Flexible Component Composition through Communication Abstraction

Fabian Gilson and Vincent Englebert

PReCISE Research... of system architectures may be deployed on a target in-frastructure, also specified with dedicated constructs.

Flexible Component Composition through Communication Abstraction
Fabian Gilson and Vincent Englebert
PReCISE Research... 978-989-758-168-7
Copyright c⃝2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Keywords: Software Architecture, Component Composition, Communication Abstraction, User-defined Properties.

Abstract: Software architectures are often abstracted as a combination of reusable components connected to each other by various means. Specifications of components’ semantics have been widely studied and many modeling languages have been proposed from coarse-grained loosely-defined elements to operational objects with behavioral semantics that may be generated and executed in a dedicated framework. All these modeling facilities have proven their advantages in many domains through either case studies or real-world applications. However, most of those approaches either consider a subset of composition facilities, i.e. the available types of bindings between components, or do not even consider communication properties at all, staying at behavioral-related compatibility between components. Verifications of communication-related properties are then postponed to the hand of software developers and finally considered at deployment-time only. Part of a general architecture framework, we propose an abstraction formalism to specify communication paths between components. This modeling facility relies on a taxonomy of types of links and the specifications of communication protocols. This protocol serves as a reification element between abstract component compositions, architecture instances and deployment infrastructure, making explicit communication-related constraints and properties.

1 INTRODUCTION

Component-based modeling languages are meant to represent complex system architectures in order to, among other, give a coarse-grained view of systems and ease their understanding by the many stakeholders of such systems. Many definitions of what a component is may be found in the literature. Still, many stick at Szyperski’s definition, stating that “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” (Szyperski, 2002). Composition is clearly highlighted as the central concept of a system, components being unit of compositions. In their seminal paper, Beugnard et al. introduced the notion of contract-aware components where they distinguish between syntactical, behavioral, synchronization and quality of service aspects as key features for component compositions (Beugnard et al., 1999).

Many formalisms have arisen to offer abstractions to these unit of compositions, but surprisingly, quality of services (QoS) properties have been scarcely studied or require to stick to a predefined environment. As examples, the component diagrams of the OMG Unified Modeling Language points to behavioral diagrams in order to specify the underlying communication facility without being able to characterize the communication medium in an abstract way (Object Management Group, 2011). Other approaches are even more formal, requiring modelers to rely on algebraic constructions (de Jonge, 2003; Gössler and Sifakis, 2005), data-only transmissions (Oquendo, 2004; Society of Automotive Engineers, 2012) or even restraining to one particular technological framework (Grinkrug, 2014).

We propose a facility to specify such quality of service properties, on top of a syntactical and a lightweight behavioral specification formalism. Our goal is to provide system designers with easy-to-use and extensible modeling elements to specify component behaviors and properties of communication paths. We rely on an architecture description language (ADL) enhanced by composition constraints. On the one hand, our ADL has been designed to face many shortcomings of current component-based languages and embeds a taxonomy for component bindings. On the other hand, the composition verification mechanism ensures that abstract descriptions of system architectures may be deployed on a target infrastructure, also specified with dedicated constructs.
We also provide a substitution mechanism for components, inspired from the duck-typing in object oriented programming\(^1\).

We start our paper by reviewing some work close to our approach in Section 2. We then give in Section 3, a brief overview of our layered formalism to specify software architectures. We introduce in Section 4 a taxonomy for components’ connections. Afterwards, in Section 5, we describe how designers may extend the semantics of model elements with custom properties. In Section 6, we present the rules applied at all levels of an architecture model to guarantee that a model is valid regarding its specifications. We illustrate in Section 7 our proposal with a fictitious online library system, where we also discuss its advantages. We finally conclude in Section 8 by summarizing the present work and the validation approach we set up previously. We also consider the drawbacks and future research tracks.

2 RELATED WORK

Probably the most widely known of component-based modeling is the UML component diagram (Object Management Group, 2011). As many of other languages, it relies on the definition of components connected to each other via connectors through interfaces. Even if components and interfaces’ semantics may be specified in a detailed manner, connectors may only be refined by behavioral diagrams to further elaborate their definitions, and QoS properties are not addressed directly. Some profiles, like SysML (Object Management Group, 2012), offer such features, but they are mainly domain-specific, which limits their usage to particular fields.

