On Syntactic and Semantic Dependencies in Service-Oriented Architectures

Diego Marmsoler Technical University of Munich
Email: diego.marmsoler@tum.de

***Abstract*—In service oriented architectures, components provide services on their output ports and consume services from other components on their input ports. Thereby, a component is said to depend on another component if the former consumes a service provided by the latter. This notion of dependency (which we call syntactic dependency) is used by many architecture analysis tools as a measure for system maintainability. With this paper, we introduce a weaker notion of dependency, still sufficient, however, to guarantee semantic independence between components. Thereby, we discover the concepts of weak and strong semantic dependency and prove that strong semantic dependency indeed implies syntactic dependency. Our alternative notion of dependency paves the way to more precise dependency analysis tools. Moreover, our results about the different types of dependencies can be used for the verification of semantic independence.**

***Index Terms*—Syntactic Dependency; Semantic Dependency; Service Oriented Architectures;**

I. Introduction

The architecture of a system describes its overall organization into components and connections between these components. In service oriented architectures (SOA), components provide services on their output ports which are realized by consuming services of other, connected components on their input ports [1], [2]. In such architectures, a component A is said to depend on another component B, if an input port of A is (transitively) connected to any output port of B. The notion of dependency is important in software architecture since it is used by many architecture analysis tools [3], [4], [5], [6], especially for maintainability analyses.

However, as the following example shows, not every such dependency is indeed influencing the semantics of a component. Assume, for example, a component A is offering some service *mult* implemented as follows:

```
int mult(int x, int y) {
  int z:=B.add(x,y);
  return x∗y;
}
```

In this simple example, service *mult* provided by component A consumes a service *add* from another component B and stores its results into variable z. Therefore, according to the traditional definition of dependency, component A would depend on component B. However, *mult* does not use the result of *add* to compute its own result. Thus, changing the implementation of *add* (provided by component B) would not have any impact on the semantics of service *mult* (provided by component A). Thus, syntactic dependency does not necessarily impact the semantics of components and dependency analysis tools which rely on syntactic dependency may be improved by considering semantic dependency instead.

Approach: To address this problem, we provide a weaker notion of dependency and investigate its relationship to syntactic dependency. Therefore, we first formalize the notion of syntactic and semantic dependency in terms of a formal model of service oriented architectures. Then, we investigated their relationships by proving properties about it. Thereby, we discover the concepts of weak and strong semantic dependency and prove, i.a., that strong semantic dependency indeed implies syntactic dependency.

Contributions: The contribution of this paper is twofold:

- First, it provides a formal characterization of the notion of syntactic and semantic dependency in service oriented architectures.
- Moreover, it provides several, theoretical results about the relationship between these two concepts.

Thereby, to the best of our knowledge, this is the first formalization of these two important notions of software architectures. Moreover, it is the first work which formally proves that strong semantic dependency indeed requires syntactic dependency.

Implications: The formal characterization of semantic dependency may be used to improve architecture analyses. Moreover, the theoretical results provide us with a powerful tool to verify semantic independence of certain components.

Overview: The remainder of the paper is structured as follows: In Sec. II, we introduce our model of service oriented architectures which serves as a foundation to characterize syntactic and semantic dependency. The different types of dependencies are then characterized and analyzed in Sec. III. The paper concludes with a review of related work in Sec. IV and a brief summary of the major results as well as a brief outlook in Sec. V.

II. A Model of Service-Oriented Architectures

In [1], [2] we introduce an abstract model of service-oriented architectures. In the following, we summarize the main concepts and notations such as ports which can be

valuated by services (Sec. II-A), components (Sec. II-B), architecture configurations (Sec. II-C), and the semantics of a component within an architecture configuration (Sec. II-D).

A. Ports and Services

For our model of service-oriented architectures, we assume the existence of sets $\mathscr{P}$ and $\mathscr{S}$ 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 $\texttt{type}\colon \mathscr{P} \to \wp(\mathscr{S})$, (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 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:

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

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

To provide a better intuition for our model, in the following, we provide some examples about concrete instantiations of the model. Thereby, we denote with $\mathbb{N}^+$ be the set of positive integers and $\mathbb{Z}$ the set of all integers.

