

Definition IV.6. An (architecture) *configuration* is a pair (C,A) where $C\subseteq\mathscr{C}$ and $A\in(\Pi_{\mathrm{i}}(C) \dashrightarrow \Pi_{\mathrm{o}}(C))$, called the *attachment*, are such that the following constraints hold.

- Different components do not share any ports, formally:

$$\forall\; c,\hat{c}\in C\colon c{=}\hat{c} \;\vee\; (c.in\cup c.out)\cap(\hat{c}.in\cup\hat{c}.out)=\emptyset.$$

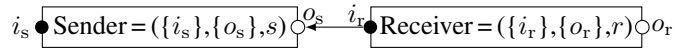
- If a service is provided at an output port that is connected to an input port, the component owning the input port must be able to employ the service, i.e., the port types are compatible. Formally:

$$\forall\; (p_i,p_o)\in A\colon \quad \mathrm{type}(p_o)\subseteq\mathrm{type}(p_i).$$ □

The domain of the attachment is a subset of the occurring input-ports, and the range is a subset of the occurring output-ports, meaning 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. In contrast, an output port may provide services to zero, one, or multiple input ports.

As a preparation for an example, let the set of optional values over a set β be defined as $\beta^{?} = (\mathtt{None} \mid \mathtt{Some}\,\beta)$ (i.e., the disjoint union of a singleton and β, written in ML style).

Example IV.7 (Distributed producer-consumer)**.** In the producer-consumer configuration



Sender is the producer component, Receiver is the consumer component, and the single arrow denotes the attachment $\{(i_{\mathrm{r}},o_{\mathrm{s}})\}$. Let D be the set from which transmitted data is taken.

The types are expressed in terms of streams of optional data. Let $\mathrm{type}(i_{\mathrm{s}}) = \{x \in D^{?}\,\mathrm{stream} \mid \forall\, i,j \in \mathrm{dom}\,x,\, d,d' \in D\colon (i{<}j \wedge x(i) = (\mathtt{Some}\,d) \wedge x(j) = (\mathtt{Some}\,d')) \Rightarrow i{+}2 < j\}$ be the set of all streams over $D^{?}$ such that between any two `Some`s there are at least two `None`s (which represent time steps needed for transmission and acquiring an acknowledgment). Let $\mathrm{type}(o_{\mathrm{s}}) = \mathrm{type}(i_{\mathrm{r}})$ be the set of all streams over $\mathtt{None} \mid \mathtt{Snd}\,D \mid \mathtt{Ack}$ (i.e., the disjoint union of a singleton, D, and another singleton, written in ML style) that, if they are nonempty, start with a nonempty stream of `None`s and, if this prefix is finite, followed by a finite or infinite number of subsequences described by the regular expression $(\mathtt{Snd}\,d)\mathtt{None}^{*}\,\mathtt{Ack}\,\mathtt{None}^{+}$, and, if this prefix is still finite,

followed by an arbitrary stream of Nones. Here, $\texttt{Snd}\,d$ serves as a transmission message and Ack as the acknowledgment of receipt. Let $\mathrm{type}(o_\mathrm{r}) = \{x \in D^? \,\mathrm{stream} \mid (0 \in \mathrm{dom}\, x \Rightarrow x(0) = \texttt{None}) \wedge (1 \in \mathrm{dom}\, x \Rightarrow x(1) = \texttt{None}) \wedge (2 \in \mathrm{dom}\, x \Rightarrow x(2) = \texttt{None}) \wedge \forall\, i,j \in \mathrm{dom}\, x,\, d,d' \in D\colon (i{<}j \wedge x(i) = (\texttt{Some}\,d) \wedge x(j) = (\texttt{Some}\,d')) \Rightarrow i{+}2 < j\}$ be the set of all streams over $D^?$ that start with three Nones and such that any two consecutive Somes are separated by at least two Nones.
The behavior map of the sender s takes a stream $x \in \mathrm{type}(i_\mathrm{s})$, and returns the set of all such streams from $\mathrm{type}(o_\mathrm{s})$ that result from x by replacing each longest contiguous subsequence $(\texttt{Some}\,d)\underbrace{\texttt{None}\ldots\texttt{None}}_{n\geq 2 \text{ times}}$ by subsequences of the form $\texttt{None}(\texttt{Snd}\,d)\underbrace{\texttt{None}\ldots\texttt{None}}_{i \text{ times}}\texttt{Ack}\underbrace{\texttt{None}\ldots\texttt{None}}_{n-i-2 \text{ times}}$ for all $i \in \mathbb{N}_0 \cap [0, n{-}2]$, $n \in \mathbb{N}^+ \cup \{\infty\}$. This way we model that the sender waits for an acknowledgment receipt.
The behavior function of the receiver r takes a stream $y \in \mathrm{type}(i_\mathrm{r})$. If y contains a subsequence of the form $(\texttt{Snd}\,d)\texttt{None}$, the empty set is returned. Otherwise the singleton is returned which contains the sequence that results from substituting each subsequence of the form $(\texttt{Snd}\,d)\texttt{Ack}\,\texttt{None}$ by $\texttt{None}\,\texttt{None}(\mathrm{Some}\,d)$ in y. This way we model that the receiver sends an acknowledgment immediately after receiving data. □

Example IV.8 (Web services)**.** Consider an online holiday booking site. In the distributed world of web services, the booking site expects that at certain web addresses, providers of real services like car hire companies, hotels, and airlines provide booking interfaces. Simplifying, the input ports are the names of those interfaces, say, web addresses with a network interface suitable for machine-to-machine communication. The input services are then maps from customer data to tokens confirming bookings. The output ports are HTTP addresses and the output services are maps from certain extended customer data to human-readable booking confirmations. We may even have a chain of servers of different specializations between the real hotel interfaces and the holiday booking site. □

2) Syntactic Dependency

In a configuration, the attachment relation induces a dependency relation between the components. We say that a component c *syntactically* depends on another components c', if an input port of c is connected to an output port of c'.

