Enhanced Connectors to Support Hierarchical Dependencies in Software Architecture

Abdelkrim Amirat, Mourad Oussalah

Enhanced Connectors to Support Hierarchical Dependencies in Software Architecture

Abdelkrim Amirat
LINA CNRS UMR 6241, Université de Nantes
2, Rue de la Houssinière, BP 92208,
44322 Nantes Cedex 03, France
Phone: +33.2.51.12.59.68
abdelkrim.amirat@univ-nantes.fr

Mourad Oussalah
LINA CNRS UMR 6241, Université de Nantes
2, Rue de la Houssinière, BP 92208,
44322 Nantes Cedex 03, France
Phone: +33.2.51.12.58.47
mourad.oussalah@univ-nantes.fr

ABSTRACT
The more important level of abstraction in the description of large and complex software is its architecture description. So, at this abstraction level we can describe the principal system components and their pathways of interaction. Software architecture is considered to be the driving aspect of the development process; it allows specifying which aspects and models in each level needed according to the software architecture design. Early Architecture Description Languages (ADLs), nearly exclusive, focus on structural abstraction hierarchy ignoring behavioural description hierarchy, conceptual hierarchy, and metamodeling hierarchy. In this paper we show that all those hierarchies constitute views to appropriately "reason about" software architectures described using our C3 metamodel which is a minimal and complete ADL. We provide a set of mechanisms to deal with different levels of each hierarchy; also we introduce a new enhanced definition for connector concept deployed in C3 architectures.

Categories and Subject Descriptors
D.2.11 **[Software Architectures]**: Data abstraction, languages description, interconnection, and definition.

General Terms
Design, Reliability, Languages.

Keywords
Metamodeling, Abstraction, Hierarchy, Connector, Component, Architecture.

1. INTRODUCTION
Nowadays, there is a completely new approach to building more reliable software systems which consist to decompose large and complex systems into smaller and well-defined units – software components.

NOTERE 2008, June 23-27, 2008, Lyon, France.

Typically, components are considered to be entities with well-defined provided (server) and required (client) interfaces, and in some cases also with formally specified behaviour. A component-based application is a collection of individual components, which are interconnected via well-defined connectors between their interfaces.

Component that have no externally observable internal structure, while having real implementation in certain programming language, are called *primitive components*. Components containing nested subcomponents, i.e. components with observable internal structure, are called *configurations (composite components)*. The structure of a configuration, commonly referred to as *configuration architecture*, is typically defined in an Architecture Description Language (ADL) [1].

Generally, software architectures are composed of components, connectors and configurations, constraints on the arrangement and behaviour of components and connectors. The architecture of a software system is a model, or abstraction of that system. Software architecture researchers need extensible, flexible architecture descriptions languages (ADLs) and equally clear and flexible mechanisms to manipulate these core elements at the architecture level.

There is not today, nor has there ever been, a clear consensus on a definition of software architecture. Recently Medvidovic [7] gives the following definition for software architecture "*A software system's architecture is the set of design decisions about the system"*. So, if those decisions are made incorrectly, they may cause your project to be cancelled. However, these design decisions encompass every aspect of the system under development, including:

- Design decisions related to system *structure* – for example, "there should be exactly three components in the system, the 'data store', the 'business logic' and the 'user interface component';"
- Design decisions related to behaviour also referred to as functional – for example, "data processing, storage, and visualisation will be handled separately;"
- Design decisions related to the system's non functional properties – for example, "the system dependability will be ensured by replicated processing modules;"
- Also, we can elicit other design decisions related to the development process or the business position (*product-line*).

The "first-generation" ADLs all shared certain traits. They all modeled the structural and, with the exception of Acme [4], functional characteristics of software systems. They invariably took a single, limited perspective on software architecture. In this paper we introduce further perspectives which are complementary to the structural one.

The majority of ADLs proposes, like reasoning model, only sub-typing (inheritance) as a mechanism for specialization (e.g. Acme [4], C2 [16]). Otherwise, for the rest of ADLs, they propose their own ad hoc mechanisms based on algorithms and methods designed specially for these ADLs.

Based on a broad survey of architecture description notations and approaches, we identified that ADLs capture aspects of software design centred around a system's *Component*, *connectors*, and *configurations*. The core elements of our model are basically defined around these three elements. So, we derive the name of our model **C3** from **C**omponent, **C**onnector, and **C**onfiguration. Taking into consideration that our C3 have no relationship with C2 defined by Taylor [16] nor with C3 which is an extension of C2 defined by Pérez-Martínez [12]. The rest of the paper is organized as follows. In section 2 presents our research motivations. Section 3 describes the C3 metamodel. In section 4 we use client-server architecture as an example to apply the presented approach. The last section presents our conclusion and the different perspectives of our work.

2. MOTIVATION

Our motivation in this work is to develop a generic model for the description of software architectures which must be minimal and complete. It is minimal because we are only interested by the core concepts in each ADL. And complete because with this minimum of concepts the architect will be able to describe any required structures he need to realize using those concepts and a set of predefined mechanisms.

However, describing only the architecture structure is not sufficient to provide correct and reliable software systems. In this paper we are even more going to focus on representation architecture model and to reason about its elements following four different types of hierarchies. Each of these hierarchies provides a particular view on the architecture. In the following sections we present more details about these hierarchies.

We expect from our approach to provide more explicit and better clarified software architecture. Mainly the approach is developed to:

- Make explicit the possible types of hierarchies being used as support of reasoning on the architecture structures, with the different possible levels in each hierarchy.
- Show semantics conveyed by every type of hierarchy by providing the necessary mechanisms used to connect elements in the same level of hierarchy and the mechanisms used to connect elements of every level with the elements of the upper level and the level below.
- Allows introducing various mechanisms of reasoning within the same architecture according to the requirements of the system in a specific application domain.
- Establish the position of existing mechanisms developed for reasoning with regard to our referential.

