Specification of Multimedia Software Systems Using an Object Oriented Architecture Description Language *

Jeffrey J. P. Tsai and Kuang Xu
Department of Electrical Engineering and Computer Science
University of Illinois, Chicago
851 South Morgan St.
Chicago, IL 60607
tsai@eecs.uic.edu

November 29, 1998

Abstract

Despite the growing importance of multimedia applications, we still know relatively little about how to specify, design and maintain this class of complex applications in a systematic manner. The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct and maintain large-scale complex software systems. Under this new paradigm, software engineers are able to do evolutionary design of complex systems through architecture specification, design rationale capture, architecture validation and verification, and architecture transformation. Several architecture description languages (ADLs) have been proposed to support the architecture development under this new software paradigm. Although current ADLs more or less support certain features of object-oriented design approach as it brings significant benefits to software design, but none of them is purely based on object-oriented paradigm. In this paper, an architecture description language — OOADL will be presented as a formal approach for the architecture level system design of multimedia software systems. This language takes object-oriented paradigm as its backbone, and also provide formal semantics for modeling architectures of software system. It also aimed at some other goals such as support for hierarchical refinement, support for reuse of architecture styles, support for analysis and support for exception handling. As another important feature of the language, we will also introduce the default architecture style, which brings extensibility and reusability into the language. Finally, we will use our OOADL to construct part of the architecture framework of a multimedia system, which also serves as a comprehensive example to illustrate the usage and modeling power of OOADL.

*This research was supported in part by Fujitsu Network Communication, Inc.

1 Introduction

Despite the growing importance of multimedia applications, we still know relatively little about how to specify, design and maintain this class of complex applications in a systematic manner. The scales of multimedia software systems are often very large which make the design of these softwares intractable. What makes the design even complicated is the interaction among the thousands of the modules and data structures of the system.

The concept of software architecture has recently emerged as a new way to improve our ability to effectively construct and maintain large-scale complex software systems. Under this new paradigm, software engineers are able to do evolutionary design of complex systems through architecture specification, design rationale capture, architecture validation and verification, and architecture transformation [13]. Several architecture description languages (ADLs) have also been developed for representing and specifying the architectures of software systems. Examples of such languages that are most commonly referred to as ADLs include: Wright [1], UniCon [10], Rapide [6], Aesop [4], Darwin [7], C2 [8], ACME [3], etc.

Though most of the current ADLs more or less support certain features of the object-oriented design approach, none of these languages is based on a pure object-oriented paradigm. Object-oriented design approaches have already been identified with many advantages: it models the problem domain more naturally, allows extensibility and reusability, and inheritance, etc. Therefore, an object-oriented architecture description language will explore these advantages in abstract level architecture design. In this paper, we introduce an object-oriented architectural description language — OOADL. This language takes object-oriented paradigm as its backbone to support the reusability and extensibility features of normal object-oriented programming languages. It also provide a formal semantic in modeling software architectures so that it make the architecture level design more concrete. As our first attempt, we use Z notation as our formal semantics. This ADL also aims at some other goals such as support for hierarchical refinement, support for default architecture styles, support for analysis, etc. We will explain how to use OOADL for the architecture level system design of multimedia software systems.

This paper is organized as following: in section 2, we discuss the goals that our OOADL intends to achieve; in section 3, we give out the outline of our OOADL, the formal grammar of it will be given out in the BNF form in APPENDIX A; in section 4, we present an architecture style — client-server style as a sample of our default architecture style; from section 5, a comprehensive example is presented, which includes the brief introduction of a heterogeneous software architecture for multimedia system, and the specification of part of that multimedia software architecture using our OOADL.

2 Language Goals

Architectural design is the intermediate step between the requirement analysis stage and further design stage, it's the first attemption to map the user requirements(non-computer related domain) to the system construction(computer science knowledge condensed domain), neverthless, this step acts as a very important role during the whole software life cycle. The major works to be done at this architectural design stage are depicted in Figure 1.