Definition IV.9. Syntactic dependency for a configuration $z = (C, A)$ is a relation $\prec_z \;\subseteq\; C \times C$ defined by

$$c' \prec_z c \quad \overset{\text{def}}{\iff} \quad \exists\, i \in c.in,\, o \in c'.out\colon \quad (i,o) \in A\,. \qquad \square$$

For a configuration $z = (C, A)$, we denote by $\prec_z^+$ the *transitive* closure of $\prec_z$ and by $\prec_z^*$ the *reflexive-transitive* closure of $\prec_z$. Moreover, we denote by $\prec_z \_ \colon C \to \wp(C)$, defined via

$$\prec_z c = \{c' \in C \mid c' \prec_z c\} \qquad \text{for } c \in C\,,$$

all components c' that a given component c syntactically depends on ($\prec_z^* \_$, $\prec_z^+ \_$ for [reflexive-] transitive dependency, respectively).

Note that the syntactic dependency relation is not transitive in general: just because a component c_1 depends on another component c_2 which depends on a third component c_3, this does not necessarily mean that c_1 depends on c_3.

3) Upper-Level and Lower-Level Components

For each component in a configuration, we can identify its upper- and lower-level components. The upper-level components are all the components that transitively syntactically depend on the original component, excluding the original component itself. The lower-level components, on the other hand, are all the components that the original component reflexively-transitively syntactically depends on. In the sequel, we drop "-level" for brevity.

Definition IV.10. Let $z = (C, A)$ be a configuration. The *upper components* of a component $c \in C$ are

$$c{\uparrow_z} \;=\; \left\{c' \in C \setminus \{c\} \mid c \prec_z^+ c'\right\},$$

and its *lower components* are

$$c{\downarrow_z} \;=\; \{c' \in C \mid c' \prec_z^* c\}\,.$$

The *lower components* of a set of components $S \subseteq C$ are

$$S{\downarrow_z} \;=\; \bigcup_{c \in S} c{\downarrow_z}\,. \qquad \square$$

4) Semantics

Now we are going to define the computational meaning of a configuration. As a preparation, we define the *open input-ports* of a configuration (C, A) as

$$\Pi_\mathrm{in}((C, A)) \;\overset{\text{def}}{=}\; \Pi_\mathrm{i}(C) \setminus (\mathrm{dom}\, A).$$

In the following, for a (partial) map f and a set Z, we write $f|_Z$ for the restriction of f to the domain $(\mathrm{dom}\, f) \cap Z$.

Definition IV.11. Given a configuration $z = (C, A)$, the semantics of a set of components $S \subseteq C$ is a map $[\![z, S]\!] : \overline{\Pi_\mathrm{in}(z)} \to \wp\left(\overline{\Pi_\mathrm{o}(S{\downarrow_z})}\right)$,

$$\begin{aligned}
\mu \mapsto \big\{\nu|_{\Pi_\mathrm{o}(S)} \text{ such that} \qquad & (1)\\
\nu \in \overline{\Pi(S{\downarrow_z})} \qquad & (2)\\
\wedge\ \nu|_{\Pi_\mathrm{in}(z)} = \mu|_{\Pi(S{\downarrow_z})} \qquad & (3)\\
\wedge\ (\forall\, i \in \Pi_\mathrm{i}(S{\downarrow_z}) \setminus (\Pi_\mathrm{in}(z))\colon \nu(i) = \nu(A(i))) \qquad & (4)\\
\wedge\ (\forall\, c' \in S{\downarrow_z}\colon\ \nu|_{c'.out} \in c'.\mathit{fun}(\nu|_{c'.in}))\big\} \qquad & (5)
\end{aligned}$$

□

Intuitively, given a valuation μ of the open input ports, an element of the component semantics $[\![z, S]\!](\mu)$ is created by restricting (line 1) a valuation ν of all the ports of the lower components (line 2) that is consistent with μ (line 3), with the attachment, restricted to the lower components (line 4), and with the behaviors of the lower components (line 5).

We shorten the semantics of the whole configuration $[\![(C, A), C]\!]$ to $[\![(C, A)]\!]$ and the semantics of one component

$[\![z, \{c\}]\!]$ to $[\![z, c]\!]$. In the sequel we concentrate on the semantics of single components.

A component of an architecture configuration is usable iff its semantics can be nonempty. Formally:

Definition IV.12. Let $z = (C, A)$ be a configuration and $c \in C$. Then c is called *usable* iff $\exists\, \mu \in \overline{\Pi_{\text{in}}(z)} \colon [\![z, c]\!](\mu) \neq \emptyset$. □

C. Semantic Dependencies

In this section we are going to investigate the notion of semantic dependency. To define it, we first have to introduce the concept of an architecture update.

1) Architecture Update

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

Definition IV.13. For a component $c = (I, O, f)$ and a map $f' \colon \overline{I} \to \wp\left(\overline{O}\right)$, a *semantic update* $[c \triangleleft f']$ is the component (I, O, f'). □

Note that a semantic update is indeed a component according to Def. IV.3.

