Reusable Connectors in Component-Based Software Architecture

Abdelkrim Amirat, Mourad Oussalah

Reusable Connectors in Component-Based Software Architecture

Abdelkrim Amirat and Mourad Oussalah
LINA Laboratoy, CNRS UMR 6241
University of Nantes, France
{abdelkrim.amirat;mourad.oussalah}@univ-nantes.fr

Abstract

In component-based system, connectors are used to compose components. Connectors should have a semantics that makes them simple to construct and use. At the same time, their semantics should be rich enough to endow them with desirable properties such as genericity, compositionality and reusability. For connector construction, compositionality would be particularly useful, since it would facilitate systematic construction. This paper we describe a hierarchical approach to component and connector definition and construction that allows components and connectors to be defined and constructed from sub-components and connectors. These composite elements are indeed generic, compositional and reusable.

1 Introduction

A component-based system can be described as a software architecture with components (boxes) and connectors (lines). Components represent parts of the system, while connectors represent interactions between components. Connectors are therefore composition operators for the components. So, in component model, the ease of building systems and reasoning about the process depends directly on the varieties of connectors available and their semantics. A crucial question therefore is how to define and construct suitable connectors.Thus Connectors should have a semantics that makes them simple to construct and use. At the same time, their semantics should be rich enough to endow them with desirable properties such as genericity, compositionality and reusability. For connector construction, compositionality would be particularly useful, since it would allow connectors to be constructed in a systematic manner [1],[2].

In this paper we describe a hierarchical approach to connector definition and construction. Using a set of basic composition connector types, we can define and construct a composite components and a composite connectors as a composition of a primitive components and connectors. In our context these composite components and connectors are called configurations. The resulting configurations are indeed generic, compositional and reusable. Because our basic component and connector types define a control structures, our configurations represent composite control structures, or composite control flow patterns. As such, they can behave like certain design patterns, and provide powerful composition operators that can be used to perform complicated compositions involving many components all in a single step [5].

So, we take a step towards this goal by proposing a metamodel for the description of software architecture called C3 (three "C" for *Component, Connector, and Configuration*). The specificities of this metamodel are: First, proposing a new structure and new types of connectors; second, definition and manipulation of configurations as first classes entities and third, description of architectures from two different views, a model architecture view (logical architecture) created by the architect and an application architecture view (physical architecture instances of the logical architecture) generated automatically which serves as support to maintain the consistency and the evolution of the application architectures.

After this introduction, the remainder of this article is organized as follows: Section 2 provides the motivations of our research. In section 3 presents the concept of a logical architecture with the key elements of the proposed metamodel. The physical architecture is defined in section 4. The last section concludes this work with a summary of our ongoing research.

2 Motivations

Our main motivation is to propose a metamodel to maintain the consistency of an architecture using new types of connectors with a richer semantics. Using these connectors, systems are built like a Lego Blocks (*Puzzle*) by assembling components and connectors, where each element can only placed in the right place in the architecture puzzle. This metamodel will make its contribution towards the following objectives: 1-Provide a higher abstraction level for connectors in order to make them more generic and more reusable. 2-Take into account the semantics of several types of relationships. Like association, composition, and propagation relationships. 3-Promote the maintenance and the evolution of architectures by the possibility of adding, deleting and substitution of different elements in the architectural. 4-The principle of reuse should be widely exploited. New components and connectors can be defined by combining already existing elements through inheritance and/or composition mechanisms. 5-Using the physical and the logical architecture, we can separate the functional aspects of architectural elements and the non-functional aspects related to the management of their consistency.

3 Logical Architecture (LA)

Our approach is based on the description of software architecture following two architectural views. The first one is a logic view defined by the architect by assembling the compatible elements available in the library of element types and the second one is a physical view constructed automatically by the system and serves as a support for user applications built in accordance with the logical architecture. The large majority of ADLs consider components as entities of first class. So, they make distinction between component-types and component-instances. However, this is not the case with other concepts such as connectors and configurations. In our metamodel we consider each concept recognized by the *C3* metamodel as architectural element of the first class citizen. So, each architectural element maybe positioned on one of the three abstraction levels defined in the following section. We believe that it is necessary to reify the core architectural elements in order to be able to represent and manipulate them and let them evolve easily.