Example II.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 $\mathit{bint} = [-2^M, 2^M{-}1]$, 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 $\mathit{bint}{\times}\mathit{bint}$ to bint:

- $\texttt{type}(i_1)$ is the singleton set containing exactly the modular addition, which is defined for all arguments.
- $\texttt{type}(i_2)$ 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.
- $\texttt{type}(o)$ 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.

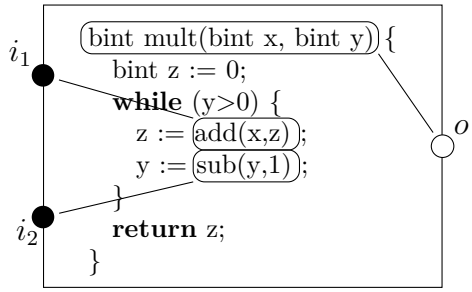

Figure 1: Instantiation of the model with stateless services.

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 $\mathit{bint}{\times}\mathit{bint}$ 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. As shown in [7], 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* [8] of concrete return values.

B. 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. In the following, we are not interested in all details about a component. Rather, we assume the existence of a set $\mathscr{C}$ containing all components.

In our model, ports can be reused by different components which leads to the notion of *component port*, i.e., a port combined with the corresponding component. For a set of component ports $P \subseteq \mathscr{C} \times \mathscr{P}$, a valuation is a function from the set P to the set of services that respects the corresponding port type. By $\overline{P}$ we denote the set of all valuations for P, formally,

$$\overline{P} \;=\; \big\{\mu \in (P \to \mathscr{S}) \mid \forall p \in P\colon \mu(p) \subseteq \texttt{type}(p)\big\}\;.$$

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 ($i{\in}\mathbb{N}_0$, $i{\leq}n$).

Now we can define the concept of a component setup.

Definition II.2 (Component setup)**.** A component setup is a triple (C, in, out, bhv), consisting of:

- a set of components $C \subseteq \mathscr{C}$,
- a mapping to assign a set of input ports to a component $in\colon C \to \wp(\mathscr{I})$,
- a mapping to assign a set of output ports to a component $out\colon C \to \wp(\mathscr{O})$, and
- a mapping to assign a behavior to each component $bhv\colon c \in C \mapsto \left(\overline{in(c)} \to \wp\left(\overline{out(c)}\right)\right)$.

For a component setup $S = (C, in, out, bhv)$ and a subset of components $C' \subseteq C$, we define the notion of port selection to obtain the ports of components:

$$\Pi_{\mathrm{i}}(S, C') \stackrel{\text{def}}{=} \bigcup_{c \in C'} \big(\{c\} \times in(c)\big), \text{ and}$$
$$\Pi_{\mathrm{o}}(S, C') \stackrel{\text{def}}{=} \bigcup_{c \in C'} \big(\{c\} \times out(c)\big).$$

To select all component ports of a component setup S, we write

$$\Pi(S, C') \stackrel{\text{def}}{=} \Pi_{\mathrm{i}}(S, C') \cup \Pi_{\mathrm{o}}(S, C').$$

For the case $C' = C$ we just write $\Pi_{\mathrm{i}}(S)$, $\Pi_{\mathrm{o}}(S)$, and $\Pi(S)$, respectively. □

C. Architecture Configuration

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 II.3 (Architecture Configuration)**.** An (architecture) *configuration* is a pair (S, A) consisting of a component setup S and a so-called *attachment* $A \in (\Pi_{\mathrm{i}}(C) \dashrightarrow \Pi_{\mathrm{o}}(C))$, such that, 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:

$$\forall\ (p_i, p_o) \in A\colon \quad \mathtt{type}(p_o) \subseteq \mathtt{type}(p_i). \qquad \square$$

Note that the domain of an 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. Moreover, the attachment is modeled as a *partial* map which means that not necessarily all input ports are internally connected. These, unconnected, input ports are called *open input ports* and they are obtained by the following mapping:

$$\Pi_{\mathrm{in}}((S, A), C') \stackrel{\text{def}}{=} \{i \in \Pi_{\mathrm{i}}(S, C') \mid i \notin \operatorname{dom} A \vee A(i) \notin C'\}$$

Again, for all C' we simply write $\Pi_{\mathrm{in}}((S, A)) = \Pi_{\mathrm{i}}(S) \setminus (\operatorname{dom} A)$.