The notion of semantic update easily generalizes to configurations:

$$(C, A)\,[c \triangleleft f'] \quad = \quad ((C \backslash \{c\}) \cup \{[c \triangleleft f']\},\ A)\,.$$

Proposition IV.14. *For a configuration $z = (C, A)$, component $(I, O, f) \in C$, and a map $f' \colon \overline{I} \to \wp\left(\overline{O}\right)$, $z\,[c \triangleleft f']$ is a configuration.*

2) Weak Semantic Dependency

Besides the syntactic dependency relation between components of a configuration we also have semantic dependency relations between those components. Informally, a component c weakly semantically depends on a component c' if updating c' may influence the semantics of c.

Definition IV.15. *Weak semantic dependency* for a configuration $z = (C, A)$ is a relation $\ll_z \subseteq\ C \times C$ defined by

$$c \ll_z c \overset{\text{def}}{\iff} \exists f \in \left(\overline{c.in} \to \wp(\overline{c.out})\right) : [\![z, c]\!] \neq [\![z[c \triangleleft f], [c \triangleleft f]]\!]$$

for all $c \in C$ and

$$c' \ll_z c \overset{\text{def}}{\iff} \exists f \in (\overline{c'.in} \to \wp(\overline{c'.out})) \colon [\![z, c]\!] \neq [\![z[c' \triangleleft f], c]\!]$$

for all $c \neq c'$ in C. □

As expected, weak semantic dependency implies the reflexive-transitive syntactic dependency:

Theorem IV.16. *In any configuration z we have $\ll_z \subseteq \prec_z^*$.*

In particular, if c does not reflexively-transitively syntactically depend on c', any changes in the behavior of c' have no influence on the semantics of c, so in the development process there is no need to re-test or re-verify c. This results in time and cost savings during development.

Weak semantic dependency is a coarse notion: the reverse of Thm. IV.16 holds under particular circumstances. Whenever c is usable and syntactically reflexively-transitively depends on some c', c' may just stop providing services, thus resulting in c also stopping to provide services; hence c weakly semantically depends on c'. It does not matter hereby how the services provided by c' are actually used; what matters, is just their presence:

Theorem IV.17. *Usable components depend reflexively-transitively syntactically iff they depend weakly semantically. Formally:*
Let $c \in C$ be a usable component of a configuration $z = (C, A)$ and $c' \in C$. Then

$$c' \ll_z c \quad \Leftrightarrow \quad c' \prec_z^* c\,.$$

3) Strong Semantic Dependency

Due to the equivalence in Thm. IV.17, we need a stronger, more fine-grained notion of dependency. We start by an auxiliary definition.

For a configuration $z = (C, A)$, a component $c \in C$, and a map $f \colon \overline{I} \to \wp\left(\overline{O}\right)$, we say that the *updating of c in z by f is live* iff

$$\forall\, \mu \in \overline{\Pi_{\text{in}}(z)} \colon [\![z, c]\!](\mu) \neq \emptyset \Rightarrow [\![z, [c \triangleleft f]]\!](\mu) \neq \emptyset\,,$$

i.e., informally, $[c \triangleleft f]$ continues to produce some output whenever c did so.

Informally, a component c strongly semantically depends on a component c' if updating c' may influence the semantics of c but not destroy it completely.

Definition IV.18. Strong semantic dependency for a configuration $z = (C, A)$ is a relation $\lll_z \subseteq\ C \times C$ defined by

$$c \lll_z c \quad \overset{\text{def}}{\iff} \left(\begin{array}{c} \exists\ f \in \left(\overline{c.in} \to \wp(\overline{c.out})\right) : \\ \text{the updating of } c \text{ in } z \text{ by } f \text{ is live and} \\ [\![z, c]\!] \neq [\![z[c \triangleleft f], [c \triangleleft f]]\!] \end{array}\right)$$

for $c \in C$ and

$$c' \lll_z c \quad \overset{\text{def}}{\iff} \left(\begin{array}{c} \exists f \in (\overline{c'.in} \to \wp(\overline{c'.out})) : \\ \text{the updating of } c' \text{ in } z \text{ by } f \text{ is live and} \\ [\![z, c]\!] \neq [\![z[c' \triangleleft f], c]\!] \end{array}\right)$$

for $c \neq c'$ in C. □

By defintion, strong semantics dependency implies the weak one, and, by Thm IV.17, also implies the reflexive-transitive syntactic dependency. In general:

- Neither weak nor strong semantic dependency imply direct syntactic dependency.
- Reflexive-transitive syntactic dependency does not imply strong semantic dependency.

D. Semantics Analysis

In this section we consider a fast way of analyzing the semantics of components in a configuration.

Constructing the *exact* semantics of a component involves constructing a set of valuations of ports of all lower components. In the finite case, the number of such valuations is in general exponential in the number of components, implying exponential worst-case space consumption and running time. We show how to obtain an *overapproximation* of the semantics that has an asymptotically linear space consumption and, under modest additional assumptions, polynomial running time.