3.1 Abstraction levels

In our approach, software architectures are described in accordance to the first three levels of modelling defined by OMG. The application level (A_0) which represents the real word application (*an instance of the architecture*), the architecture level (A_1) which represents the architecture model and meta-architecture level (A_2) which represents the meta-language for the description of the architecture [1].

3.2 Basic concepts of C3 metamodel

3.2.1 Architectural elements

In our metamodel, an architectural element may be a component, a connector or architectural configuration. Each An architectural element may have several properties as well as constraints on these properties, as it may have one or more possible implementations. The interaction points of each architectural element with its environment are the interfaces. Each architectural element is defined by its interfaces through which they publish its required and provided services to and from its environment. Each service may use one or more ports. We approach in the following sections with more detail the most important concepts of our C3 metamodel.

3.2.2 Component

A generally accepted view of a software component is that it is a software unit with provided services and required services. The provided services are operations performed by the component. The required services are the services needed by the component to produce the provided services. The interface of a component consists of the specifications of its provided and required services. It should specify any dependencies between its provided and required services. To specify these dependencies precisely, it is necessary to mach the required services to the corresponding provided services. Services are carried using ports. Thus, we can define a generic interface of a component type as follows:

myComponent componentType (requiredInterf, provideInterf);

3.2.3 Connector

Connectors are architectural building blocks used to model the interactions between components and rules that govern these interactions. They correspond to lines in box-line descriptions. Examples are pipes, procedure call, method in-vocation, client-server protocol, and SQL link between database and application. Unlike components, connectors may not correspond to compilation entities. However, the specifications of connectors in an ADL may also contain rules to implement a specific type of connectors. From connector view point current ADLs can be classified into three different kinds:

ADLs with implicit connectors. Some ADLs, such as Darwin, Leda, and Radipe [5] do not consider connectors as first class citizens. However these ADLs make difficult the reusability of components because they have the coordination process tangled with the compotation inside them, and they are aware of the coordination process that has to happen in order to communicate with the rest. The notion of connector emerges from the need to separate the interaction from the computation in order to obtain more reusable and modularized components and to improve the level of abstraction of software architecture description. David Garlan [2] presents the need for connectors due to the fact that the specification of software systems with complex coordination protocols is very difficult without the notion of connector.

ADLs with predefined set of connectors. UniCon [5] is a typical representative of ADLs supporting a predefined set of built-in connector types only. The semantics of built-in connector types are defined as part of the language, and are intended to correspond to the usual interaction primitives supported by underlaying operating system or programming language. A connector in the UniCon language is specified by its protocol. A connector's protocol consists of the connector's type, specific set of properties, and a list of typed roles. Each role serves as a point through which the connector is connected to a component.

ADLs with explicit connector types. Most ADLs provide connectors as first order citizens of the language such as: ACME, Aesop,C2, Wright, ArchWare [2][4] [3], etc. All of these languages go a step forward with regard to the previous kind of ADLs. They improve the reusability of components and connectors by separating computation from coordination.

In our approach we opt for ADLs with explicit connector types category. So, in the C3 metamodel we present some explicit and generic types of connectors that the user can specialize following her/his needs in each application field.

3.2.4 Configuration

A configuration represents a graph of components and connectors. Configuration specifies how components are connected with connectors. This concept is needed to determine if the components are well connected, whether their interfaces agree, and so on. A configuration is described by an interface which enables its interaction.

myConfiguration configurationType (requiredInterf, provideInterf);

The following UML diagrams (Figure1) represent the main elements of C3 metamodel. For clarity raisons, these diagrams present a simplified version of our metamodel. In the rest of this article we will only deal with connectors with more detail as they represent the mainstream of our research topic in this paper.