D. Composition

In the following, we define the semantics of a component within an architecture configuration. Therefore, we first introduce the concept of architecture valuation. In the following, we write $f|_Z$ for the restriction of a (partial) map f to the domain $(\operatorname{dom} f) \cap Z$.

Definition II.4 (Architecture valuation)**.** Given an architecture configuration $z = (S, A)$ over a component setup $S = (C, in, out, bhv)$ and a set of components $C' \subseteq C$, a valuation $\nu \in \overline{\Pi(S, C')}$ is an *architecture valuation* iff the following conditions hold:

- ν respects every connection in A which is closed under $C'\colon \forall (i, o) \in \big(A \cap (\Pi_{\mathrm{i}}(S, C') \times \Pi_{\mathrm{o}}(S, C'))\big)\colon \nu(i) = \nu(o)$ and
- ν respects the behavior of every component in C': $\forall\, c' \in C'\colon\ \nu|_{out(c')} \in bhv(c')(\nu|_{in(c')})$.

The set of all architecture valuations for a configuration z and set of components C' is denoted $\overline{C'_z}$. If $C' = C$ we just write $\overline{z}$. □

Now we can use the notion of architecture valuation to define the semantics of the composition.

Definition II.5 (Composition)**.** Given a configuration $z = (S, A)$ over a component setup $S = (C, in, out, bhv)$, the semantics of a component $c \in C$ is given by a map $[\![c]\!]_z\colon \overline{\Pi_{\mathrm{in}}(z)} \to \wp\left(\overline{out(c)}\right)$, defined as

$$\mu \mapsto \big\{(\lambda p \in out(c).\ \nu(c, p)) \mid \nu \in \overline{z}\ \wedge\ \nu|_{\Pi_{\mathrm{in}}(z)} = \mu\big\}\ . \qquad \square$$

Intuitively, given a valuation μ of the open input ports, an element of the component semantics $[\![c]\!]_z(\mu)$ is created by restricting an architecture valuation ν that is consistent with μ on the configuration's input ports $\Pi_{\mathrm{in}}(z)$.

Note II.6. Whenever $[\![c]\!]_z(\mu) = \emptyset$ for one component, it follows that $[\![c']\!]_z(\mu) = \emptyset$ for all components.

We say that an architecture configuration z is *consistent* whenever for each $\mu \in \overline{\Pi_{\mathrm{in}}(z)}$ there exists a $c \in C$, such that $[\![c]\!]_z(\mu) \neq \emptyset$.

III. Dependencies

In the following, we introduce and study the concepts of syntactic and semantic dependency. We begin with the notion of syntactic dependency.

A. Syntactic Dependency

Syntactic dependency of components is induced by the attachment relation of a configuration. We say that a component c *syntactically* depends on another components c', if any input port of c is connected to any output port of c'.

Definition III.1 (Syntactic dependency)**.** Syntactic dependency for a configuration $z = ((C, in, out, bhv), A)$ is a relation $\overset{z}{\square\!\!\rightarrow} \subseteq\ C \times \wp(C)$ defined by

$$c \overset{z}{\square\!\!\rightarrow} C' \stackrel{\text{def}}{\iff} \exists i \in \Pi_{\mathrm{i}}(z, \{c\}), o \in \Pi_{\mathrm{o}}(z, C')\colon (i, o) \in A\ . \qquad \square$$

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.