For this section we fix a configuration $z = (C, A)$ and one of its components $c{\in}C$ whose semantics we are going to analyze. Given a valuation $\mu \in \overline{\Pi_{\mathrm{in}}(z)}$ of the open input-ports, let the *concrete domain*

$$\begin{aligned} D_{z,c}(\mu) \;=\; \wp(\{v \in \overline{\Pi(c{\downarrow_z})} \mid v|_{\Pi_{\mathrm{in}}(z)} = \mu|_{\Pi(c{\downarrow_z})} \\ \wedge\, \forall\, i \in \Pi_{\mathrm{i}}(c{\downarrow_z}) \setminus (\Pi_{\mathrm{in}}(z)) \colon \\ v(i) = v(A(i))\}) \end{aligned}$$

be the power set of valuations of all the relevant ports such that the services at each pair of attached ports coincide. Let us equip $D_{z,c}(\mu)$ with the subset partial order $\subseteq$. Let the *abstract domain*

$$D^{\#}_{z,c} = \prod_{r \prec^*_z c} \wp\left(\overline{r.out}\right)$$

be the set of tuples of sets of valuations of relevant output ports of individual components, equipped with the componentwise subset partial order. Both domains are complete lattices. We use $D_{z,c}(\mu)$ to provide the exact semantics of c in z (as in Def. IV.11) and $D^{\#}_{z,c}$ to define the approximate semantics.

A *component-Cartesian step* of the configuration is given by the map

$$\begin{aligned} &\mathrm{post}^{\#}_{z,c}(\mu) : D^{\#} \to D^{\#}, \\ &S \;\mapsto\; \Big(\big\{w|_O \mid \exists\, I,f \colon s = (I,O,f) \;\wedge\; w|_O \in f(w|_I) \\ &\qquad \wedge\, w \in \overline{I \cup O \cup \{o \mid \exists\, i{\in}I \colon (i,o){\in}A\}} \\ &\qquad \wedge\, w|_{\Pi_{\mathrm{in}}(z)} = \mu|_I \\ &\qquad \wedge\, (\forall\, (i,o) \in A \colon i \in I \Rightarrow w(i) = w(o)) \\ &\qquad \wedge\, (\forall\, r \in \prec_z\, s\; \exists\, v{\in}S_r \colon v|_{\{o \mid \exists\, i{\in}I \colon (i,o){\in}A\}} = \\ &\qquad\qquad w|_{\{o \in r.out \mid \exists\, i{\in}I \colon (i,o){\in}A\}})\big\}\Big)_{s \in c{\downarrow_z}}. \end{aligned}$$

Intuitively, the component-Cartesian step takes an element of the abstract domain and constructs all the possible resulting services, forgetting all dependencies between the components except those imposed by applying the behavior maps to parts of μ. This map is isotone, hence it has the greatest fixpoint by Tarski's fixpoint theorem [26].

Let the *component-Cartesian semantics* of component c be the valuation of the output ports of c of the greatest fixpoint of the component-Cartesian step: $[\![z, c]\!](\mu) := (\mathrm{gfp}\;\; \mathrm{post}^{\#}_{z,c}(\mu))_c$.

Theorem IV.19. *For all valuations μ of open input-ports, the component-Cartesian semantics of a component c is an overapproximation of the semantics of c, formally: $[\![z, c]\!](\mu) \subseteq [\![z, c]\!](\mu)$.*

The greatest fixpoint (and, hence, the component-Cartesian semantics) is typically constructed by computing the limit of a lower iteration sequence [27].

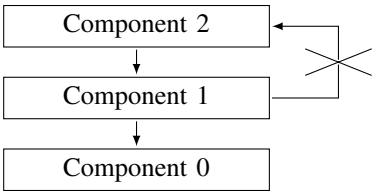


Figure 3: Layered architecture style.

In the finite case (to be a bit more precise, when $\left|\bigcup_{r \prec^*_z c}(c.in \cup c.out) \cup \bigcup \mathrm{type}\big(\bigcup_{r \prec^*_z c}(c.in \cup c.out)\big)\right| < \infty$) and certain restricted infinite cases the exact semantics and the component-Cartesian semantics can be computed precisely. Let us assume the finite case and a constant upper bound on the number of ports of each component. Then $\frac{|D^{\#}_{z,c}|}{|D_{z,c}(\mu)|}$ approaches zero with the growing number of components (e.g., when copies of the components with their ports are created and μ is extended appropriately). Since $|D^{\#}_{z,c}|$ is linear in the number of components, and the constructed valuations have a bounded-size domain, the time to construct the component-Cartesian semantics has a polynomial (hence better compared to the exact semantics) asymptotic upper bound in the number of components.

Background on variants of (Cartesian) abstract semantics can be found in, e.g., [28], [29], [30], [31], [32].

V. Variants

Now we are going to impose certain constraints on our model and investigate the obtained variants. We are going to consider the layered architecture variant and the strict variant.

A. Layered Variant

Fig. 3 depicts the layered architecture style as usually seen in informal descriptions of the style. The picture suggests that in a layered architecture style, upper components use services from the lower components, but not vice versa: lower components are not allowed to use services from the upper components. Thus, the topology of the configuration forms, up to self-loops, a directed acyclic graph. Informally, in the layered variant, components from the same level are gathered into layers. In a nonempty set of components one should be able to distinguish the lowest layers, leading us to the following definition:

Definition V.1. A configuration is *layered* iff its syntactic dependency relation is well-founded, i.e., iff every nonempty set of components contains a component that syntactically depends at most on itself:

$$\begin{aligned} &(C, A) \text{ is } \textit{layered} \quad \overset{\text{def}}{\iff} \\ &(\forall\, S \subseteq C \colon S{\neq}\emptyset \;\Rightarrow\; (\exists\, s{\in}S\; \forall\, t{\in}S \colon\; t \prec_z s \;\Rightarrow\; t{=}s)) \,. \end{aligned}$$

□

In layered configurations the syntactic dependency is always acyclic up to self-loops. If the number of components is finite (which is usually the case in real-world architectures), also the reverse holds. Formally, for all configurations $z = (C, A)$ we

have:

$$(z \text{ layered } \Rightarrow \forall\, c{\in}C\colon (c,c) \notin (\prec_z \setminus \mathrm{id}_C)^+)$$
$$\wedge\ (C \text{ finite } \Rightarrow (z \text{ layered } \Leftarrow \forall\, c{\in}C\colon (c,c) \notin (\prec_z \setminus \mathrm{id}_C)^+)).$$

In the finite case, the computation of the component-Cartesian semantics can proceed bottom-up.