Other approaches like π-ADL (Oquendo, 2004), or AADL (Society of Automotive Engineers, 2012) provide formal constructions to specify, analyze, validate and sometimes deploy component-based specifications. Despite all these advantages, formal methods usually lack of understandability and scalability when models become quite large. Furthermore, QoS properties are not always expressible in an algebraic way.

Alternative languages, like ACME (Garlan et al., 1997), xADL (Dashofy et al., 2005) and Archimate (Open Group, 2013) face the problem the other way around, providing extensible constructs where any kind of properties may be specified for modeling elements. However, either their semantics is so vague that most of it must be specified via user-defined properties, or communication facilities may not be refined by properties.

In our approach, we furnish a set of reusable communication links, with minimal semantics, as well as the possibility to annotate them with user-defined properties. Furthermore, we provide validation rules regarding the communication facilities between platform-independent model, platform-specific models and target deployment infrastructures.

3 LAYERED REPRESENTATION

The present work is part of a larger architecture framework, named pick One, Document And transform Strategy (IODASS)\(^2\). We briefly sum up the main architectural constructs used to support the overall modeling of software systems.

3.1 IODASS Model

The IODASS framework relies on a layered representation of software architectures. Those three IODASS layers are named Definition, Assemblage and Deployment. The Definition layer is used to give abstract specifications of an architecture. Roughly, ComponentTypes are connected to each others through Facets typed by Interfaces via LinkTypes. Interfaces gather semantically correlated Services and statically typed Parameters. These connections are called LinkageTypes. LinkTypes support one or more Protocols. This separation between the type of binding for components (point-to-point, for example) and the communication protocol itself (HTTP, for example) will be further discussed in this paper. Within a Definition layer, infrastructure-related constructs may also be specified in order to describe the target deployment environment. Those are the NodeTypes, i.e. any type of computing node, Gates being the physical interaction ports on these nodes, typed by GateTypes and, last, MediumTypes that are concrete links binding nodes (e.g. network facilities). A Definition may be seen as a structural architectural style, completed by the representation of a deployment infrastructure.

The Assemblage layer is a particular instantiation of a Definition, with some additional constraints.

\(^1\)Duck-typing is a technique in object oriented programming where the semantics of a method relies on its signature, instead of being specified by the inheritance of a particular interface or class. It has been named in reference to the poet James Whitcomb Riley for his well-known quote “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

\(^2\)More details may be found in (Gilson, 2015)
SetOfInstances are connected via their Ports in Linkages. At this layer, running-time constraints may be specified, such as creation relationships. An Assemblage is somewhat comparable to a particular instantiation of an architectural style.

The Deployment layer specifies a mapping of the Assemblage on a representation of the target infrastructure, the infrastructure elements being specified at the Definition layer too. Here, modelers can describe a particular infrastructure of nodes and cables (or any other type of communication medium), and also write a set of rules to specify what kind of instances may be deployed on what kind of computation nodes.

As we will discuss in Section 5, any modeling element may be refined by user-defined properties in order to add, among other, QoS properties. Those properties are particularly valuable when modelers want to compare required properties of services, for example in terms of communication throughput, and the characteristics of the target deployment infrastructure.

Note that the Protocol is a central concept in our modeling language. At each layer, any communication facility that describes the type of binding, i.e. topological constraints between inter-connectionable elements, is always related to this concept of Protocol, specifying, those QoS-related properties.

### 3.2 Model Element Extensions

Many ODASS constructs may inherit from other constructs of the same type. Inheritance mechanism is required for substitution purposes in order to ease model evolution. ComponentTypes may inherit from multiple other ComponentTypes, and may be composed by inner-configurations, being sub-architecture models. A complex ComponentType may be seen as a black-box with visible interfaces and a particular hidden implementation. The inheritance relationship means that the sub-ComponentType gathers all external Facets from its super-ComponentTypes and may add other externally visible Facets. It may also overwrite its inner-configuration completely, i.e. providing a different implementation. Validity of substitutions between ComponentTypes will be formalized in Section 6 when we discuss the validity of linkages between components.