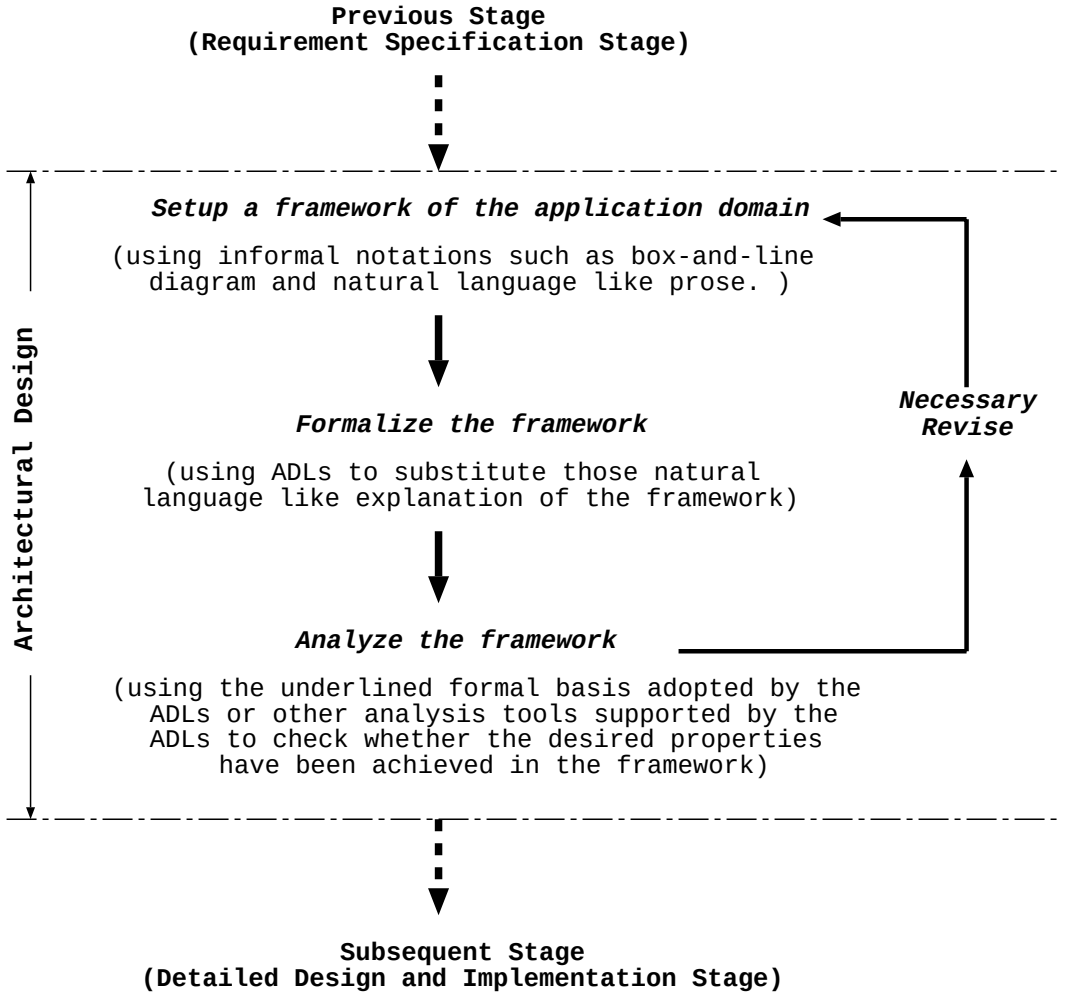


Figure 1: **Architectural Design Stage**

The first step in architectural design is to construct a framework of the system, it just gives out a very rough outlook of the system, and the framework might (and in most case it is) just be represented by completely informal method. But unfortunately, software engineers can acquire little informations from such an informal framework. Further more, such kind of framework also might lead to misunderstandings between different development team, or among different members

of the same team as it is too 'flexible'. Thus, formalism finds its place in the architectural design, and architecture description languages (ADLs) with formal semantics come out for this purpose. Further more, many ADLs either provide strong analysis capabilities by themselves or can be combined with other mature verification tools independent of software architecture [5]. Thus, the software architectures together with the ADLs can take the full responsibility of the architectural design stage. From this point of view, it's obvious that ADLs play an important role during the architectural design stage (at least it acts in the two steps out of total three and which are also more concrete steps). This important role ADLs played leads us to the following requirements our OOADL should satisfy.