3. The Meta Model of C3

In order to have a complete C3 model, we have defined mainly two complementary models to describe and reason about system's architecture. We use *representation model* to describe architectures based on C3 elements and we use *reasoning model* to understand and analyse the representation model.

3.1 Representation Model

The core elements of the C3 representation model are components, connectors, and configurations, each of these elements have an interface to interact with its environment like depicted in Figure 1.

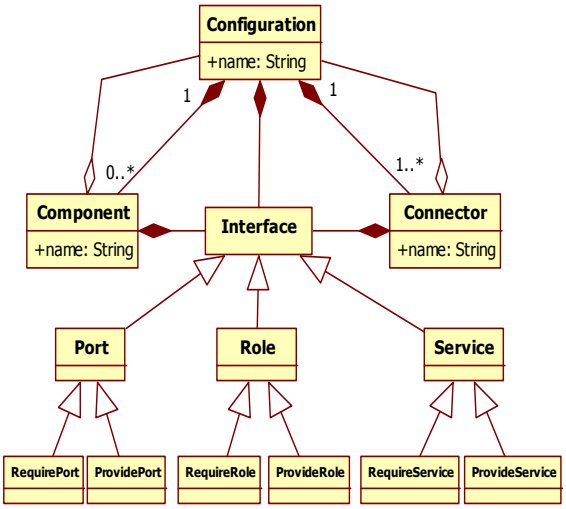


Figure 1. Basic elements of C3 Meta model

3.1.1 Component

A component is a unit of computation or data storage. Therefore, components are loci of computation and state. A component may be as small as a single procedure or as entire application. It may require its own data or execution space, or it may share them with other components [8].

In order to be able to adequately reason about a component and the architecture that includes it, C3 should also provides facilities for specifying components needs, i.e, services required of other components in the architecture. An interface thus defines computational commitments a component can make and constraints on its usage.

Each interaction point of a component is called a *port*. Ports are named and typed. We distinguish between required and provided ports. Each port can be used by one or more services.

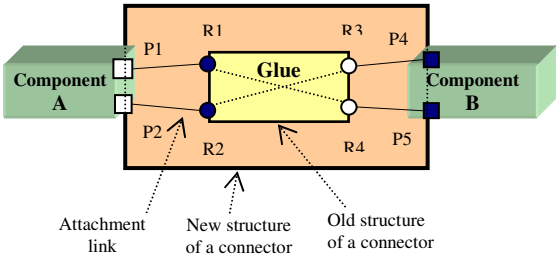
Component semantics are modeled to enable evolution, analysis, enforcement of constraints, and consistent mappings of architectures from one level of abstraction to another. The structure of component is the specification of its required and provided ports. The behaviour of a component is the specification of its required and provided services exchanged with its environment.

3.1.2 Connector

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. In order to enable proper connectivity of components and their communication, a connector should export as its interface those services it exports [8].

C3 refers to connector interaction points as *roles*. Explicit connection (attachments) of component ports and connector roles is required in an architecture configuration. Roles are named and typed and are in many ways similar to component ports. Connector services are described inside the glue code [15]. Therefore, a connector's interface is a set of interaction points between it and the components/configurations attached to it. It enables reasoning about well-formedness of an architectural configuration.

Our contribution at this level consists in enhancing the structure of connectors by encapsulating the attachment links inside the definition of connector types (Figure 2.a). So, the application builder will have to spend no effort in connecting connectors with its compatible components and configuration. Consequently, the task of the developer consist only in choosing the suitable type of connector which is compatible with the interface types of components and/or configurations which are expected to be connected [2].



Legend: P_i : Port i ; R_j : Role j

Figure 2.a. The new structure of a connector

We have given the following signature definition for connectors (Figure 2.b).

Connector_TypeName (*List of element interfaces*)
{
Roles {*List of roles*}
Services {*List of services*}
Properties {*List of properties*}
Constraints {*List of constraints*}
Glue {*The communication protocol*}
Attachments {*List of attachments*}
}

Figure 2.b. Signature of C3 connector

So, by encapsulating attachments inside connectors and having well defined connector interfaces with previously known elements to be connected by each connector, consequently components and configurations assembled in an easy and coherent way in the form of *Lego Blocks.*

3.1.3 Configuration

Architecture configurations or topologies are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether appropriate components are connected, their interfaces mach, connectors enables proper communication, and their combined semantics result in desired behaviour.

The goal of configuration is to abstract away the details of individual components and connectors and to describe how they are fastened to each other. They depict the system at a high level that can be potentially be understood by people with various levels of technical expertise and familiarity with the problem at hand.

For more clarity, in C3 model each component or connector is perceived and handled from the outside as primitive element. But their inside can be real primitive elements, or composite with a configuration which encapsulates all the internal elements of this composite. These Configurations are first-class entities. A configuration may have ports similar to component ports, and each port is perceived like a bridge (*binding*) between the internal environment of the configuration and the external one. In C3 this binding is realised using connectors. Generally configurations can be hierarchical where the internal components and connectors can represent sub-configurations with their proper internal architectures and so on as depicted in Figure 1.

3.1.4 Interface

Every architectural element has an interface. Each interface is associated with a type which corresponds to a set of operations which it defines. Via this interface the element publishes to the outside environment it needs in term of required services as well as the services which it provides. However, elements are selected and connected from their published interface. So, the interface is thought as a *contract* with the environment that the element should *honour*.