Apart ComponentTypes, all other elements may inherit from only one supertype. We restricted to single inheritance for Protocols, LinkTypes, MediumTypes, GateTypes and NodeTypes since for all of them, multi-inheritance was unreasonable. For instance, it would have make no sense for a particular NodeType where hardware properties have been specified, like the CPU clock or amount of memory, could inherit from multiple nodes with orthogonal specifications. Analogous reasoning may be applied to all these elements.

### 4 A TAXONOMY OF LINK TYPES

The flexibility of component compositions is achieved via the separation of topological constraints between components, i.e., how they are connected, and the specification of communication properties, notably QoS constraints. To this end, we created a taxonomy of available link types used to bind ComponentTypes to each others, and by extension, between SetOfInstances since their inter-connections are constrained by the style defined at the Definition layer. Figure 1 presents this taxonomy.

![LinkType taxonomy](image)

**Figure 1:** LinkType taxonomy.

LinkTypes are either ConnectorTypes or DelegationTypes. ConnectorTypes are used to connect two Facets, under a provide-require contract, analogously to the ball-and-socket in UML 2. At the opposite, DelegationType are used to link two Facets of the same polarity, transferring the responsibility of an Interface between two components. Both types of LinkTypes may be specialized in more concrete constructs.

Opposite Facets may be bound to each other in a point-to-point way (One2one), multicast way (One2many) or in a Many2many manner. In case of One2many, if the connection is set to be distributive, a One2one connection is set between all instances. In case of multiple targets (and sources), restrictions may apply regarding the returned values of services, since no such assumption can be taken.

Delegations between Facets may be of various types too. A Simple delegation depicts a point-to-point connection strictly between two instances where the delegator will be in charge of providing the implementation for this Facet. The other types of delegation describe a kind of delegation contract when multiple instances are present on the concrete implementation side. Random delegation will simply transfer the service call to a randomly chosen implementation.
Broadcast means that all implementations are called either in a point-to-point or in a multicast way. If the connection is forward, all called implementations are required to answer. Other LoadBalancing strategies may also be specified. Return types for Services are allowed for all but the Broadcast DelegationType.

These LinkTypes only concentrate on the possible topology, but does not say anything regarding the properties of the connection itself which are under the responsibility of the Protocol.

5 USER-DEFINED PROPERTIES

For any type of IODASS construct, particularly Protocols, user-defined properties may be created, as presented in Listing 1.

```
1 package be.ioidass.sample;
2
datatypes network {
3   enum CommunicationLayer {
4     physical, datalink, network, transport, session,
5     presentation, application, namespace }
6   property for protocol layer {
7     type CommunicationLayer;
8     semantics "Communication layer: seven OSI layers with an
9     extra one for processes sharing the same memory space";
10   }
11   property for mediumtype DataTransmission {
12     type decimal; unit "Mbps";
13     semantics "Maximum transmission rate in Mbps";
14   }
15   property for mediumtype Bandwidth {
16     type integer; unit "MHz";
17     semantics "Maximal bit rate";
18   }
```

Listing 1: Sample property definitions.

A property is statically typed, targets a IODASS construct and its semantics must be described. Predefined types, such as int, or string are available, but enumerated values may also be specified (like in the above example for CommunicationLayer). Groups of properties may be created too, in order to gather correlated properties in one element. This simple mechanism allows models to refine the semantics of many model elements as well as perform some validation on properties where, for example, a ComponentType requires a specific amount of disk space or CPU power that must be provided by its deployment target.

At present time, even if the verification must be performed manually, we formalized a couple of consistency checks regarding requirements bound to model elements. Requirement descriptions are attached to any model where the purpose and objectives of model elements is explained\(^3\).

\(^3\) A complete description of the associated requirement models may be found in (Gilson, 2015)

6 ARCHITECTURE VALIDITY

Around the separation between LinkType and Protocol, we are now able to define validation rules for all three layers. Those rules ensure that only valid configurations are specified at the Definition layer, only conforming Assemblies are created and that all Deployment mapping rules respect the needed properties regarding connection requirements.

6.1 Validity of Component Composition

As explained in Section 3, some kind of architecture style must be written at first. This Definition gathers specifications of domain-specific building blocks as well as it constraints valid configurations that may be expressed when instantiating this abstract model.