Support for Object-Oriented approach: As we stated in section 1, object-oriented approach possesses many good properties. Among those properties, the most important one is that it models the problem domain instead of starting from the programming domain, (the imperative and functional paradigm are more closer to the later one). Software architecture design is the first stage after the requirement specification, it's the first attemption to provide the solution of the problem domain, the closer it is to the problem domain, the better the framework meets the user requirement, and also the less for future revision. So, it seems the object-oriented approach is more suitable for architectural design. Other good properties of Object-Oriented approach also include reusability and extensibility. Actually, as the reader will see in later sections, we will provide several default architecture styles which enclose the basic properties of those architecture style, but leave places for users to extend them and fit them into the different specific problem domains, and this approach is purely aimed at the reusability and extensibility features. But in order to make it best fit for architecture design, in which case the inter-components communication plays a role as important as that of the components themselves, we adapted the pure message passing in normal sense object-oriented paradigm and explicitely specify the export and import messages together with the receiver and provider of those messages to make the inter-components communication as first-class language construct.

Support for hierarchical refinement: There are two kinds of meanings for hierarchical refinement. The first one is user can construct the software architecture via a step-wise approach, i.e., divide and conquer. This feature is useful because the purpose of frameworks and ADLs is for large and complex system design. The second one is the atomic elements of one level might be a composite structure of another level, this feature is particular useful in heterogeneous architectures which will be illustrated in the comprehensive example in section 5. Our OOADL should suppor both of these two features.

Support for reuse of architecture styles: Current days, certain kinds of architecture styles have been widely accepted to be used in constructing software architecture frameworks. Such architecture styles include pipe-and-filter, client-server, layered system, feedback system, event-based system, and some more. In our OOADL, we attempt to provide those architecture styles as default ones supported by the system, i.e, to provide a set of default architectures. But instead of making them fixed ones, we'd like to only specify the basic characteristics of these

styles and remain the specification open to let users add user specified and system dependent features. Thus, a broad class of architecture styles can be constructed either from scratches or by extending default architecture styles. That is, user can inherit the features defined in the default styles, then refine or restricte the properties of the style to fit his own purpose.

Support for analysis: From figure 1, we can see the last (but not least) step in architecture design stage is to analyze the framework to find out whether the desired properties have been achieved. Analysis is necessary for architecture design stage as it can eliminate the undesired properties which will cost lots of efforts to correct at the later stages. As the framework itself is informal, which conveys insufficient information for analysis, it is ADL's responsibility to carry this task. There're two possible approaches to performa analysis using ADLs. The first one is to utilize the underline mathematical formalism of the ADL. The second one is to apply the existing analysis tools independent of the software architectures to the software architecture formalized by ADLs. This approach requires the ADL specification be friend enough to the different input languages of those tools, i.e., require a straightforward transformation from the ADL to the different input languages of the tools. Our ADL should support at least one of the above two approaches.

Support for exception handling: Though not wanted, exception conditions are unavoidable, especially in large scale and complex systems. We would like to provide exception handling mechanisms in our ADL by make it an explicit part of the language.

3 The OOADL

In this section, we give an outlook of our OOADL, and we use a simple example to illustrate the usage of the language, a complete set of syntax of the language is given out in APPENDIX A.