In Example IV.7 we find a very simple layered configuration.

1) Implications

In the following we are going to investigate properties of layered configurations.

One direct consequence is that the upper and the lower components of any component are distinct:

Lemma V.2. *For a* layered *configuration $z = (C, A)$, we have*

$$\forall\, c \in C\colon\ c{\uparrow_z} \cap c{\downarrow_z} = \emptyset.$$

We shall use this property later on to prove some other, characteristic properties for layered configurations.

a) Semantic Independence from Upper Components

A useful property of layered configurations is that for each component, changing the semantics of an upper component does indeed not change the semantics of the original component.

Theorem V.3. *For a layered configuration $z = (C, A)$ and a component $c \in C$, we have:*

$$\forall\, m \in c{\uparrow_z},\ f \in (\overline{m.in} \to \wp(\overline{m.out}))\colon\ [\![z, c]\!] = [\![z[m \lhd f], c]\!].$$

A direct consequence of the above property is that a component of a layered configuration is semantically independent of all its upper components.

Corollary V.4. *For a configuration $z = (C, A)$ and a component $c \in C$, we have:*

$$\forall\, m \in c{\uparrow_z}\colon\ m \not\ll_z c.$$

The above properties ensure that for a component c of a layered configuration, every modification of its upper components does not impact the semantics of c. Thus, re-testing or re-verifying c after such modifications is not necessary, not to say useless. This is a good example of how architectural design decisions influence quality attributes of the resulting software system: in that we restrict a software systems architecture to a layered architecture style, we increase the maintainability of the resulting system.

Note V.5. The above properties only hold for *layered* configurations; the proofs rely on acyclic dependencies. For configurations with cyclic dependencies, the above properties do not hold in general. □

b) Semantic Independence of Lower Components

Another important property of layered configurations regards the influence of changing (adapting) a component's behavior to its upper components. It turns out that under certain circumstances, changing a component's behavior does indeed not influence its upper components.

Notice that `SERVICE` might contain not the actual low-level services, but their abstractions. Hence, changing the actual low-level implementation of a component does not necessarily imply that its formal behavior changes or that the semantics of the upper components change:

$$\forall\, g \in (\overline{c.in} \to \wp(\overline{c.out}))\colon g = c.fun \Rightarrow \forall\, c' \in c{\uparrow_z}\colon [\![z, c']\!] = [\![z[c \lhd g], c']\!].$$

The constraint $g = c.fun$ is actually the strongest one to ensure that the configuration semantics of upper components is not influenced by the change. In the following we will give another, weaker constraint which yields the same result.

Theorem V.6. *Let $z = (C, A)$ be a layered configuration, $c{\in}C$, and $g : \overline{c.in} \to \wp\left(\overline{c.out}\right)$ some map. Let $\forall\, \mu \in \overline{\Pi_{\mathrm{in}}(z)}$, $\eta \in \overline{c.in}\colon c.fun(\eta) = g(\eta)\ \Leftarrow$*

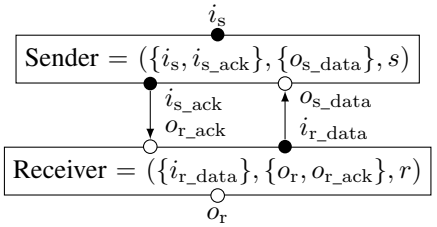
$$\left(\begin{array}{l}\eta|_{\Pi_{\mathrm{in}}(z)} = \mu|_{c.in}\ \wedge \\ \forall m{\in}C\setminus\{c\}\colon \left(\begin{array}{l} m \prec_z c\ \Rightarrow\ \exists\, \xi{\in}[\![z, m]\!](\mu)\colon \\ \forall\, (i,o){\in}A{\cap}(c.in{\times}m.out)\colon \eta(i){=}\xi(o)\end{array}\right)\end{array}\right). \quad (6)$$

Then $\forall\, c' \in C\colon [\![z, c']\!] = [\![z[c \lhd g], \text{if } c'{=}c \text{ then } [c \lhd g] \text{ else } c']\!]$.

The assumption of Thm. V.6 is a weaker constraint that ensures that a change of a component's behavior does not influence the other components of the architecture. It requires the new component to behave the same as the old one under some circumstances, which are described by (6) and characterize all valuations which are possible for the changed component in its environment.

Our proof of Thm. V.6 uses neither well-foundedness nor acyclicity for the "$\subseteq$" part; thus, the left inclusion holds also for nonlayered configurations. However, as shown by Ex. V.7 below, the right inclusion "$\supseteq$" requires acyclicity. This example will also demonstrate a possibility of a more fine-grained modeling of a system which was coarsely modeled earlier.

Example V.7 (Distributed producer-consumer)**.** Let us model the producer and the consumer by a configuration $z = (C, A)$, in which, as opposed to Example IV.7, the channels for data transmission and receipt acknowledgment are represented separately:



Sender is the producer component, Receiver is the consumer component, and the arrows denote the attachment $\{(i_{\mathrm{r\_data}}, o_{\mathrm{s\_data}}), (i_{\mathrm{s\_ack}}, o_{\mathrm{r\_ack}})\}$. Reusing notation from Example IV.7, let D be a nonempty set from which transmitted data is taken.