Informally, at this stage, we must verify that all Services of a source Facet have compatible corresponding Services in the target one. This loose compatibility may be compared to a duck typing system, where two services are considered compatible if at least all needed services with all needed parameters "looks like" being covered by the provider. Matches must be found for all required Services, but the set of provided Services can be larger\(^4\). More formally,

- \(p\) denotes a Parameter, such as \(p=(d, f)\) with \(d\) the parameter direction and \(f\) its GenericType
- \(e\) denotes an Exception
- \(s\) denotes a Service, such as \((s \leftarrow d_1, ..., d_n)\), an ordered list of Parameters followed by Exceptions.
- \(I\) denotes an Interface, such as \(I = [r_1, ..., r_m]\), a list of Services
- \(F\) denotes a Facet
- \(F \leftarrow I\) denotes the is typed by relationship, such as \(F \leftarrow I\)
- \(F \subseteq C\) denotes that the Facet \(F\) is exposed by the ComponentType \(C\)
- \(L\) denotes the directed LinkageType between Facets with a LinkType \(L\)
- \(C\) represents a LeakUsage, such that \(C\)\(\leftarrow C\) denotes \(C\) uses \(C\)

For two Parameters \(p_1\) and \(p_2\), they are considered equivalent, denoted by \(\equiv\), if formally,

\[
p_1 = (d_1, s_1) \quad \land \quad p_2 = (d_2, s_2) \quad \quad p_1 \approx p_2 \quad \iff \quad d_1 = d_2 \land s_1 = s_2
\]

\(^4\) Note that Exceptions may be raised from Services, and their effective handling must also be checked. The Exceptions raised from a Service are also verified, but the check is more stringent here. All Exceptions raised by provided Services must be taken into account by the required Facet.
For two Services, $s_1$ and $s_2$, they are considered equivalent, denoted by $\approx$, if formally, $s_1 = \langle p_1, \ldots, p_n \rangle < \ldots < e \rangle$.

Then, let $C_1$, $C_2$ two ComponentTypes and $l_1$, $l_2$ two Interfaces. If a LinkageType has been defined between two Facets, the following conditions must hold:

$$\forall F_1 \in C_1, F_2 \in C_2 \mid F_1 \vdash I_1 \land F_2 \vdash I_2$$

$$F_1 \overset{l_1}{\rightarrow} F_2 \Rightarrow \forall x_1 \in I_1, \exists x_2' \in I_2 \mid x_1 \approx x_2'$$

### 6.2 Validity of Instance Composition

At the Assemblage layer, the validation of Linkages is analogous to the one for LinkageTypes. For a given Port, we have to retrieve its typing Facet and we have to ensure that the chosen Protocol for this Port belongs to the accepted list of Protocols for the LinkType actually used. Formally, let

- $S$ denotes a SetOfInstance, being a set of Ports
- $P$ denotes a Port and $P \subseteq S$ denotes that $P$ is exposed by $S$
- $\vdash$ denotes the is typed by relationship, such as $(S \vdash C)$
- $T$ denotes a Protocol
- $L$ denotes a LinkType
- $\leftarrow$ denotes the Protocol support, such that $P \leftarrow T$ and $L \leftarrow T$

If a Linkage exists between two Ports with a LinkType $L$, noted by $L$, the following conditions must hold:

$$\forall P_1 \in S_1, P_2 \in S_2 \mid P_1 \vdash I_1 \land P_2 \vdash I_2$$

$$P_1 \leftarrow P_2 \Rightarrow \exists C_1 \mid C_1 \vdash C_2 : S_2 \vdash C_2$$

$$\land \{ F_1 \in C_1 \mid F_1 \vdash l_1 \land F_2 \in C_2 \mid F_2 \vdash l_2 \} \mid F_1 \leftarrow F_2$$

$$\land \exists T \mid L \leftarrow T \land P_1 \leftarrow T \land P_2 \leftarrow T$$

### 6.3 Validity of Instances Deployment

Last, we must ensure that the mapping rules regarding all connections, aka Linkages, are actually supported by the target Deployment infrastructure. First, we have to check that when plugging a communication MediumType between two Gates, both Gates and the Medium support a particular Protocol (the one that will be used for the communication afterwards).