For a configuration z, we denote by $\overset{z}{\square\!\!\rightarrow}_+$ the *transitive* closure of $\overset{z}{\square\!\!\rightarrow}$ and by $\overset{z}{\square\!\!\rightarrow}_*$ the *reflexive-transitive* closure of $\overset{z}{\square\!\!\rightarrow}$. Moreover, we denote by $[\_]\ \overset{z}{\square\!\!\rightarrow}_*\colon C \to \wp(C)$, defined via

$$c\ \overset{z}{\square\!\!\rightarrow}_*\ \overset{\text{def}}{=}\ \{c' \in C \mid c\ \overset{z}{\square\!\!\rightarrow}_*\ c'\} \qquad \text{for } c \in C\,,$$

all components c' that a given component c syntactically (transitively) depends on.

B. Semantic Dependency

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 for the discussion of semantic dependency is the notion of semantic update. We model such a change of the semantics of a component through an update function.

Definition III.2 (Semantic update)**.** For a component setup $S = (C, in, out, bhv)$, a subset of components $C' \subseteq C$, and a family of maps $(f_c)_{c\in C'}$ with $f_c\colon \overline{in(c)} \to \wp\left(\overline{out(c)}\right)$, a *semantic update* $[S \triangleleft f]$ is a component setup (C, in, out, fun'), such that

$$fun'(c) = \begin{cases} f_c & \text{if} \quad c \in C' \\ bhv(c) & \text{else}\ . \end{cases} \qquad \square$$

The notion of semantic update easily generalizes to configurations: $[(S, A) \triangleleft f'] \ \overset{\text{def}}{=}\ ([S \triangleleft f'], A)\ .$

Note that since an update does not change the interface of a component, an update of an architecture configuration is still an architecture configuration according to Def. II.3.

2) Semantic Dependency: Definition: Using the notion of semantic update, we can now define the concept of semantic dependency. Informally, a component c semantically depends on a component c' if updating c' may influence the semantics of c in an architecture configuration z. Note that the semantics of a component may be independent from *isolated* semantic changes of other components, but it may change if the semantics of these components changes *simultaneously*. Thus, the definition of semantic dependency needs to consider sets of components, rather than single components:

Definition III.3 (Semantic dependency)**.** *Semantic dependency* for a configuration $z = ((C, in, out, fun), A)$ is a relation $\overset{z}{\square\!\!\Rightarrow} \subseteq\ C \times \wp(C)$ defined by

$$\begin{aligned} c\ \overset{z}{\square\!\!\Rightarrow}\ C' \overset{\text{def}}{\iff} &(\exists (f_c)_{c\in C'}\colon [\![c]\!]_z \neq [\![c]\!]_{[z\triangleleft f]}) \quad \wedge \\ &\nexists C'' \subseteq C',\ (f_c)_{c\in C''}\colon [\![c]\!]_z \neq [\![c]\!]_{[z\triangleleft f]}\ . \end{aligned} \qquad \square$$

3) Relating Syntactic and Semantic Dependencies: As shown in the introduction, syntactic dependency does not necessarily imply weak semantic dependency. However, one may expect that semantic dependency indeed implies syntactic dependency. However, as the following example shows, this is not the case, in general.

Figure 2: Architecture configuration with two components.

Example III.4 (Why weak semantic dependency does not necessarily imply syntactic dependency)**.** Consider the architecture configuration z depicted in Fig. 2. According to Def. II.5, the semantics of the composition is given by a valuation $\nu\colon \overline{\{(C_1, o), (C_2, o)\}}$, such that $\nu((C_1, o)) = S_1$ and $\nu((C_2, o)) = S_2$ and the restriction to the output ports of C_1 provides us with the semantics of C_1 in z: $[\![C_1]\!]_z = \{[(C_1, o) \mapsto S_1]\}$.

Now assume that we are updating component C_2 with the empty behavior function. Note that it is not possible to satisfy Def. II.4 by any valuation $\nu\colon \overline{\{(C_1, o), (C_2, o)\}}$, which is why the semantics of C_1 in z is given by the empty set: $[\![C_1]\!]_{z[C_2\triangleleft\emptyset]} = \emptyset$.