3.1 Overview

As stated earlier, our OOADL is based on an object-oriented paradigm, every components of the system framework is an **OBJECT**. For every such **OBJECT**, it will contain two **BEHAVIOR** parts: one is **INTERNAL BEHAVIOR**, which represents the computation steps performed inside the component; the other is **ARCHITECTURAL BEHAVIOR**, which specifies the communication with other components inside the architecture. While as we intend to achieve the hierarchical refinement goal setup in section 2, we introduce the keyword **PARTS**. With this keyword, an intuitive approach to use our OOADL to specify system architecture is first to specify the whole system as an **OBJECT** with all of its components declared under **PARTS**, then specify each components as an **OBJECT** with all of their sub-components declared under **PARTS** at a lower level, and continue this process until reaching the lowest level at which no further decomposition is needed. Further more, to enhance the reusability and inheritance features, an *Abstract-Relation-Type* that states the relationship among **OBJECT**s will be specified for each **OBJECT**. Actually, the underline structure of our OOADL is developed from FRORL [12], the key words *a_kind_of*, *a_part_of*

and *an_instance_of* which represent the generalization, aggregation and instantiation relationships among component objects respectively, are reserved to represent the *Abstract-Relation-Type*. And also, for the **BEHAVIOR** part, we retain the pre-condition (**PRECOND**), action (**ACTION**) and alternative action (**EXCEPTION**) syntax from FRORL. The major differences between our OOADL and FRORL are: first, the frame concept of FRORL no longer exists in OOADL, this is because we adopt a pure object-oriented paradigm in OOADL and the activities of each object (which is the activity frame in FRORL) are enclosed in the specification of object's behavior part; second, as a consequence of enclosing object's activities into the behavior part, the behavior specification part can contain more than one activities, i.e., it's no longer the case that only one activity specification per activity frame like that in FRORL. Thus, an outlook of the OOADL would be like:

OBJECT: <component-name>
Abstract-Relation-Type: <object-name>
PARTS: <object-name>, <object-name>, ...
INTERNAL BEHAVIOR:
PRECOND: <pre-condition specification>
ACTION: <action specification>
EXCEPTION: <exception handling>
⋮
ARCHITECTURAL BEHAVIOR:
IMPORT PART:
IMPORT: <message>
PROVIDER: <object-name>
⋮
EXPORT PART:
EXPORT: <message>
RECEIVER: <object-name>
⋮

Note: actually, the two terms — object and component are inter-changeable in the above specification.

As we claimed in previous sections, a formal mathematical tool is needed to construct the semantics of our OOADL. Such formalism will be used to specify the *pre-condition specification, action specification, exception handling* and *constraints specification* parts during system construction. Now there are several kinds of mathematical tools available such as Z Notation, CCS and CSP, as a first attemption, here we selected Z Notation to construct the architecture specification.

3.2 A Simple Example

In the following paragraphs of this section, we would like to give a simple example specification to illustrate the usage of our OOADL. for detailed information about Z Notation, interested readers could refer to the Z Notation user manual [11].

Suppose we want to construct a small client-server system, the server actually is a database which contains personal archives such as name, social security number, address, etc., for a certain group of people, while the client is the terminals for user to input query information, to simplify the system and achieve the best demonstration purpose, we assume only one operation can be performed by the client, namely, submit query request to the server to check whether a person with a given name is in the database. The system structure will be like:

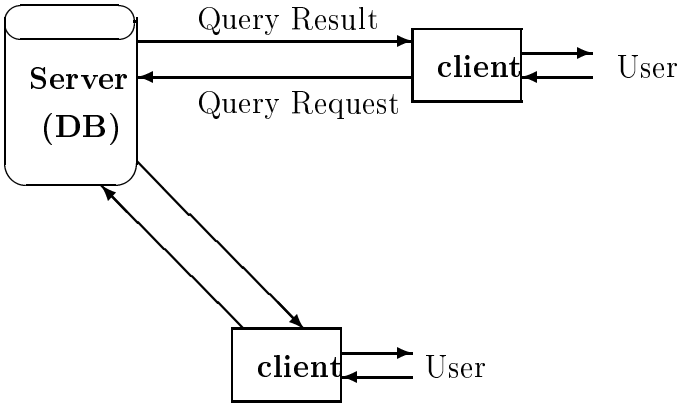


Figure 2. A simple example

From this framework, the first look is the system should have one server and several clients as its components, their relationship should be the so called "client-server system". There should be communications between clients and server, but no communications exists among clients. The communication from client to server is the request for query, while the communication from server to client is to convey the result of the query. Assume *NAME, SS#, ADDR, PHONE#* and *PLATFORM* are *basic types*, and the type *DB* has been introduced in as:

DB

RECORD: **P** { NAME, SS#, ADDR, PHONE#}

search_name: NAME → RECORD

Thus, based on the step-wise approach, we concrete the structure of the whole system using our OOADL as follows:

OBJECT: Query_system

A KIND OF: CLIENT_SERVER_SYSTEM
PARTS: serv, **F** clnt
INTERNAL BEHAVIOR:
PRECOND: Query_system = ϕ
ACTION:

initialize

Query_system: client_server_component
serv: DB, clnt: PLATFORM

---

serv=Query_system.server
#Query_system.client=1 $\wedge$ clnt $\in$ Query_system.client

EXCEPTION: nil
ARCHITECTURAL BEHAVIOR: nil

Figure 3. Specification of the *Query_system*

Figure 3 is the highest level specification. It states that the Query_system is *a kind of* client_server_system, which means it will inherit all the properties defined in the default architecture: client_server_system, except for those overidden in the Query_system definition. (Nothing is overriden in this example as the Query_system is a common client_server_system without any specialties.) Note: for the default architecture, please refer to section 4, client_server_system will be presented as an example in that section. The only action will take place is the initialization, which will create one server and a client set which has only one client at first. The architectural behavior part is nil because as a whole system, the Query_system won't be connected to anything else.

Now, the next step is to specify the two parts of the Query_system, namely, *serv* and *clnt*:

OBJECT: clnt
A PART OF: Query_system
PARTS: nil
INTERNAL BEHAVIOR:
PRECOND:

content: IMPORT $\cup$ EXPORT $\rightarrow$ seq DATA

$\exists$ in: IMPORT_PART | in.IMPORT=inputName $\wedge$ content(in.IMPORT) $\neq \phi$
$\exists$ out: EXPORT_PART | out.EXPORT=queryName

ACTION:

Transfer

Δ clnt

content (in'.IMPORT) = *tail*(content(in.IMPORT))

content (out'.EXPORT) = content(out.EXPORT) $\frown$ *head*(content(in.IMPORT))

EXCEPTION:

$\forall$ in:IMPORT_PART, out:EXPORT_PART

| in.IMPORT $\neq$ queryName $\vee$ out.EXPORT $\neq$ result

• register_as_client[clnt]

PRECOND:

$\exists$ in: IMPORT_PART | in.IMPORT=result $\wedge$ content(in.IMPORT) $\neq \phi$

$\exists$ out: EXPORT_PART | out.EXPORT=returnResult

ACTION: Transfer

EXCEPTION:

$\forall$ in:IMPORT_PART, out:EXPORT_PART

| in.IMPORT $\neq$ result $\vee$ out.EXPORT $\neq$ returnResult

• register_as_client[clnt]

ARCHITECTURAL BEHAVIOR:

IMPORT PART:

IMPORT: inputName
PROVIDER: outside
IMPORT: result
PROVIDER: serv

EXPORT PART:

EXPORT: queryName
RECEIVER: serv
EXPORT: returnResult
RECEIVER: outside

Figure 4. Specification of the *client*

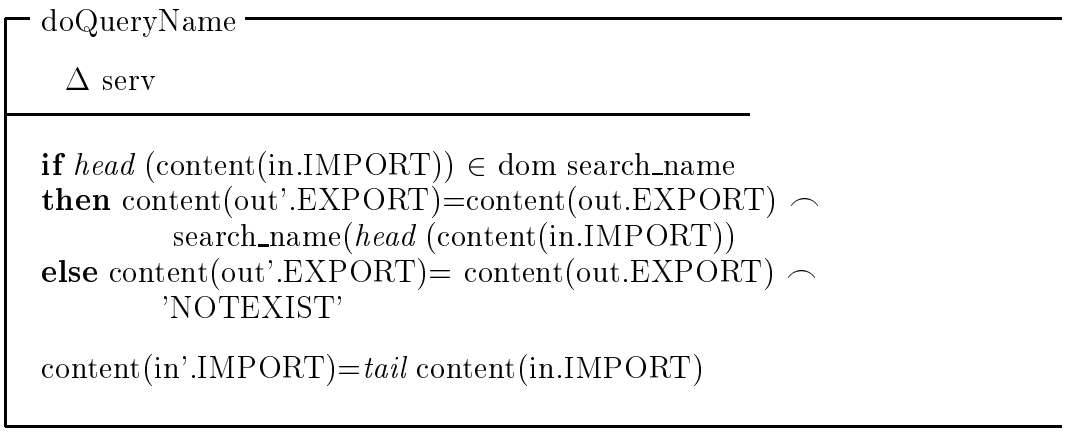
In the above *client* part specification, we need to achieve the following goals "implicitely" enclosed in the framework shown in Figure 2. First, client is a component of the *Query_system*, thus, "*A PART OF:* Query_system" is a proper specification of *Abstract-Relation-Type* as a counterpart of

the *PARTS* specification of *Query_system*. Second, from the architecture, each client should have up to four kinds of communication as indicated by the four arrowed lines in the figure, which are user input, submit user request to server, get search result from server and return the result to the user respectively. So, we setup two import parts among which one is used to get user input with "outside" as its provider and the other is used to receive query result from server with "serv" as its provider, we also setup two export parts among which one is used to submit query request to server with "serv" as its receiver and the other is used to return the query result to user with "outside" as its receiver. Third, the *client* part should do nothing itself but only convey messages between user and server as indicated by the natural language description of this example. Thus, the action part of the specification only consists of two actions: any data input in the *inputName* import will trigger the client to submit the data to the *queryName* export, any data input in the *result* import will trigger the client to return the data to the *returnResult* export. Notice that as the actual action in both case is to transfer data, only one action—Transfer is defined. Fourth, the component itself should only concern about its internal operations regardless the data supply is from where or the result generated should go to where, thus, in the above specification of the action part, only the IMPORT or EXPORT information is needed for I/O, the PROVIDER and RECEIVER are totally uninvolved and they will only appear in the ARCHITECTURAL BEHAVIOR part. This feature is best reflected from the action "Transfer": though it deals with two different cases which have different data supplier and consumer, only one "Transfer" is needed as it only cares about the fact that there's one data resource and one data sink. Finally, we setup a simplified exception handling in the specification, that is, in both action cases, if one of the total four communication routes of the "clnt" is disconnected, we just redeclare the "clnt" as one of the clients of the system. (we use the *register_as_client[argument]* schema as it is inherited from the default basic style *CLIENT_SERVER_SYSTEM*, please refer to section 4 for detail of this schema).

OBJECT: serv
A PART OF: Query_system
PARTS: nil
INTERNAL BEHAVIOR:
PRECOND:

$\exists$ in: IMPORT_PART | in.IMPORT=queryName $\wedge$ content(in.IMPORT) $\neq \phi$
$\exists$ out: EXPORT_PART | out.EXPORT=result

ACTION:



EXCEPTION: nil
ARCHITECTURAL BEHAVIOR:
IMPORT PART:
IMPORT: queryName
PROVIDER: clnt
EXPORT PART:
EXPORT: result
RECEIVER: clnt

Figure 5. Specification of the *server*

Now let us look at the server part. First, the server is a component of the *Query_system*, thus, like that of client, "*A PART OF:* Query_system" is the proper specification of *Abstract-Relation-Type* as an echo of the *PARTS* specification of *Query_system*. Second, the server will serve the query requests from each server in a first-in-first-out sequence as there might be a batch of query requests from clients, this is realized in the action part by "content(in'.IMPORT)=*tail* content(in.IMPORT)" and "content(out'.EXPORT)=content(out.EXPORT) ⌢ search_name(*head* (content(in.IMPORT))". Third, as stated earlier, the server will only handle the query-by-name request, the only action of the server is to get triggered and execute the "doQueryName" operation if there's data input in its import port, and output the query result to the client through the export port.