- $N$ denotes a NodeType, being a set of Gates
- $H$ denotes a Node and inherits from the Gates defined in its typing GateType
- $G$ denotes a GateType
- $A$ denotes a Gate and $A \leftarrow H$ denotes that $A$ is exposed by $H$
- $\leftarrow$ denotes the is typed by relationship, such that $H \leftarrow R$ and $A \leftarrow G$
- $M$ denotes a MediumType
- $\rightarrow$ is overloaded such that $(M \rightarrow T)$, $T$ supports the Protocol $T$

If a Plug exists between two Gates with a MediumType $M$, noted by $\rightarrow$, the following conditions must hold:

$$\forall A_1 \in H_1 \land \forall A_2 \in H_2$$

$$A_1 \leftarrow A_2 \Rightarrow \exists T \mid A_1 \leftarrow T \land A_2 \leftarrow T \land M \rightarrow T$$

Now, when abstractly deploying instances on nodes, Ports are bound to Gates that will effectively support the connection between Nodes via a MediumType. We have to guarantee that the Protocol chosen to link both Ports is actually supported by the Gates and MediumType. In other words, we must allege that when deploying a model instance, the target infrastructure will certainly allow the communication between instances. Formally,

- $\leftarrow$ denotes the Deploy rule, such that $(S \leftarrow H)$
- $\rightarrow$ denotes an Opening, such that $(P \rightarrow A)$, the Port $P$ is opened on $A$

Then, the following conditions, must hold:

$$\forall A_1 \in H_1 \land \forall A_2 \in H_2 \land \forall P_1 \leftarrow A_1 \land \forall P_2 \leftarrow A_2$$

$$P_1 \leftarrow P_2 \land A_1 \leftarrow A_2 \Rightarrow \exists S \mid P_1 \in S_1 \land S_2 \mid P_2 \in S_2$$

$$\land S_1 \leftarrow H_1 \land S_2 \leftarrow H_2$$

$$\land \exists T \mid L \leftarrow T \land M \rightarrow T$$

$$\land A_1 \leftarrow T \land A_2 \leftarrow T$$

### 7 DISCUSSION

We will now illustrate our modeling language with a fictitious online library system. In this system, users connect to a webpage where they can browse over a list of books, and possibly buy some. The list of available books is the conjunction of all catalogs sent by partner bookstores. When a book is bought on the library, an auction is conducted between all stores providing the book to find the cheapest price among them. A delivery service will then come at the winning store to bring the book at the customer’s place.
A naive graphical representation of the architecture Definition is shown in Figure 2.

Listing 2 shows the corresponding textual model5.

```java
package be.iodass.onlinelibrary;

dadmodel onlinelibrary {
    definition {
        struct Book { /* book details */ }
        interface BookSelling {
            sync Book[] browseCatalog();
            sync boolean buyBook(int isbn);
            event confirmDelivery(Book b);
        }
        interface BookCatalog { sync Book[] getBookCatalog(); }
        interface Auction {
            sync float getPrice(int isbn, int float price);
        }
        interface Delivery {
            sync boolean deliverBook(Book b);
        }
        componentType Customer { uses BookSelling as bs; }
        componentType Library { implements BookSelling as bs; /* for customer */
            uses BookCatalog as bc; /* to stores */
            uses Auction as a; /* to stores */
            uses Delivery as d; /* to delivery */
            // up to 1000 simultaneous customers
        }
        componentType Bookseller { implements BookSelling as bs;
            implements Auction as a; }
        componentType Delivery { implements Delivery as d; }
        componentType One2one { mode one2one; }
        componentType One2many { mode one2many; distributive: true; }
        // up to 10 request can be handled at a time
        // up to 1000 simultaneous customers
        // the unloader is unique
    }
    // customer - bookseller,
    // library - bookseller, store
    // bookseller - library, store
    // library - store
    // store - delivery
    "layer" : Application, "reliable": true, "ordered": true;
}

package be.iodass.onlinelibrary;

dadmodel onlinelibrary_2 {
    definition {
        protocol HTTP {
            "layer" : "application";
            "reliable": true; "ordered": true;
            "mode" : "one2one";
            "accept": "HTTP";
            "connectortype": "One2one"
        }
        protocol MOM {
            "layer" : "application";
            "reliable": true; "ordered": true;
            "mode" : "one2many";
            "accept": "MOM";
            "connectortype": "One2many"
        }
        protocol DLSS {
            "layer" : "transport";
            "reliable": true; "ordered": true;
            "mode" : "one2one";
            "accept": "DLSS"
        }
        protocol SOAP {
            "layer" : "rpc";
            "reliable": true; "ordered": true;
            "mode" : "one2one";
            "accept": "SOAP"
        }
        protocol HTTP {
            "layer" : "web";
            "reliable": true; "ordered": true;
            "mode" : "one2one";
            "accept": "HTTP"
        }
        "assemblage" {
            "customer" : Library, "library" : Bookseller,
            "bookseller" : Library, "store" : Bookstore,
            "store" : Delivery,
            "library" : bookseller;
            "bookseller" : store;
            "store" : delivery;
            // up to 10 request can be handled at a time
            // up to 1000 simultaneous customers
            // the unloader is unique
        }
    }
    // customer - bookseller - library - store - delivery
```