To establish connections between elements we use required/provided ports for component and configuration elements and required/provided roles for connector elements and we assign the services to each port and role with the necessary set of constraints to be respected during the connections. From conceptual view ports, roles and services are concrete classes inherited from the interface abstract class as shown in Figure 1.

Also, at modelling level we use cardinality to describe the multiplicity of each relation (*connection*) between architectural elements. This cardinality express the number of ports associated with components and configurations and the number of roles associated with connectors. Each port or role is considered as a channel to carry in/out required/provided services exchanged with elements of the environment.

In our approach, components and connectors are assembled in an easy and coherent way in the form of an architectural puzzle (*Lego Blocks*) without any effort to describe links among components and connectors or among configurations and connectors because attachments are predefined in each type of connector, so can only connect two components by their

compatible connector. Consequently, this approach accelerates the development of components, improves testability, coherence, maintainability and promotes component markets More details about the structure of these architectural elements are presented in previous works [2], [11].

The previous architectural elements are manipulated and used via predefined mechanisms in the reasoning model. Essentially, we are going to study the instantiation, specialization, composition, decomposition, and connections mechanisms. In the following section, we define the using context of each mechanism.

3.2 Reasoning Model

In our approach we plan to analyze the software architecture by using different hierarchy views where each hierarchy is investigated at different levels of representations. Figure 3 illustrates the C3 reasoning model. This model is defined by four types of hierarchies and each type represents a specific view on the C3 representation model different from the others.

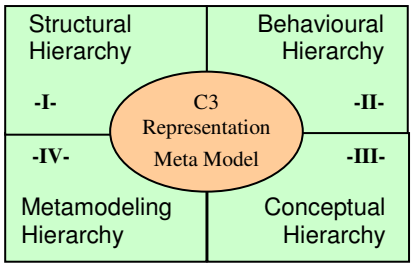


Figure 3. C3 Representation and reasoning models

The four hierarchies are: 1- The structural hierarchy used to explicit the different nested levels of structural hierarchies that the system's architecture can have. 2- The behavioural description hierarchy to explicit the different levels of system's behaviour hierarchy generally represented by protocols. 3- The conceptual hierarchy to describe the libraries of element types corresponding to structural or behavioural element at each level of the architecture description. 4- The metamodeling hierarchy to locate where our model coming from and what we can do with it. Obviously those two sides will belong to the pyramid of hierarchies defined by Object Management Group [9], [10].

We associate to each hierarchy two points of views. The first one is an external view "*the logical architecture*" as it is perceived by the user (*designer or developer*) of the architecture. The second kind of view is internal view "*the physical architecture*" which represents the memory image of the logical architecture. Some details about the logical and physical architecture are presented in [11]. In the following sections, we present those types of hierarchy and we investigate the possible levels of each hierarchy with the associated mechanisms.

3.2.1 Structural Hierarchy (SH)

Structural hierarchy also called abstraction hierarchy has to provide the structure of particular system architecture in terms of the architectural elements defined by the used ADL. The majority of academic ADLs like Aesop, MetaH, Rapide, SADL, and other [6] or the industrials like CORBA, CCM/CORBA, EJB/J2EE [13] allow only a flat description of software architectures.

Using those ADLs architecture is described only in terms of components connected by connectors without any nested elements without any structural hierarchy. This design choice was made in order to simplify the structure and also by lack of concepts and mechanisms that respectively define and manipulate configurations of components and connectors.

In our C3 model the structure of architectures is described using components, connectors, and configurations, where configurations are composite elements. Each element in this configuration (component or connector) can be a primitive (with a basic behaviour scenario) or a new configuration which contains another set of components and connectors, which in their turn can be primitive or composite material, and so on.

However, the metamodel C3 allows the representation of architecture with a real hierarchy with a number of abstraction levels (L_n, L_{n-1}, L_{n-2},…, L_1, L_0) depending on the complexity of the problem. It is important to note that practically all architectural solutions for domain problems have a nested hierarchical nature.

Thus, real software architecture can be viewed as a graph where each internal node of this graph represents a configuration and each end-node represents a primitive component and arcs between nodes are connectors.

In Figure 4.a, the root node with double circles represents is the first level of the abstraction; it is also the global configuration which encapsulates all elements of the architecture. The small white circles represent primitive components and small black circles represent sub-configurations in the system architecture. These configurations contain other elements inside. Thus, a configuration will never be an end-node in the hierarchy tree of abstractions. Arcs represent the bonds of levels - the father/child relationship. This relationship does not necessarily imply a service-connection between the father node and the child one.

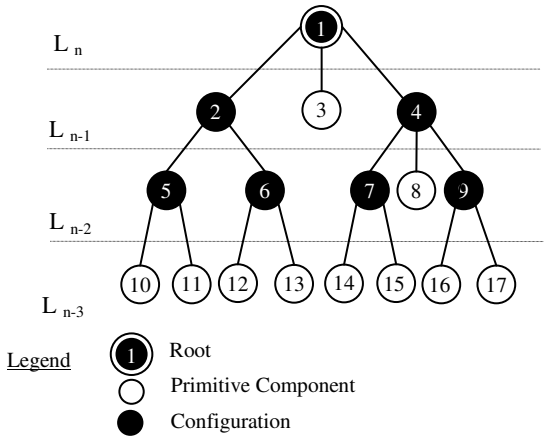


Figure 4.a. External view of structural hierarchy

To navigate among structural hierarchy levels we define the following type of connector:

3.2.1.1 Composition-Decomposition Connector (CDC)

This type of connectors is used to link each configuration to its underling elements. Therefore, this type of connector allows the navigation among levels of the structural hierarchy. So, we can determine the childs or the father, if it is the case, of each elements deployed in the architecture. Figure 4.b illustrates how to use CDC connector to represent abstraction details (nodes 7, 8, and 9) at Level (n-2) corresponding to node (4) at Level (n-1). So, the internal view of Figure 4.a is described by seven CDC connectors.