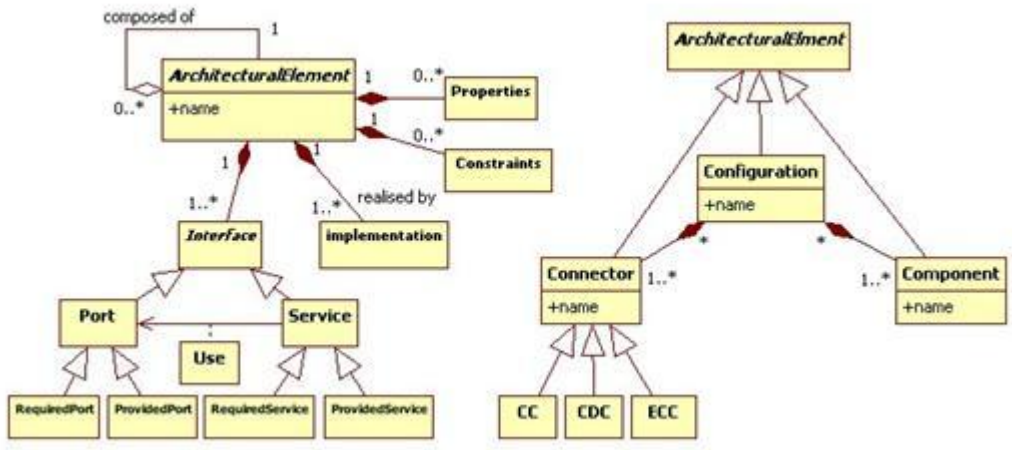

Figure 1: Architectural elements and their structure in C3

3.3 Connector in C3

A connector is mainly represented by an interface and a glue specification. Basically, the interface shows the necessary information of the connector, including the number of interaction points, service type that a connector provides (communication, conversion, coordination, facilitation), connection mode (synchronous, asynchronous), transfer mode (parallel, serial) etc[5]. In C3 interaction points of an interface are called Ports. A port is the interface of a connector intended to be tied to a component interface. A port is either a provided port or a required port. A provide port serves as entry point to a component interaction represented by a connector type instance and it is intended to be connected to the require port of a component. Similarly, a require port serves as the outlet point of a component interaction represented by a connector type instance and it is intended to be connected to the provide port of a component (or to the provide role of another connector). The number of ports within a connector denotes the degree of a connector type. For example, in client-server architecture a connector type representing procedure call interaction between client and server entities is a connector with degree two. More complex interactions among three or more components are typically represented by connector types of higher degrees.

The glue specification describes the functionality that is expected from a connector. It represents the hidden part of a connector. The glue could be just a simple protocol links ports or it could be a complex protocol that does various operations. Connectors can also have an internal architecture that includes computation and information storage.

3.3.1 Connector structure

In this work we enhance the structure of connectors by encapsulating the attachment links. So, the application builder will have to spend no effort in connecting connectors with its compatible components. Consequently, the task of the developer consists only in choosing from the library the suitable type of

connectors where its interfaces are compatible with the interfaces of component expected to be assembled.

In order to illustrate our metamodel we use cleint-server exemple as case study. In client-server configuration (CS-config) we have a client and a server. The server component itself is defined by a configuration (S-config) whose internal components are Coordinator (Coor.), securityManager (SM) and dataBase (DB). These elements are interconnected via connector services that determine the interactions that can occur between the server and client on one hand and between the server and its internal elements on the other hand.

In Figure 2 we describe the structure of the RPC connector used to connect the client component (C) with the server component (S). In this new structure the RPC connector encapsulates attachments that represent links between the client and server. Also, the same figure represents the signature specification of the connector PRC. Inside this connector type we have the glue code which describes how the activities of the client and server are coordinated. It must indicate that the activities should be sequenced in a well defined order: the customer asks for a service, the server processes the request, the server provides the result and the customer gets the result.