Thus, the semantics of C_1 in the original configuration is different from the semantics of C_1 after updating C_2 which is why C_1 depends semantically on C_2 according to Def. III.3. However, since we do not have any connection between the ports of C_1 and C_2, they are syntactically independent according to Def. III.1. $\square$

In this example we leverage the fact that Def. III.2 allows an update with the empty function. Note, however, that a similar example can be given without using an empty update. Instead we would use an update which creates an inconsistency in a part of the configuration which is not connected.

The intuition behind a semantic dependency is that this kind of dependency also considers compilation issues. If one component does not compile, than the whole architecture does so, as well.

C. Strong Semantic Dependency

In the following, we define a stronger, more fine-grained notion of semantic dependency. We start with an auxiliary definition.

For a component setup (C, in, out, bhv), a subset of components $C' \subseteq C$, and a family of maps $(f_c)_{c\in C'}$ with $f_c\colon \overline{in(c)} \to \wp\left(\overline{out(c)}\right)$, we say that the updating of C' in z by f is *consistency preserving* iff

$$\begin{aligned} &\left(\forall\, \mu \in \overline{\Pi_{\text{in}}(z)}\colon [\![c]\!]_z(\mu) \neq \emptyset\right) \\ &\qquad \implies \left(\forall\, \mu \in \overline{\Pi_{\text{in}}(z)}\colon [\![c]\!]_{[z\triangleleft f]}(\mu) \neq \emptyset\right)\ , \end{aligned} \tag{1}$$

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

We can use the concept of a consistency preserving update to introduce the notion of strong semantic dependency. Informally, a component c strongly semantically

depends on a component c' if a consistency preserving update of c' may influence the semantics of c in z.

Definition III.5 (Strong semantic dependency)**.** Strong semantic dependency for a consistent configuration $z = ((C, in, out, fun), A)$ is a relation $\overset{z}{\boxdot\!\!\Rightarrow} \subseteq C \times \wp(C)$ defined by

$$c \overset{z}{\boxdot\!\!\Rightarrow} C' \overset{\text{def}}{\iff} (\exists (f_c)_{c\in C'} \colon [z \lhd f] \text{ is cp} \wedge [\![c]\!]_z \neq [\![c]\!]_{[z\lhd f]}) \quad \wedge$$
$$\nexists C'' \subseteq C',\ (f_c)_{c\in C''} \colon [z \lhd f] \text{ is cp} \wedge [\![c]\!]_z \neq [\![c]\!]_{[z\lhd f]} \ . \quad \square$$

Note III.6. Note that strong semantic dependency is only defined for consistent architecture configurations.

1) Relating Weak and Strong Semantic Dependency: As expected, strong semantic dependency implies weak semantic dependency.

Proposition III.7. *Given a consistent architecture configuration $z = ((C, in, out, fun), A)$, we have:*

$$(c \overset{z}{\boxdot\!\!\Rightarrow} C') \implies (c \overset{z}{\square\!\!\Rightarrow} C') \ . \quad \square$$

As the following example shows, the backward direction does not hold in general.

Example III.8 (Why weak semantic dependency does not imply strong semantic dependency?)**.** Consider again the architecture configuration depicted in Fig. 2. According to Ex. III.4, C_1 is weakly semantic dependent on C_2. Moreover, recall, that the semantic update used in Ex. III.4 to demonstrate weak semantic dependency was non consistency preserving. Indeed, there is no consistency preserving update $f \in (\overline{C_2.in} \to \wp(\overline{C_2.out}))$, such that $[\![C_1]\!]_z \neq [\![C_1]\!]_{z[C_2 \lhd f]}$. $\square$

2) Relating Syntactic and Strong Semantic Dependency: Since strong semantic dependency implies weak semantic dependency (Prop. III.7) and syntactic dependency does not necessarily imply weak semantic dependency, we can conclude that syntactic dependency does not necessarily imply strong semantic dependency either.