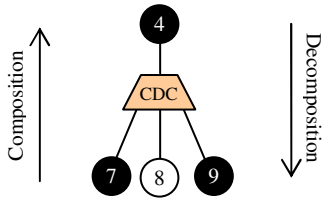


Figure 4.b. CDC connector

Figures 5.a illustrates two service-connection types of connector; the fist one is generally represented by an implicit link called *Binding* and the second one is defined by several ADLs as *Attachment* link. In our model those two types of connections are explicit and first class entities. They are defined as follows:

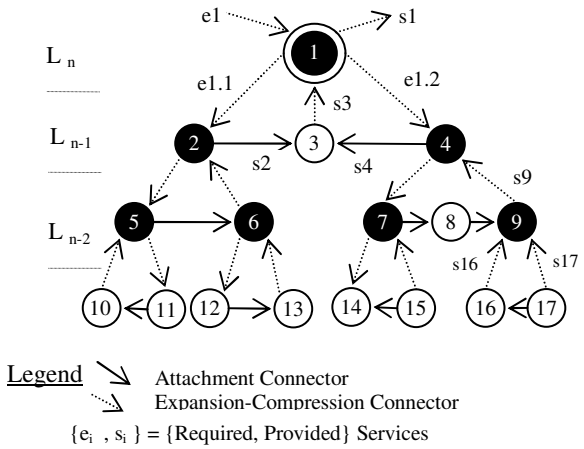


Figure 5.a. Internal view of structural hierarchy

3.2.1.2 Attachment Connector (AC)

Attachment connector is represented by a solid-line arc in Figure 5.a. We use this type of connectors to establish service-connections between components and/or configurations which are deployed in the same level of abstraction. More details about AC connector are depicted in Figure 5.b. The example described in Figure 5.b is independent from the one described in Figure 5.a. In some ADLs this type of connector in called *assembling connector* and represented by first class entity (e.g. Acme [4]).

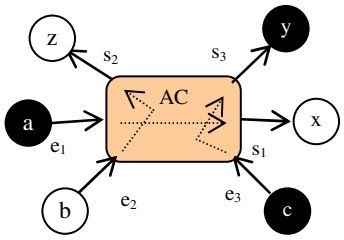


Figure 5.b. AC connector

Inside the AC connector the glue code (Figure 5.b) defines the following mapping among communicating elements:

- The provided service of "a" is required by "x" (e_1=s_1),
- The provided service of "b" is required by "z" (e_2=s_2),
- The provided service of "c" is required by "y" (e_3=s_3).

Roles are not indicated in the figure for space reasons. Attachment links among elements {a,b,c,x,y,z} inside the AC connector are represented by a dotted-line oriented arcs.

It is important to note that in structural hierarchy we use at each level a different set of mechanisms and tools to deal with the input interfaces and the output interfaces. For this reason inputs are generally expanded when we shift from Level (i) to Level (i-1) and outputs are compressed when we shift from $level_{i-1}$ to level L_i. So, the data format will change when we change the level. (e.g. if we manipulate a graph as a data in level L_i then our inputs and outputs are graphs and we have a specific set of tools to manipulate them; at a lower level of the abstraction we have an other format of data to represent graphs, so it is normal that tools used at this last level are not those used in the upper level of the abstraction. From this observation we will define, in the following, a connector for expansion of inputs and compression of outputs.

3.2.1.3 Expansion-Compression Connector (ECC)

ECC type of connectors represented by a dotted-line oriented arc in Figure 5.a is represented in more details in Figure 5.c. We use this type of connectors to establish service-connections between each configuration and its underling elements (Figure 5.c). In some ADLs this type of link is called *binding (e.g. Acme) or delegation (e.g. UML)* but it is not defined as a first class entity.

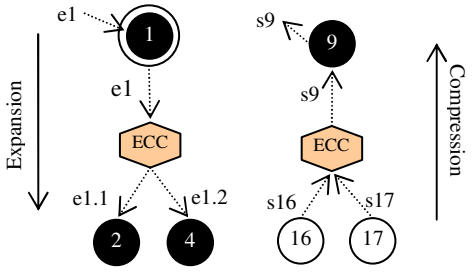
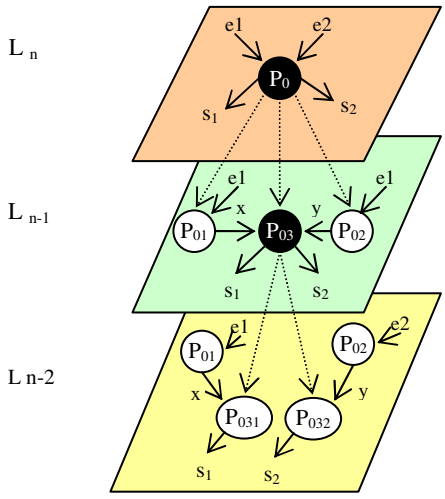


Figure 5.c. ECC connector

In Figure 5.c we use the ECC connector to do the expansion of the inputs or to do the compression of the outputs. The different elements of the architecture are connected through their interfaces. Thus the types of interfaces are checked if they are compatible or not (*interface matching*). Consequently, in the structural hierarchy, the consistency of the assembled elements is controlled syntactically.