4 Default Architecture Styles

4.1 What is default architecture sytles

It is so common nowadays that when someone mentions a system architecture is a pipe-and-filter style or a client-server style, what in your mind as a first reaction to it must be a brief diagram depicting those structures. Though it might not be exactly the same as it is supposed to be according to the designer's original intention, it can roughly reflect the overall structure and also reflect

most of the characteristics it possesses. From this phenomenon, we can learn two facts: First, some architecture styles are used widely and become so popular that the major characteristics of those styles are well known and accepted by the users as a common law. So no matter what kind of interpretations different users will give to them, those basic characteristics will be left unchanged. Second, there do exist several architecture styles that have been widely used in system design and their basic features are also recognized as closely related to those styles, such styles include pipes-and-filters, layered systems, client-server systems, event-based systems and feedback systems, etc. Thus, from the above observation, we start to think why not providing users a set of specifications for those most popular styles, so that users can directly utilize those styles in architecture specification without defining them redundantly.

Bearing this in mind, we provide the default architecture styles in our OOADL. For those default styles, we will specify the most common properties of them, and user can access those default styles by specifying their own system is *A KIND OF* those default styles. (See the specification of *Query_system* in section 3) When considering the alternation of these default styles in different system design (such as user need to add more features to the basic ones, or certain basic features are undesirable in the user's own system), thanks for the object-oriented paradigm we adopted, user can easily using the refinement or restriction features of inheritance to modify the default styles to fit into their own purpose.

4.2 The client-server default architecture style

As we selected client-server style in our ADL example, we would like to illustrate the default style specification for client-server system here also. To define a default client-server structure, the first thing is to find out the basic features of this style that are commonly accepted by different users, our observation is presented as following.

general requirements/constraints for client-server system:
1. only 1 server per system; (note for those systems with more than 1 server, we can view it as a composition of several smaller client-server system each only contains one server);
2. zero or more clients might be in the client-server system;
3. clients request service, service provided by server, services are provided to clients as a response of clients' request;
4. to get services from server, one must first register as client of this server of this particular system;
5. every client of the system is linked to the unique server;
6. hierarchical structure is possible: client of one system might be the server of another system at the same time, so for server;

Based on the above assumptions, we first define the components of the client-server-system as follows:

[PLATFORM, SERVICE, COMMAND]

client_server_component

server: PLATFORM
client: **P** PLATFORM
available_service: COMMAND → SERVICE
linkage: client → (client ↔ server)

---

dom linkage = client

Then we define the behavior of the client-server system as follows:

client_server_behavior

cs: client_server_component
serve_action: client × COMMAND → client × SERVICE

---

1. ∀ c: cs.client • (c ≠ cs.server) ∨ (∃ cs':client_server_component |
   cs' ≠ cs • c=cs'.server)
2. $\exists_1$ s: cs.server | ∀ c: cs.client • (c,(c,s)) ∈ linkage
3. serve_action ≙ provide_service ∘ submit_request

(Note: though not included in standard Z specifications, we numbered each specification for the convenience of later explanation.)