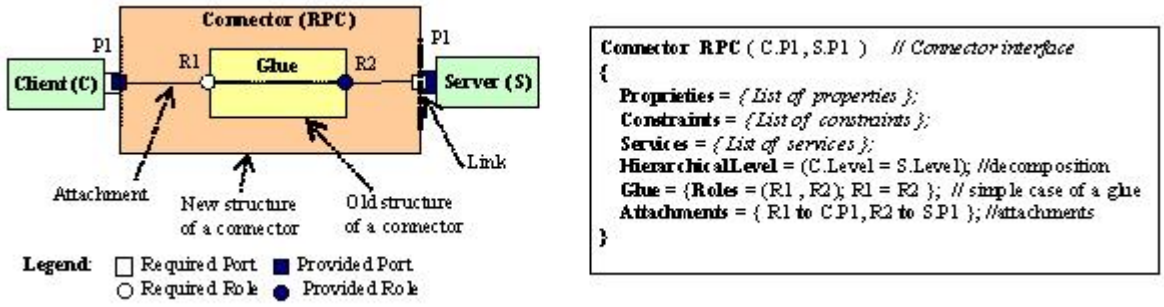

Figure 2: RPC connector structure and signature in C3

3.3.2 Connector types in C3

In C3 we have defined three connector types as illustrated in Figure 1: the *Connection Connector* type (CC), the *Composition Decomposition Connector* type (CDC), and *Expansion Compression Connector* type (ECC). Each type has its own semantic and has the following signature form:

myConnector ConnectorType (requiredInterf, providedInterf);

Where *requiredInterf* represents all required ports and services and *providedInterf* represents all provided ports and services of a connector. Obviously each interface also contains services, but in the following definitions we focus only on structural aspect of the interface (*ports*). The functional aspect (*services*) will not be addressed in this paper and therefore they will not be specified in the descriptions that follow. Consider that each service can use one or more ports of the same interface. In the following we give the exact function of each type of connector in C3 metamodel.

Connection connector (CC). This type is used to connect components belonging to the same level of decomposition. The ports of this type of connector can be *required* or *provided*. Thus, through these ports elements can exchange services between them.

myConnector CC $(\{X_i.requiredPort\}, \{Y_j.providedPort\})$

where $X_i, Y_j \in \{component, configuration\}$; $X_i.level, Y_j.level \in L_k$;

i.e. the same hierarchical level (L_k), $X_i.Level = Y_j.Level$;

$i = 1,2,..,M$; $j = 1,2,..,N$, and L_k represents the decomposition level $(k = 1,2,..,R)$

Where (M+N) is the maximum number of elements which can be linked by CC connector. Hence, CC may have to (M+N) ports. The mapping between the inputs and outputs is described by an exchange protocol called glue defined inside of the connector. Figure 3 represents CC1 a connection connector type used to link a client component with s-config configuration of the previous example. This type connector has two ports: portC1 in client side and portS1 in server side. Hence, the interface CC1 will be defined as follows: *CC1 CC(portC1, portS1);*