3.2.2 Behavioural Hierarchy (BH)

The behavioural hierarchy represent the description of the system's behaviour at different levels. Each primitive element of the architecture has its own behaviour. The behaviour description associated with the highest level of the hierarchy - L_n - in Figure 6.a represents the overall behaviour of the system. This behaviour is described by a global protocol P_0. The system architecture at this level is perceived as a black box with a set of inputs (*required services*) and outputs (*provided services*). At lower level each component, connector, configuration, port, or role has its own protocol to describe its functionality (e.g. glue code is the protocol describing the connector behaviour), also the element behaviour can be described by a statechart diagram or Petri nets [17]. So, a protocol is a mechanism used to specify the behaviour function of an architectural element by defining the relationship among the possible states of this element and its ability to produce coherent results. Figure 6.a sketches how to decompose the protocol P_0 at level L_n into its sub-protocols at Level (n-1). This decomposition process produces a set of other protocols {P_{01}, P_{02}, and P_{03}}. By the same process each protocol of the level (n-1) is decomposed to produce an other set of sub-protocols at the level L_{n-2}, and so on until level L_0. The last level of the hierarchy is a set of protocols representing the primitive behaviour of elements which are available in the library of the architect. The total set of protocol levels represents the behaviour hierarchy of the system architecture.



Legend: P_i : Protocol i ; e_i : Input i ; s_i : Output i ; x , y : intermediate results

Figure 6.a. Decomposition of behavioural hierarchy

To explicit the parental relationship among elements belonging to successive levels of the behavioural hierarchy we define the following types of connector:

3.2.2.1 Composition-Decomposition Connector (CDC)

CDC connector is used to link each protocol to its possible sub-protocols. Therefore, this type of connector allows the navigation among levels of the behavioural hierarchy. Also, we can determine the childs or the father, if it is the case, of each protocol used in the architecture. Figure 6.b represents the notation adopted.

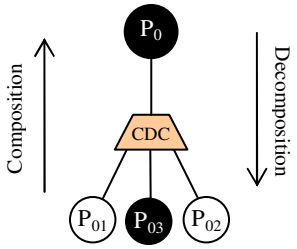


Figure 6.b. CDC connector

3.2.2.2 Attachment connector (AC)

In behavioural hierarchy, attachment connectors are used to connect protocols belonging to the same level of hierarchy. If we use, for example, a transition-based system to specify the behaviour protocol associated with each element then connections between behaviours are made by simple transitions between the end-state of the fist protocol and the start-state of the second one. Inputs and outputs of each protocol are respectively required and provided services (Figure 6.d).

3.2.2.3 Binding Identity Connector (BIC)

Binding identity connector is used to keep the identity and the traceability of inputs and outputs of protocols. In contrast to the structural hierarchy, in the behavioural hierarchy we have neither expansion of inputs nor compression of outputs when we change the level of hierarchy. Consequently, we use the same set of mechanisms and tools at all hierarchical levels. So, when we move to the below level this in fact amounts to just slicing (*sampling*) the behaviour function to several sub-functions in order to understand and analyze the global complex behaviour of the system (Figure 6.c).

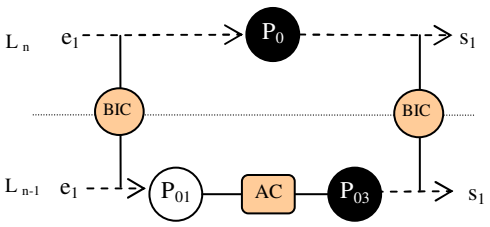


Figure 6.c. BIC and AC connectors

The syntactic correction (discussed before in SH) of the assembled elements cannot insure the validation of the produced architecture. The syntactic correction checks only the compatibility of interfaces types. So, elements are compatible to exchange information, but fail to check if their collaboration "*the semantic of connections*" can produce a coherent result. Consequently, during the development of behavioural hierarchy we will insure the compatibility of protocols (*protocol matching*) associated with connected elements at any level of this hierarchy [5] [17].

It is important to note that there is no priority relationship between structural and behaviour hierarchies. So, the designer can start describing the architecture using the structural or the behaviour hierarchy. It depends on the type of information he/she has at first. But generally if the structural hierarchy is known it will be suitable to start by the structural hierarchy and at each level of this hierarchy we develop the corresponding behavioural hierarchy.

3.2.3 Conceptual Hierarchy (CH)

The conceptual hierarchy allows the architect to model the relationship among elements of the same family as illustrated in Figure 7.a for the component types. The architectural entities are represented by types.

Each type is an element in the library and each element has its sub-elements in the same library. So, we can shape the graph representing entities hierarchy of the same family. Each graph has its proper number of levels (sub-type levels). At the highest level of hierarchy we have the basic element types developed to be reused. The element types of the intermediate levels are created by reusing the previous ones.

Those intermediate element types are reused to produce others (elements are *developed by reuse and to be reused*) or used as end-elements to describe architectures, and so on. Element types at the last level of the hierarchy are only created to be used in the description of architectures [3].

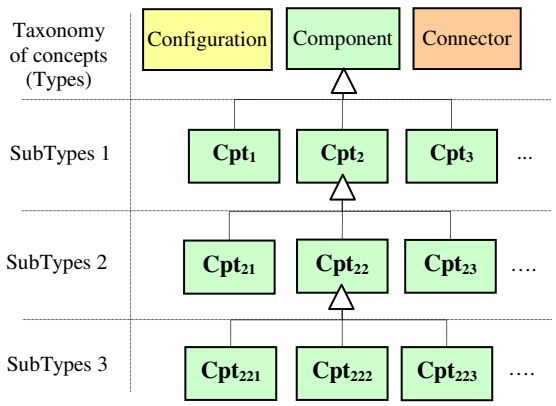


Figure 7.a. Conceptual hierarchy

Through the mechanism of specialization (e.g. *inheritance*) the architect will create and classify element libraries according to architecture development needs in each target domain. The number of sub-type levels is unlimited. But we must remain at reasonable levels of specialization in order to keep compromise between the use and the reuse of the architectural elements. To implement the conceptual hierarchy we define the following type of connector:

3.2.3.1 Specialisation-Generalisation Connector (SGC)

This type of connectors is used to connect element types coming from the same type (e.g. this connector is implemented by the inheritance mechanism in Java). So, we can construct easily all trees representing the classification library types. Figure 7.b represents the notation adopted.