The types are similar to before. Formally, let $\mathrm{type}(i_{\mathrm{s}}) = \{x \in D^? \text{ stream} \mid \forall\, i{,}j \in \mathrm{dom}\, x,\ d{,}d' \in D\colon (i{<}j \wedge x(i) = (\mathtt{Some}\, d) \wedge x(j) = (\mathtt{Some}\, d')) \Rightarrow i{+}2 < j\}$, and $\mathrm{type}(o_{\mathrm{r}})$

$= \{x \in D^? \,\mathrm{stream} \mid (0 \in \mathrm{dom}\, x \Rightarrow x(0) = \mathtt{None}) \wedge (1 \in \mathrm{dom}\, x \Rightarrow x(1) = \mathtt{None}) \wedge (2 \in \mathrm{dom}\, x \Rightarrow x(2) = \mathtt{None}) \wedge \forall i,j \in \mathrm{dom}\, x,\, d,d' \in D\colon (i<j \wedge x(i) = (\mathtt{Some}\, d) \wedge x(j) = (\mathtt{Some}\, d')) \Rightarrow i+2 < j\}$, and $\mathrm{type}(o_{\mathrm{s\_data}}) = \mathrm{type}(i_{\mathrm{r\_data}}) = \{y \in (\mathtt{None}|\mathtt{Snd}\, D)\mathrm{stream} \mid (0 \in \mathrm{dom}\, y \Rightarrow y(0) = \mathtt{None}) \wedge \forall i,j \in \mathrm{dom}\, x,\, d,d' \in D\colon (i<j \wedge x(i) = (\mathtt{Snd}\, d) \wedge x(j) = (\mathtt{Snd}\, d')) \Rightarrow i+2 < j\}$, and $\mathrm{type}(i_{\mathrm{s\_ack}}) = \mathrm{type}(o_{\mathrm{r\_ack}}) = \{x \in (\mathtt{None}|\mathtt{Ack})\mathrm{stream} \mid (0 \in \mathrm{dom}\, x \Rightarrow x(0) = \mathtt{None}) \wedge (1 \in \mathrm{dom}\, x \Rightarrow x(1) = \mathtt{None}) \wedge \forall i,j \in \mathrm{dom}\, x\colon (i<j \wedge x(i) = x(j) = \mathtt{Ack}) \Rightarrow i+2 < j\}$.

The sender, roughly speaking, waits for exactly one acknowledgment from a previous transmission on port $i_{\mathrm{s\_ack}}$ if there was any, and only then accepts new data from i_{s} for sending through $o_{\mathrm{s\_data}}$ in the next step. Formally, the behavior s takes a valuation $[i_{\mathrm{s}}, i_{\mathrm{s\_ack}} \mapsto x, y]$ and returns a singleton containing a valuation of $o_{\mathrm{s\_data}}$ by a stream of the maximal length n, which may be infinity, such that for all $m<n$, (the number of $\mathtt{Some}$s till and including position m in x) $-$ (the number of $\mathtt{Ack}$s till and including position m in y) $\in \{0,1\}$, and for all $d \in D$, whenever $x_m = \mathtt{Some}\, d$, then $\exists\, k \in \mathrm{dom}\, y\colon k \geq m+2 \wedge k \in \mathrm{dom}\, y \wedge y_m = y_{m+1} = \mathtt{None} \wedge y_k = \mathtt{Ack} \wedge \forall\, j \in \mathbb{N}^+ \cap [m+1, k]\colon (j \in \mathrm{dom}\, x \Rightarrow x_j = \mathtt{None})$. The returned stream for $o_{\mathrm{s\_data}}$ results from replacing substrings of the form $(\mathtt{Some}\, d)\mathtt{None}$ in $(x_i)_{i<n}$ by $\mathtt{None}(\mathtt{Snd}\, d)$ (where $d \in D$), and, if $x = (x_i)_{i<n}$ and the final symbol in x exists and is $\mathtt{Some}\, d$, replacing it by $\mathtt{None}(\mathtt{Snd}\, d)$ (where $d \in D$).

For the sake of the example, let us first assume a useless receiver whose behavior r always produces the empty set. Then, by Def. IV.11, $[\![z, \mathrm{Sender}]\!](\mu) = \emptyset$ for all $\mu \in \overline{\{i_{\mathrm{s}}\}}$.