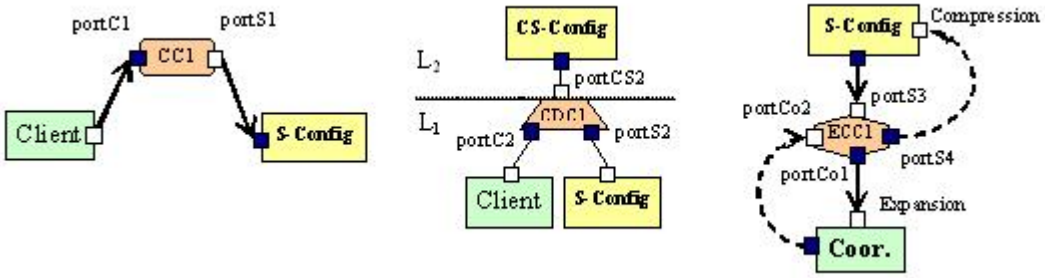

Figure 3: CC, CDC and ECC connector types in client-server architecture

Decomposition/composition connector (CDC). CDC connector type is used to realize a top-down refinement (i.e. to link a configuration with its internal elements) also we call this relationship a decomposition model. Likewise CDC connector can be used to realize bottom-up abstraction (i.e. to link a set of elements to their container or configuration also we call this relationship a composition model. However, this type of connectors can play two semantic roles with two different glue protocols.

myConnector CDC $(X.requiredPort, \{Y_i.providedPort\})$; decomposition of X to its internals;

myConnector CDC$(\{Y_i.requiredPort\}, X.providedPort)$; composition of Y_i elements to constitute X;

Where X is a configuration, $Y_i \in \{component, configuration\}$, *and*

$i = 1,2,..,N; X \in L_k$ and $Y_i \in L_{k-j}$ *(i.e.* $X.Level > Y_i.Level$*) L is the hierarchical level.*

Thus, a CDC connector will have (N+1) ports, where N is the number of internal elements in the corresponding configuration. This type of connector has the following interests: first it allows us to shape the genealogical tree of the different elements deployed in an architecture, second it enables a configuration to spread information to all these internal elements without exception (to-down propagation) and inversely; i.e. it allows any internal element to send information to its configuration.

Figure 3 represents CDC1 a decomposition composition connector type used to link client-server configuration (CS-config) defined at the hierarchical level (L_2) with its internals namely client component (Client) and server configuration (s-config) defined at the lower hierarchical level (L_1). Consequently, the interface of CDC1 connector type will be specified as follows: *CDC1 CDC(portCS,portC2,portS2);*

Where portC2, portS2, and portCS are respectively used to connect CDC1 with the client component, the server configuration, and client-server configuration (CS-config).

Expansion / compression connector (CDC). The ECC is used to establish a service link between a configuration and its internal elements. Also, ECC can be used as an expansion operator of services to several sub-services and it can be used in reverse as a compression operator of set of services to a global service. The CDC may have an interface for expansion and another for compression. So, these interfaces are defined as follows:

myConnector ECC $(X.requiredPort, \{Y_i.providedPort\})$; //expansion

myConnector ECC $(\{Y_i.requiredPort\}, X.providedPort)$; //compression
$i = 1,2,..,N$ *where* $N \leq$ *number of internal elements.*
$X \in L_k$ et $Yi \in L_{k-1}$; $(i.e.\ X.Level > Y_i.Level)$ *L is the hierarchical level.*

ECC connector type can be implemented using either single glue for one function (expansion or compression) or using two separate glues for expansion and compression functions. This will depend on the design decision.

Figure 3 illustrates the connector type ECC1 which allows exchange of information between the server configuration (s-config) and the coordinator component (Coor.). Thus, to achieve a bidirectional communication between the server and coordinator, ECC1 must have the following ports: portS3 and portCo1 are used to ensure the expansion function from the server to coordinator. The portCo2 and portS4 are used to ensure compression function. The interface of this ECC1 type will be as follows: *ECC1 ECC(portS3, portCo1, portS4, portCo2);*

4 Physical Architecture (PA)

The physical architecture is a memory image of the application instance of the logical architecture. This image is built in the form of a graph whose nodes are instances of a connections manager. Each instance created corresponds to a component or a configuration instanced to construct the real application. Nodes of this graph are connected by arcs. We have three types of arcs. Each type of arc corresponds to specific type of connector. The physical architecture is built to serve as support for updating and evolution operations of the application instance.