Listing 3: Assemblage layer (Component instantiation).

For this naive model, connections are simply drawn between ComponentTypes, but no assumptions regarding Protocols have been set. Only the way how connections should be made (point-to-point or not) is stated.

We will now refine the definitions of both connector types illustrated in previous model to make them accept a particular Protocol, also defined in Listing 3. Figure 3 first shows a simplified representation of the Assemblage layer conforming to the (updated) Definition layer given in previous listing.

An Assemblage model is a kind of instantiation of a Definition model completed with some runtime constraints, as briefly stated in Section 3. For example, a library instance may handle up to 1000 customers at the same time, but none of them is required. On the other hand, at least one stores and one delivery instances are needed to make the system run correctly. Linkages are represented by directed arrows, depicting which instance is initiating the call (called instance being at the arrow’s side). Linkages are also annotated by one particular Protocol, chosen from the list of available protocols accepted by the Facets specified at the Definition layer. Listing 3 shows the complete updated model with the new Protocol, the modified ConnectorTypes as well as the Assemblage. Note that, the IODASS modeling language allows model imports (using the import keyword). Existing elements from the imported model (names are fully qualified) are simply overridden by their new definitions in the new model, if any.

5We created an Eclipse plugin available at https://sites.google.com/site/memodiaresearchproject/
instead of using stereotypes, we rely on semantics inheritance for modeling elements. The updated model is given in Listing 4.

```java
package be.iodass.onlinelibrary;
import be.iodass.onlinelibrary.library;
import be.iodass.onlinelibrary.library_2;

// create a gatetype that accepts HTTP requests

gatetype Ethernet { supports HTTP; }

// node types

nodeType Client { Ethernet eth1; }
nodeType BasicServer { Ethernet eth0, eth1; }
nodeType Gateway { Ethernet [3] eth; }
nodeType Server extends BasicServer { Ethernet eth1; }

// CPU: 3.2 GHz; CPUCores: 4,
CPUArchitecture: "Intel Itanium";

// two types of media

mediumType Cable { supports Bandwidth, DataTransmission; }
mediumType E100BaseT extends Cable { Bandwidth: 100, DataTransmission: 100; }

// deployment
node server[0..19]::Server; node client[0..100]::Client;

deploy client[0..100]::Client;
deploy server[0..19]::Server; deploy delivery.d on server[20];
deploy gateway::Gateway; deploy customer[0..999] on client[0..999]; deploy library.on server[0];
deploy stores on server[0..19];
site TheOffice (contains server, gateway;)

// open ports on network gates
open library.on server::eth0; // customer
open library.on server::eth1; // book catalog
open store.on server::eth1; // auction
open store.on server::eth0; // library
open library.on server::eth1; // delivery
open delivery.d on server[20]::eth0;
bind nodes to each other

plug Cable from client[0..999]::eth0 to gateway::gengate;
plug E100BaseT from server::eth0 to gateway::eth[0];
plug E100BaseT from server::eth1 to gateway::eth[1];
plug Cable from gateway::gengate to server[0..21]::eth0;
}
```

Listing 4: Component Deployment (textual representation).