Second, consider a usable behavior r' (intended to replace the useless one for the receiver), which, roughly speaking, accepts the data from $i_{\mathrm{r\_data}}$, sends the acknowledgment receipt in the next step to $o_{\mathrm{r\_ack}}$, and then outputs the data to o_{r}. Formally, consider the map $r' : \overline{\{i_{\mathrm{r\_data}}\}} \to \wp\left(\overline{\{o_{\mathrm{r\_ack}}, o_{\mathrm{r}}\}}\right)$ that takes a valuation $[i_{\mathrm{r\_data}} \mapsto w]$ for some w and returns a singleton containing a valuation of $\{o_{\mathrm{r\_ack}}, o_{\mathrm{r}}\}$ by two streams of lengths n and $n+1$ (if $n<\infty$) or both ∞ (if $n=\infty$), such that n is the maximal element of $\mathbb{N}_0 \cup \{\infty\}$ such that for all $m<n$, whenever for some $d \in D$ we have $w_m = (\mathtt{Snd}\, d)$, then $(m+1 \in \mathrm{dom}\, w \Rightarrow w_{m+1} = \mathtt{None}) \wedge (m+2 \in \mathrm{dom}\, w \Rightarrow w_{m+2} = \mathtt{None})$. The returned stream for $o_{\mathrm{r\_ack}}$ is built from $(w_i)_{i<n}$ by replacing substrings of the form $(\mathtt{Snd}\, d)\mathtt{None}$ by $\mathtt{NoneAck}$ (where $d \in D$), and, if $w = (w_i)_{i<n}$ and the final symbol of w is $\mathtt{Snd}\, d$, replacing it by $\mathtt{NoneAck}$ (where $d \in D$). The returned stream for o_{r} is built from $(w_i)_{i<n}$ by replacing substrings of the form $(\mathtt{Snd}\, d)\mathtt{NoneNone}$ by $\mathtt{NoneNone}(\mathtt{Some}\, d)$ (where $d \in D$), and, if $w = (w_i)_{i<n}$ and w ends in $\mathtt{Snd}\, d$ or $(\mathtt{Snd}\, d)\mathtt{None}$, replacing this suffix by $\mathtt{NoneNone}(\mathtt{Some}\, d)$ (where $d \in D$).

Notice that the only $m \in C$ such that $m \prec_z \mathrm{Receiver}$ is $m = \mathrm{Sender}$, and $[\![z, m]\!](\mu)$ is empty for every $\mu \in \overline{\{i_{\mathrm{s}}\}}$ due to the uselessness of the receiver in z. Hence the condition "$\exists\, \xi \in [\![z, m]\!](\mu) \ldots$" from (6) is not satisfied, hence the condition
$\forall\, \mu \in \overline{\Pi_{\mathrm{in}}(z)},\, \eta \in \overline{\mathrm{Receiver}.in}\colon \mathrm{Receiver}.fun(\eta) = r(\eta) \;\Leftarrow$

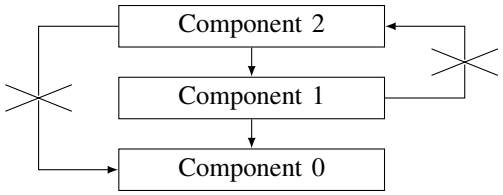


Figure 4: Strict layered architecture style

$\left(\begin{array}{l}\eta|_{\Pi_{\mathrm{in}}(z)} = \mu|_{\mathrm{Receiver}.in} \wedge \\ \forall m \in C \setminus \{\mathrm{Receiver}\}\colon (m \prec_z \mathrm{Receiver} \Rightarrow \exists \xi \in [\![z, m]\!](\mu) \ldots)\end{array}\right)$
of Thm. V.6 is fulfilled. Nevertheless, e.g., for each single-data input-port valuation $\mu = [i_{\mathrm{s}} \mapsto \mathtt{Some}\, d]$ we have $[\![z[\mathrm{Receiver} \lhd r'], \mathrm{Sender}]\!](\mu) = \{[o_{\mathrm{s\_data}} \mapsto \mathtt{NoneNoneNone}(\mathtt{Snd}\, d)]\} \supsetneq \emptyset = [\![z, \mathrm{Sender}]\!](\mu)$ (where $d \in D$). □

Thm. V.6 can be used by an architect to modify a component without the need to test the other components, since their semantics is guaranteed to stay the same after the modification.

B. Strict Variant

Figure 4 depicts the strict variant of the layered architecture style as usually found in informal descriptions of the style.

A configuration is called strict if every pair of components is transitively syntactically connected in at most one way.

Formally, a configuration $z = (C, A)$ is *strict* iff for all $c, c' \in C$ and all walks τ, θ in the graph $(C, \prec_z \backslash \mathrm{id}_C)$ from c to c' we have $\tau = \theta$.

Strict configurations with finitely many components are layered. In a strict configuration with finitely many components it is possible to compute the component-Cartesian semantics of a component in linear time in the number of components. Namely, given a valuation μ of the open input-ports, start with the bottom components and compute the entries of $\mathrm{gfp}\ \mathrm{post}^{\#}_{z,c}(\mu)$ bottom-up. In layered configurations we may lose precision this way, but in strict ones the component-Cartesian analysis is exact:

Theorem V.8. *In a strict configuration z, the component-Cartesian semantics of a component c is equal to the semantics of c, formally: $[\![z, c]\!] = [\![\![z, c]\!]\!]$.*

VI. Conclusion

This work provides an abstract model for the service-oriented architecture style and two variants thereof: the *layered* variant and the *strict* variant.

Our model is based on the notion of services and ports which can supply services. A component contains input and output ports; its behavior is modeled as a function from input-port valuations to sets of output-port valuations. A (service-oriented architecture) configuration is a pair of a set of components and an attachment describing the connections between the components' input and output ports. The layered variant requires the syntactic dependency relation to be well-founded, while the strict variant adds an antitransitivity requirement. Components can depend on each other syntactically or semantically, while semantic dependence is separated into a weak and a strong one.

We showed how to analyze components and prove properties of the semantics of a component. Our approach applied well-known Cartesian abstract interpretation to the service-oriented architectures. We also formally identified upper and lower levels of components relative to a fixed component. For layered architectures, we demonstrated that a component is indeed semantically independent of its upper components. Finally, we revealed conditions that ensure that changing a component's behavior does indeed not influence the configuration semantics of other components.