4.1 Connections Manager (CM)

The PA is described using only two levels of abstractions; type level and instance level. In the type level we have the CM type represented by a class that encapsulates all information about links that a component or a configuration may have with its environment.Each CM is identified by a name and the following attributes: *ElementName*: represents the name of the architectural element associated with this CM (i.e. the name of the component or the configuration corresponding); *CC-Links*: list of connection connector names connected to the element associated with this CM; *CDC-link*: the name of the composition decomposition connector connected to the element associated with this CM; *ECC-Link*: the name of the expansion compression connector connected to the element associated with this CM;

4.2 Operations on Connections Manager

The possible operations on the connections manager are: *Instantiation*: the connection manager is instantiated at the instance level (A_0) of the physical architecture. Whenever an architectural element is instantiated at the application level the associated CM is automatically created in the physical architecture; *Installation*: each time a connector is installed at the application level between a set of element instances, so the attributes of the associated CMs are updated with the necessary information about this connector instance. *Propagation*: the mechanism of propagation is used to update information about links needed between CMs. These links are published by the interface of the connector installed at the application level. Once the application is built by the user, the corresponding physical architecture is also built in parallel. Thereafter if we need to intervene on the application to maintain or evolve it we must locate the concerned elements on the physical architecture using graph searching routines and graph updating operations like add (node), delete (node) or replace (node).

Finally we can represent the logical architecture and the physical architecture and the relationship between them by an architecture model described in C3 metamodel where the logical architecture and

the physical one are represented by two components and the relationship between the by a connection connector. Any action performed at the logical architecture causes a sending a message from first architecture type to the second architecture type. This message will interpreted as an action to be performed by the physical architecture.

Exchanged services (*operations*) between the types of architectures are: A component instantiation at the logical architecture level causes sending a message *CM-creation* from *LA-Interface* to *PA-Interface*. When this message is received by the physical architecture a connection manager instance will be created to represent this component at the physical architecture level.

A connector instantiation at the logical architecture level causes sending a message *CM-connection* from *LA-Interface* to *PA-Interface*. When this message is received by the physical architecture a set links are created to link connection manager instances corresponding to all components connected by this connector instance. Any updating action (*replacement or deleting of a component or a connector*) at the logical architecture causes sending a message *CM-update* from LAinterface to PAinterface. When this message is received by the physical architecture a set of updating operations are performed to rearrange links among the corresponding CMs.

5 Conclusion

In this article we have presented the core elements of C3 metamodel and how to describe software architecture using C3. The elements defined by C3 are assembled through their interfaces to build software architectures. So, we must ensure syntactic checks by checking the compatibility of interfaces. We found interesting to give a new structure for connectors in which attachments are encapsulated within the definition of connectors. Hence, the interface connector is now a set of services and ports. This new structure allows us to assemble connectors only with elements that are defined in its interface. We have identified three types of connectors. CC which refer to the links among components belonging to the same level of decomposition. CDC which refer to the links between a configuration and its internal components and connectors. ECC which refer to the links used to realize any transformation of information or data exchanged between a configuration and its internal components.

Also, we have defined a PA as a graph whose nodes are CMs associated with architectural elements and arcs represent links that correspond to the connectors. The PA reflects the application architecture which is an instance of the logical architecture and serves as a support for maintenance and evolution operations applied on architecture of the application. As extension for this work, we planned to define more than one hierarchical view to describe component-based architectures, relationship between these hierarchies, and the different connection mechanisms used to enable interactions between elements from different hierarchy views.

References

[1] Szyperski C. Component software: Beyond object-oriented programming. Addison-Wesley, 2002.

[2] Garlan D., Monroe, R.T., and Wile D. Acme: Architectural description component-based systems, foundations of component-based systems. Cambridge University Press, 2000.

[3] Dashofy E., Hoek A.v.d., and Taylor R.N. A comprehensive approach for the development of xml-based software architecture description languages. Transactions on Software Engineering Methodology, 2005.

[4] Medvidovic N., Dashofy E., and Taylor R.N. Moving architectural description from under the technology lamppost. Information and Software Technology, 2007.

[5] Medvidovic N. and Taylor R.N. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 2000.