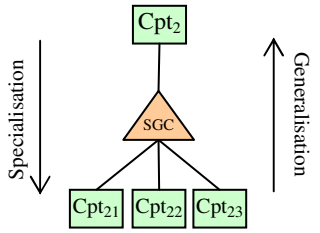


Figure 7.b. SGC connector

3.2.4 Metamodeling Hierarchy (MH)

The metamodeling hierarchy is viewed like pyramid composed exactly from 4 architectural levels endowed with an instantiation mechanism (*instance-of relationship*). Thus, according to Figure 8.a, each level (A_i) must conforms to the description given above in $A_{(i+1)}$ level. The level A_3 conforms to itself. Symmetrically, each level (A_i) describes the inferior level $A_{(i-1)}$. A_0 is the end-level (*application instance*) [9], [10].

A_0 Level is the real word level (*application level*) which is an instance of the architecture model (level A_1). At this level the developer has the possibility to select and instantiate elements any times as he needs to describe a complete application. Instances are created from element types which are defined at A_1. Elements are created and assembled with respect to the different constraints defined at A_1 level.

A_1 Level is also called *architecture level*. At this level we have models of architecture described using language constructions or notations defined at A_2 level (e.g. C3 metamodel, UML 2.0). Thus, each architecture model is an instance of the metamodel defined in the above level.

A_2 Level (*meta-architecture level*) defines the language or the notation used to describe architectures at A_1 level. This level is also used to modify or adapt the description language. All operations undertaken at this level are always in conformance with the top level of the pyramid.

A_3 Level (*meta meta-architecture*) has the top level concepts and elements used when we want to define any new architecture description language or new notation. In our previous work we have defined our proper meta meta-architecture model called Meta Architecture Description Language (MADL) [14]. So, our C3 metamodel is defined in conformance with MADL. MADL is similar to MOF but it is a component-oriented.

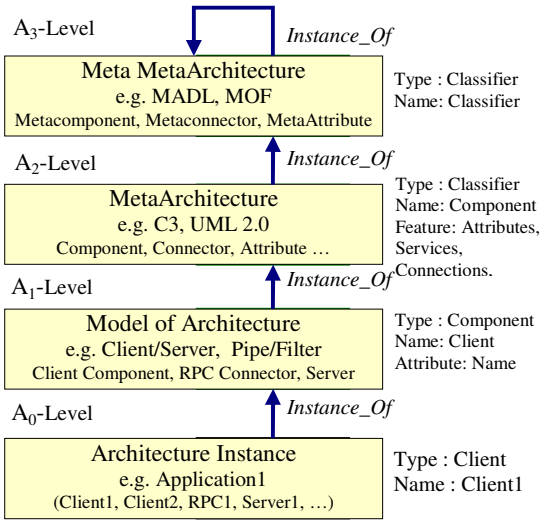


Figure 8.a. Metamodeling hierarchy

To connect each architectural element to its type at the above level we define:

3.2.4.1 Instance-Of Connector (IOC)

Instance-of connectors are used to establish connection among elements of a given level (*model*) with their classifier defined in the above level (*metamodel*). Figure 8.b represents an example of the notation adopted.

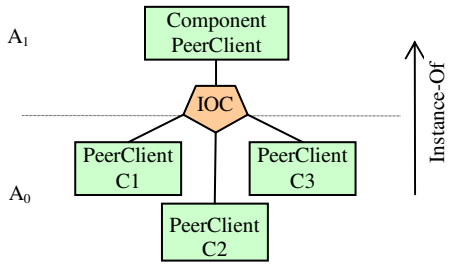


Figure 8.b. IOC connector

4. CASE STUDY

As a simple illustrative example, Figure 9 depicts the client-server architecture. In the following subsections we tray to analyse this example from the provided view of each type of hierarchy introduced in this paper. In the next figures we use number notations to represent the following elements:

1- Client-Server Architecture,
2- Client Component,
3- Server Configuration,
4- Connection Manager Component,
5- Security Manager Component,
6- Data Bases Component.

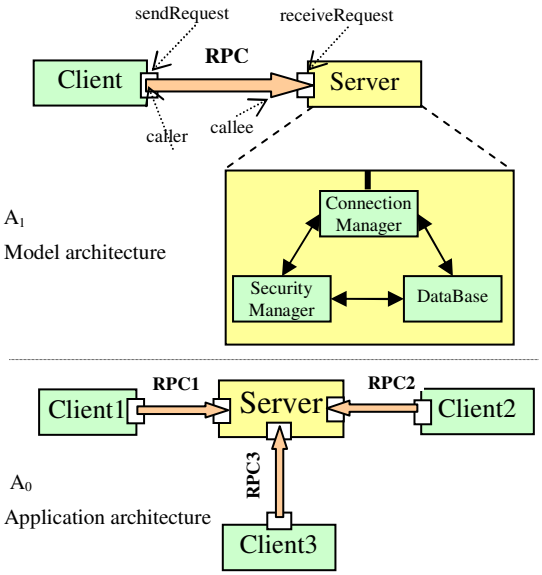


Figure 9. Simple client-server architecture / application

4.1 Structural Hierarchy

The structural hierarchy corresponding to the client-server example can be represented by three levels. In Figure 10.a we represent this hierarchy by means of two graphs. Each graph gives particular view for the same set of nodes (six nodes). There is one node for each element of system. The left one illustrates the composition-decomposition view for the structural hierarchy using two CDC connectors, and the right one illustrates the same hierarchy but from physical structure interconnections view using two ECC connectors and four AC connectors. So, at this point we can say that all structural elements of the client-server system are depicted by the structural hierarchy.