Lemma III.9. *Given an architecture configuration $z = (S, A)$ over component setup $S = (C, in, out, fun)$ and a set of components $C' \subseteq C$. Then,*

$$(\nu' \cup \nu'') \in \overline{z} \iff$$
$$\nu' \in \overline{C'_z} \ \wedge \ \nu'' \in \overline{(C \setminus C')_z} \wedge$$
$$\forall (i, o) \in A \cap \big(\Pi_{\mathrm{i}}(S, C') \times \Pi_{\mathrm{o}}(S, C \setminus C')\big) \colon \nu'(i) = \nu''(o) \wedge$$
$$\forall (i, o) \in A \cap \big(\Pi_{\mathrm{i}}(S, C \setminus C') \times \Pi_{\mathrm{o}}(S, C')\big) \colon \nu''(i) = \nu'(o) \ .$$

Applying the lemma to the syntactic transitive closure leads to a lemma which turns out to be useful later on.

Lemma III.10. *Given an architecture configuration $z = (S, A)$ over component setup $S = (C, in, out, bhv)$ and a component $c \in C$. Then,*

$$(\nu' \cup \nu'') \in \overline{z} \iff$$
$$\nu' \in \overline{\big(c \overset{z}{\square\!\!\to}_* \big)_z} \ \wedge \ \nu'' \in \overline{\big(C \setminus c \overset{z}{\square\!\!\to}_* \big)_z} \wedge$$
$$\forall (i, o) \in A \cap \big(\Pi_{\mathrm{i}}(S, C \setminus (c \overset{z}{\square\!\!\to}_*)) \times \Pi_{\mathrm{o}}(S, c \overset{z}{\square\!\!\to}_*)\big) \colon$$
$$\nu''(i) = \nu'(o) \ .$$

Proof sketch. Follows from Lem. III.9, since there exists no input port of $c \overset{z}{\square\!\!\to}_*$ which is connected with any output port of $C \setminus (c \overset{z}{\square\!\!\to}_*)$. $\square$

This lemma can now be used to proof the following key result of this paper.

Theorem III.11. *For a consistent architecture configuration $z = (S, A)$ over component setup $S = (C, in, out, bhv)$, strong semantic dependency implies the transitive closure of syntactic dependency:*

$$\forall c \in C, C' \subseteq C \colon \big(c \overset{z}{\boxdot\!\!\Rightarrow} C' \implies c \overset{z}{\square\!\!\to}_* C'\big)$$

The contrapositive provides us with a powerful rule which is often employed in software architecture:

> **Corollary III.12.** *If a component does not transitively connect to another component, it is guaranteed that it does not semantically depend on that component!*

One final interesting observation is, that a component may be semantic independent of one component and of another component but semantically dependent on both!

D. Dependencies: Summary

In the following, we summarize our results about the relationships between syntactic and semantic dependency:

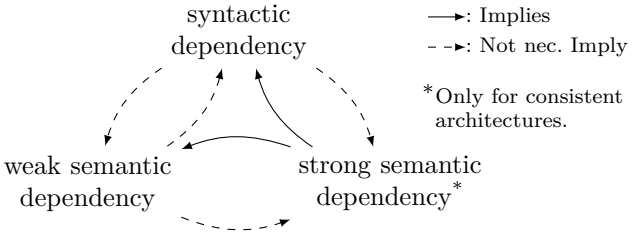

Figure 3: Relating syntactic, weak semantic, and strong semantic dependency.

IV. Related Work

To the best of our knowledge, this paper provides the first formalization of semantic dependency of components and results about its relation to syntactic dependency. However, related work can be found in two related areas:

One source for related work can be found in code dependency analyses. One of the first works in this area is Podgurksi and Clarke [9] which provide formal definitions of weak and strong syntactic dependency between program statements. Moreover, they provide also a notion of semantic dependency between statements. In their work about the Dependent Core Calculus, Abadi et. al [10] provide a calculus of dependency to reason about different types of program dependencies. While many of the ideas in these works are similar to our work, these works are concerned with dependenciess at the level of code and therefore they cannot be used at the architecture level. With our work,

we provide a more abstract characterization at the level of components which allows our results to be applied to architecture analyses.

Related work can also be found in studies of dependencies at the architectural level. An example of work in this area comes from Gu et al. [11] which use the concept of Port Dependency Graph as a notion of component dependency. Another approach in this area comes from Guo [12] who applies category theory to model software component dependencies. While these approaches indeed investigate dependencies at the architecture level, they focus only on the notion of syntactic dependencies. With our work we complement these works by providing a characterization and analysis of semantic dependency.