In the above listing, a GateType is created in order to type the Gates available on the NodeTypes. Those Gates express, among others, what kind of Protocol they accept. For example, an Ethernet port on a personal computer will not be able to accept WiFi connections. The GateType is literally representing this constraint, any type of physical port on a computer (or any other likewise device) are not able to receive any type of communication. Moreover, even if this port supports a particular protocol, more stringent constraints may exist at the Definition or Assemblage layers regarding needed data transmission rate, for instance. Here again, user-defined properties serve the role of validation helpers for these kind of constraints.

For illustration purposes, we also added such properties to the Server to refine its semantics in terms of computation architecture. It also inherits from the BasicServer, meaning that it owns two Ethernet Gates. Exactly as model elements may be completely overridden (when importing a model), specific properties may be overridden too, like the DataTransmission in the E100BaseT MediumType.

The Deployment clause in the above model describes then one concrete deployment of the Assemblage onto a target infrastructure. In short, instances must be deployed on Nodes, their Ports must be opened on Gates and MediumTypes must be plugged into those Gates. Shorthand rules may contain indexes interval in order to limit the number of lines of code to write (as in lines 53, 57 and 63).

Hence, the overall communication paths, from the Definition layer to the Deployment layer is reified around the concept of Protocol, making possible to specify a very wide range of communication facilities and, in some ways, validate the conformance between an abstract architecture model, a particular instantiation and its target deployment infrastructure. The Protocol acts as a central point used in both the Assemblage and Deployment layers around which connection facilities are bound, such that any constraint expressed in connection paths and their connected elements (ComponentTypes, SetOfInstances or Nodes) can be verified. Since properties are statically typed (sometimes with an ordering rule), verifications may be performed between needed properties (expressed in the aforementioned requirement models in Section 5) and actual properties offered by a model.

8 CONCLUSIONS, LIMITATIONS AND FUTURE WORK

A recent survey regarding the need of modeling support for software architecture conducted in the industry showed that, even if many theoretically powerful frameworks have been proposed for the last twenty years, the most popular language was still UML2 component diagrams (Malavolta et al., 2013). This survey also highlighted that, (non-) practitioners are also interested in iterative design support, model versioning and analysis, and also in the ability to represent a wide range of architecture and communication facilities using the same tweakable formalism.
In previous work, we detailed our contribution regarding the design support and model versioning, but we did not demonstrate how our architecture framework may help architects for analysis and flexible component compositions. In the present work, we explained how components may be substituted by other ones having compatible external definitions using a special kind of inheritance mechanism inspired from the object-oriented duck-typing concept. We also provided a taxonomy of link types between components for both provide-require contracts as well as for delegation ones. We introduced a clear separation between topological configurations of components, supported by the aforementioned taxonomy, and the behavioral or Quality of Service properties whose responsibility is transferred in a dedicated protocol modeling construct.

The overall framework has been subject to an empirical validation, even if its size was rather small and the participants were master students (Gilson, 2015). However, despite its limitations, the case study highlighted some advantages in terms of model completeness and in terms of expressiveness regarding modeling communication facilities.

Tool support is provided for the overall framework, as an Eclipse plugin, enabling designers to write and transform models in order to refine, update or modify them in a structured and fully traceable way. The validation of composition and substitutions presented in this paper are fully part of that tool.

Still, the mechanism provided to write user-defined properties has been thought in an automatic way, validation of properties is, at present time, a manual task. When the size of model increases, such an automated validation is particularly needed, especially when model elements may be substituted easily (by model transformations in our case). However, this validation feature can be added in the tool support with reasonable effort, especially because we already provided a theoretical ground regarding consistency verifications between properties. Combined to the user-defined properties, other behavioral specifications could be envisioned.

At present time, only static relations between properties are defined, but the property mechanism could be extended to specify relations between properties of different constructs. For example, some kind of relation could be expressed between the response time of a server, the data transmission rate of its connected communication medium and its CPU capacity. Such complex relations would help the aforementioned validation feature to handle possible problems in architecture models for correlated properties, without requiring to specify each time the mapping between those properties.

Last, no inheritance exists for data types (e.g. interfaces, parameter types or data structures). Such inheritance would raise the flexibility for component composition and substitution, but would require to enhance our compatibility verifications to handle co-and contravariance of data types.

REFERENCES