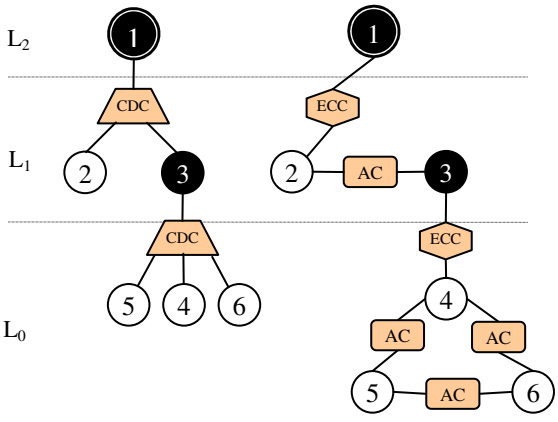


Figure 10.a. Structural hierarchy

In Figure 10.b we give, in an illustrative example, some details about AC connector representing the RPC connector used to connect the client component (node 2) to the server configuration (node 3).

Component Client {Port {send-request}}
Configuration Server {Port {receive-request}}
Connector RPC: AC (Client, Server)
{
Roles {Caller, Callee}
Services {List of services}
Properties {maxRoles: integer = 2,
Synchronous: Boolean = true}
Constraints {List of constraints}
Glue {caller = callee}
Attachments {Client.send-request to RPC.caller
Server.receive-request to RPC.callee}
}

Figure 10.b. Description of RPC Connector

4.2 Behavioural Hierarchy

To simplify our representation for the behavioural hierarchy, we describe three graphs. Each graph illustrates the behavioural hierarchy using a particular type of connector. In Figure 11 diagram (a) we use two CDC connectors to describe the compression and decompression of the behaviour protocols. So, at Level (2) we have the global protocol (P1), at Level (1) we define (P2) client protocol and (P3) server protocol, and finally at Level (0) we have (P4), (P5), and (P6) protocols associated respectively to connection-manager, security-manager and database components.

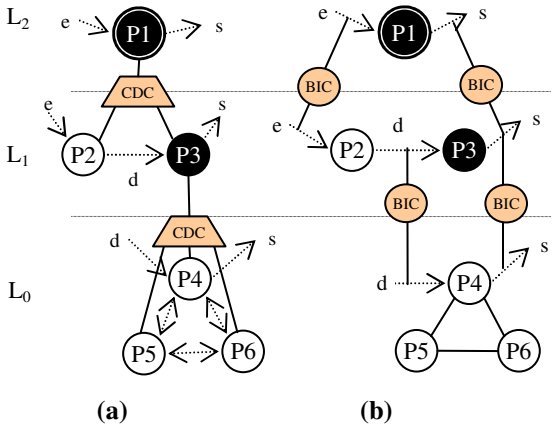


Figure 11. BH with CDC and BIC connectors

In Figure 11 diagram (b) we represent exactly the same hierarchy in which we show the traceability of inputs and outputs using four BIC connectors.

But in Figure 12 we only focus on continuous flow of protocols at each level of the behavioural hierarchy using four AC connectors.

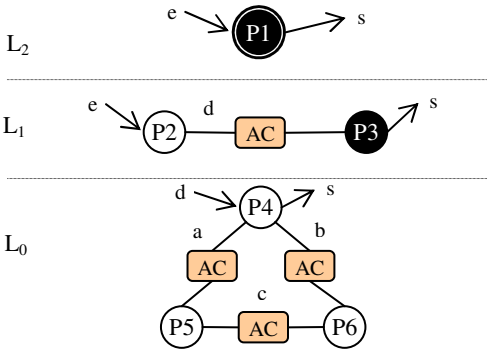


Figure 12. BH with AC connectors

4.3 Conceptual Hierarchy

The conceptual hierarchy is depicted in Figure 13 by diagram (a) at level A_2. At this level we use the SGC connector to generate the five meta-connector types from the first meta-connector defined by C3 metamodel. The SGC meta-connector used at this level is the bootstrap connector for the others meta-connector types. Of course and by the same way we can use the SGC connector to specialise any architecture element described either at level A_1 (architecture level) or at level A_2 (meta-architecture level).

4.4 Metamodeling Hierarchy

The metamodeling hierarchy depicted in Figure 13 by diagram (b) represents the connections between all instances of components used in the application level (A_0) with their component types at the architecture level (A_1), and the connections between all component types of the architecture level (A_1) with their meta-component defined in C3 metamodel. Those connections are realised using *Instance-Of* connectors (IOC). Also, we illustrate in Figure 13 diagram (a) how to generate RPC connector type from AC meta connector using IOC connector instance, and how to generate RPC1, RPC2, and RPC3 connectors from the RPC Connector type using always the IOC connector instance.