One exception to the above approaches is Broy's work on traceability [13] in which he introduces a notion of semantic dependency between system specifications. Roughly speaking, two specifications are considered dependent if one implies the other. While this definitions allows to investigate dependencies at the specification level, our work focuses on dependencies between concrete components of an architecture.

V. Conclusion

This paper presents a formalization and analysis of different types of component dependencies. Thereby, the major results of the paper can be summarized as follows:

- We provide a formal characterization of the notion of syntactic dependency in SOA.
- We provide a formal characterization of the notion of semantic dependency in SOA.
  - Thereby we introduce the notion of semantic update.
  - Moreover, we distinguish between weak and strong semantic dependency.
- We show that syntactic dependency does not necessarily imply weak semantic dependency
- We show that weak semantic dependency does not necessarily imply syntactic dependency
- We show that strong semantic dependency implies weak semantic dependency
- We show that weak semantic dependency does not necessarily imply strong semantic dependency
- We show that syntactic dependency does not necessarily imply strong semantic dependency
- As a major result we show that strong semantic dependency implies the transitive closure of weak semantic dependency.

Possible Implications: Taking the contrapositive of the last result provides us with a powerful tool for the verification of semantic independence in service oriented architectures.

Moreover, the results provided in this paper can serve as a foundation for the development of tools which can be used to investigate semantic dependencies in service oriented architectures.

Future Work: Future work arises in two directions:

- Based on the results provided in this paper, a calculus should be developed which can be used to verify semantic and syntactic dependencies in service oriented architectures.
- Second, the results should be implemented into tools which can be used to investigate semantic dependencies in service oriented architectures.

Acknowledgments: We would like to thank Manfred Broy for useful discussions about the topic. Moreover, we would like to thank Wolfgang Boehm and all the anonymous reviewers of TASE 2018 for comments and helpful suggestions on earlier versions of this paper. Parts of the work on which we report in this paper was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01Is16043A.

References

[1] A. Malkis and D. Marmsoler, "A model of service-oriented architectures," in *Components, Architectures and Reuse Software (SBCARS), 2015 IX Brazilian Symposium on*. IEEE, 2015, pp. 110–119.

[2] D. Marmsoler, "Towards a theory of architectural styles," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, ACM. ACM Press, 2014, pp. 823–825.

[3] C. Pich, L. Nachmanson, and G. G. Robertson, "Visual analysis of importance and grouping in software dependency graphs," in *Proceedings of the 4th ACM symposium on Software visualization*. ACM, 2008, pp. 29–32.

[4] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *ACM Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 167–176.

[5] A. Srivastava, J. Thiagarajan, and C. Schertz, "Efficient integration testing using dependency analysis," Technical Report MSR-TR-2005-94, Microsoft Research, Tech. Rep., 2005.

[6] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 967–977.

[7] D. Marmsoler, A. Malkis, and J. Eckhardt, "A model of layered architectures," in *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2015, London, United Kingdom, April 12th, 2015.*, ser. EPTCS, B. Buhnova, L. Happe, and J. Kofron, Eds., vol. 178, 2015, pp. 47–61. [Online]. Available: https://doi.org/10.4204/EPTCS.178.5

[8] M. Broy, "A logical basis for component-oriented software and systems engineering," *The Computer Journal*, vol. 53, no. 10, pp. 1758–1782, Feb. 2010.

[9] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[10] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1999, pp. 147–160.

[11] Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A model-based approach to system-level dependency and real-time analysis of embedded software," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*. IEEE, 2003, pp. 78–85.

[12] J. Guo, "Using category theory to model software component dependencies," in *Engineering of Computer-Based Systems, 2002. Proceedings. Ninth Annual IEEE International Conference and Workshop on the*. IEEE, 2002, pp. 185–192.

[13] M. Broy, "A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views," *Software & Systems Modeling*, pp. 1–29, 2017.