Our results offer a technology-independent characterization of the service-oriented architecture style, which may be used by software architects to ensure that a system is indeed built according to that style. Moreover, our results give the software architect a set of guarantees which are assured to hold for a software system built according to the style.

Concluding, we want to discuss one potential weak point of all formal approaches in software engineering, namely scalability. With our approach we address this issue by concentrating on the specification and analysis of system families rather than of single systems. Thus, the specification of a style (such as the model for layered architectures provided in this article) can be used for all systems built according to that style. Moreover, the results of style analysis (such as the theorems regarding semantic independence of layers provided in this article) apply to all systems built according to that style.

Having developed a formal model of service-oriented architectures and two common variants thereof, the model can now be used for a rigorous analysis of the style. Thus, future work arises in three main areas:

1) First, new variants of the style should be identified and defined through constraints over our model.
2) Then, for each variant (existing and new ones), a set of properties should be formulated and proved from the constraints.
3) Finally, the approach should be applied to investigate other architectural styles as well (e.g., Blackboard) to establish a rigorous theory of architectural styles.

Acknowledgment

We would like to thank Jonas Eckhardt and Xiuna Zhu for comments and helpful suggestions. We acknowledge the financial support of the ARAMiS project of the German Federal Ministry for Education and Research, ID 01IS11035, as well as of the chair of foundations of software and systems engineering, led by Manfred Broy.

References

[1] M. Broy, "Can practitioners neglect theory and theoreticians neglect practice?" *IEEE Computer*, vol. 44, no. 10, pp. 19–24, 2011.

[2] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the theory for software engineering?" *IEEE software*, vol. 29, no. 5, p. 96, 2012.

[3] D. Garlan, "Software architecture: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA: ACM, 2000, pp. 91–101.

[4] M. Shaw and P. Clements, "The golden age of software architecture," *Software, IEEE*, vol. 23, no. 2, pp. 31–39, 2006.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture In Practice*, 3rd ed. Pearson Education, Inc., 2012.

[6] D. Marmsoler, "Towards a theory of architectural styles," in *22th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-22)*, 2014, to appear, accepted for publication.

[7] G. D. Abowd, R. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 4, pp. 319–364, 1995.

[8] D. Marmsoler, A. Malkis, and J. Eckhardt, "A model of layered architectures," in *FESCA'2015*, 2014.

[9] R. Allen, "A formal approach to software architecture," Ph.D. dissertation, Carnegie Mellon, School of Computer Science, January 1997, issued as CMU Technical Report CMU-CS-97-144.

[10] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall Englewood Cliffs, 1985, vol. 178. [Online]. Available: http://www.usingcsp.com/cspbook.pdf

[11] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356–372, 1995.

[12] D. Le Métayer, "Describing software architecture styles using graph grammars," *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 521–533, 1998.

[13] J. L. Fiadeiro, *Categories for software engineering*. Springer, 2005.

[14] M. Bernardo, P. Ciancarini, and L. Donatiello, "On the formalization of architectural types with process algebras," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 140–148, 2000.

[15] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[16] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2010.

[17] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. John Wiley & Sons, 2007, vol. 5.

[18] D. Garlan and D. Notkin, "Formalizing design spaces: Implicit invocation mechanisms," in *VDM'91*, vol. 1, 1991, pp. 31–44.

[19] D. Garlan and N. Delisle, "Formal specifications as reusable frameworks," in *VDM'90*. Springer, 1990, pp. 150–163.

[20] R. Allen and D. Garlan, "A formal approach to software architectures," *Proceedings of the IFIP 12th World Computer Congress*, vol. 1, pp. 134–141, 1992.

[21] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[22] J. P. Sousa and D. Garlan, "Formal modeling of the enterprise JavaBeans component integration framework," *Information and Software Technology*, vol. 43, no. 3, pp. 171–188, 2001.

[23] P. Zave and J. Rexford, "Compositional network mobility," in *Verified Software: Theories, Tools, Experiments - 5th International Conference*, 2013, pp. 68–87.

[24] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT Press, 2012.

[25] M. Broy, "Service-oriented systems engineering: Specification and design of services and layered architectures," in *Engineering Theories of Software Intensive Systems*, ser. NATO Science Series, M. Broy, J. Grünbauer, D. Harel, and C. A. R. Hoare, Eds. Springer Netherlands, 2005, vol. 195, pp. 47–81. [Online]. Available: http://dx.doi.org/10.1007/1-4020-3532-2_2

[26] A. Tarski, "A lattice theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[27] P. Cousot and R. Cousot, "Constructive versions of Tarski's fixed point theorems," *Pacific Journal of Mathematics*, vol. 82, no. 1, pp. 43–57, 1979.

[28] B. Blanchet, "Introduction to abstract interpretation," 2002, lecture script, http://prosecco.gforge.inria.fr/personal/bblanche/absint.pdf.

[29] P. Cousot and R. Cousot, "Formal language, grammar and set-constraint-based program analysis by abstract interpretation," in *FPCA*. ACM Press, New York, NY, 1995, pp. 170–181.

[30] N. D. Jones and S. S. Muchnik, "Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra," in *Program Flow Analysis: theory and applications*, 1981.

[31] A. Malkis, A. Podelski, and A. Rybalchenko, "Thread-modular verification is Cartesian abstract interpretation," in *ICTAC'06*, ser. Lecture Notes in Computer Science, K. Barkaoui, A. Cavalcanti, and A. Cerone, Eds., vol. 4281. Springer, 2006, pp. 183–197.

[32] ——, "Thread-modular verification and Cartesian abstraction," 2006, Thread Verification (TV) workshop.