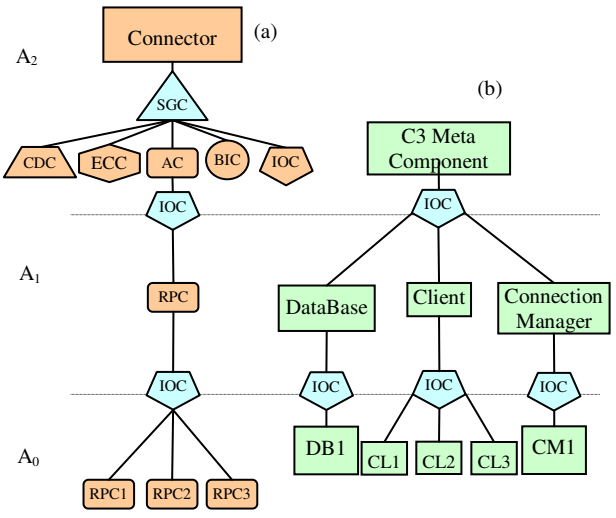


Figure13. Metamodeling hierarchy

5. CONCLUSION

In this work we have defined a minimal and a complete representation metamodel called C3 to describe software architecture and to reason about this architecture from different perspective view. The core elements of C3 are components, connectors and configurations. Elements are assembled using their interfaces. Syntactic and semantic corrections are carried out using respectively interfaces-matching and protocols-matching. Perspective views are defined by different kind hierarchies. Mainly, we use structural hierarchy to describe the structural decomposition hierarchy, behaviour description hierarchy to describe the behaviour function decomposition, conceptual hierarchy to describe sub-architectural elements. The new elements generated by conceptual hierarchy will be used to populate the component libraries. Finally, we use the metamodeling hierarchy to show how we can modify the metamodel C3 and how to use it. Each hierarchy is supported and tooled by explicit connection mechanisms to provide the different form of connections required in each hierarchy. Contrary to the usual ADLs, which define only the attachment connectors, in C3 we define six types of connector to deal with different types of connections. Structural hierarchy uses composition-decomposition connector, expansion-compression connector, and attachment connector. Behavioural hierarchy uses composition-decomposition connector, binding-identity connector, and the attachment connector. Conceptual hierarchy uses the specialization-generalisation connector and finally metamodeling hierarchy uses the instance-of connector.

Some of our ongoing works are: 1- Establishing the relationship between the different views associated to hierarchies. It is a crucial part when we want to bring together different views to the same architecture in a sound way. 2- We work on the development of an UML 2.0 profile to C3 metamodel. This will enable the mapping of any architecture developed using C3 to its corresponding architecture in UML 2.0 notation. The aim of this part is allow using the available tools associated with UML 2.0 to automatically generate the application code corresponding to the architecture.

6. REFERENCES

[1] Allen, R.J. *A Formal Approach to Software Architecture*. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1997.

[2] Amirat, A., Oussalah, M., and Khammaci, T. Towards an Approach for Building Reliable Architectures. *In Proceedings of (IEEE IRI'07) International Conference on Information Reuse and Integration (IEEE IRI'07)*, Las Vegas, Nevada, USA, August 2007, 467-472.

[3] Frakes, W. B. and, Kang, K. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31, 7, (July 2005), 529-536.

[4] Garlan, D., Monroe, R.T., and Wile, D. *Acme: Architectural Description Component-Based Systems, Foundations of Component-Based Systems*. Cambridge Univ. Press, 2000, 47-68.

[5] Lanoix, A., Hatebur, D., Heisel, M., and Souquières, J. Enhancing Dependability of Component-Based Systems. *In Proceedings of the International Conference on Reliable Software Technologies* (*Ada-Europe'07),* 2007, 41-54.

[6] Matevska-Meyer, J., Hasselbring, W., and Reussner, R. Software architecture description supporting component deployment and system runtime reconfiguration. *In the Proceedings of WCOP 2004, Workshop on Component-Oriented Programming* , Oslo, June 2004.

[7] Medvidovic, N., Dashofy, E., and Taylor, R.N. Moving Architectural Description from Under the Technology Lamppost. *Information and Software Technology,* 49, 1, (January 2007), 12-31.

[8] Medvidovic, N. *Architecture-Based Specification-Time Software Evolution*. Ph.D. Thesis, University of California, Irvine, 1999.

[9] OMG. *Unified Modeling Superstructure*. From http://www.omg.org/docs/ptc/06-04-02.pdf, 2006.

[10] OMG. *Unified Modeling Language: Infrastructure*. From http://www.omg.org/docs/formal/07-02-06.pdf, 2007.

[11] Oussalah, M., Amirat, A., and Khammaci, T. Software Architecture Based Connection Manager. *In Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE'07)*, Las Vegas, Nevada, USA, July 2007, 194-199.

[12] Pérez-Martínez, J.E. Heavyweight extensions to the UML metamodel to describe the C3 architectural style. *ACM SIGSOFT Software Engineering Notes*, 28, 3, (May 2003).

[13] Pinto, M., Fluentes, L., and Troya, M. A Dynamic Component and Aspect-Oriented Platform. *The Computer Journal* , 48, 4, (July 2005), 401-420.

[14] Smeda, A., Oussalah, M., and Khammaci, T. MADL: Meta Architecture Description Language. *In Proceedings of the International conference on Software Engineering Research, Management & Applications (SERA'05)*, Pleasant, Michigan, USA, August 2005, 152-159.

[15] Smeda, A., Oussalah, M., and Khammaci, T. Improving Component-Based Software Architecture by Separating Computations from Interactions. *In Proceedings of the ECOOP Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04)*, Oslo, Norway, 2004.

[16] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, JR., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component and message-based architectural style for GUI software. *IEEE Trans. Soft. Eng.,* 22, 6, (June 1996), 390–406.

[17] Schmidt, H., Trustworthy components-compositionality and prediction. *Journal of Systems and Software, Special issue on: Component-based software engineering*, Elsevier Science Inc., 65, 3, (March 2003), 215-225.