The *provide_service* and *submit_request* used for defining the *serve_action* above are defined as:

submit_request: client × COMMAND → (client ↔ server) × COMMAND

∀ m: PLATFORM, cmd: COMMAND | m ∈ client
• submit_request(m,cmd) = (linkage(m), cmd)

provide_service: (client ↔ server) × COMMAND → client × SERVICE

∀ c: client, cmd: COMMAND • provide_service(linkage(c), cmd)
= (c, available_service(cmd)) ∨ cmd ∉ dom available_service

Now let's see how each basic requirements of the default client-server system is met in the above specifications. In the definition of *client_server_component*, we declare the type of server and client as *server:* $PLATFORM$ and *client:* $\mathbf{F}\ PLATFORM$, which indicate that *server* is a type of $PLATFORM$(only one server exists) and *client* is a set of $PLATFORM$s (one or more clients), thus, we got requirements #1 and #2 clarified. As we define the communication between client and server with a function *linkage* which maps each client to the bidirectional communication route $client \leftrightarrow server$, we request each *client* of the system be connected to the server to meet the request #5, that is $dom\ linkage = client$. The remaining requirements/constraints are specified in the schema *client_server_behavior*: in specification 1, we declared that for every *client* of the system, it either should not be the server of this system or it can be the server of another system at the same time, which is exactly the requirement #6. Specification 2 also expressed the requirement #5, but with the stress on 'unique server'. Finally, in specification 3, we define the service providing procedure as a backward relational composition of the two function *submit_request* and *provide_service*, which refers to *client* post a command on the linkage between *client* and *server* first, and then *server* collect that request and forward back the service related to that request to the *client*, thus, requirement #3 also fulfiled. And as indicated in specification 3 that only *client* and *server* involved in the service providing procedure, requirement #4 is naturally reached.

Besides the above requirements we should put in the default *client_server_system* specification, certain kinds of common operations should also be provided. Such operations include register a platform as a client of the system, retract from the system when the platform no longer want to be a client of the system, add a new service to the service set that the server can provide and remove a 'stale' service from the server etc. These operations are illustrated in the following Z schemas:

register_as_client [m]

Δ client_server_component
Ξ client_server_behavior
m? : PLATFORM

---

server' = server
client' = client $\cup$ {m?}
available_service' = available_service
linkage' = linkage $\cup$ { m? $\mapsto$ (m? $\leftrightarrow$ server') }

(Note: instead of checking whether the request m is already a client of the system we simply override the original one.)

retract [m]

Δ client_server_component
Ξ client_server_behavior
m? : PLATFORM

---

server' = server
client' = client / {m?}
available_service' = available_service
linkage' = linkage / { m? $\mapsto$ (m? $\leftrightarrow$ server') }

add_service [cmd, svc]

Δ client_server_component
Ξ client_server_behavior
cmd?: COMMAND, svc?: SERVICE

---

server' = server
client' = client
available_service' = available_service $\cup$ { cmd? $\mapsto$ svc?}
linkage' = linkage

remove_service [cmd, svc]

Δ client_server_component
Ξ client_server_behavior
cmd?: COMMAND, svc?: SERVICE

---

server' = server
client' = client
available_service' = available_service / { cmd? $\mapsto$ svc?}
linkage' = linkage

Figure 6. Specification of the basic operations of the default *client-server* system

Now, after we defined the system component, system behavior and the general operations, we can define the default client-server architecture style as follows:

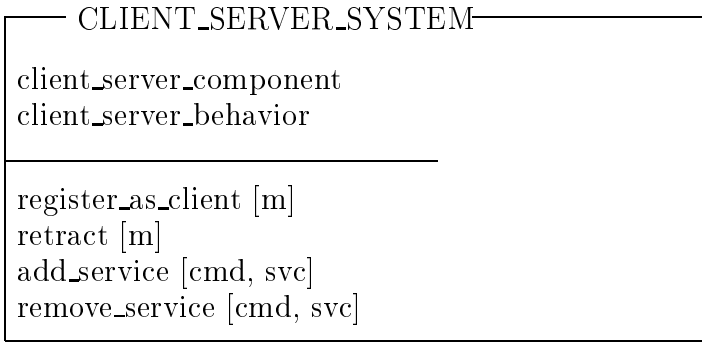


Figure 7. Specification of the CLIENT_SERVER_SYSTEM

Besides the above client-server default style, we can also create other similar well-known architecutral styles. As another example, we specify the layered-style in APPENDIX B, this style will also be used in section 5 as a default structure that the *regional_server_frame* will base on.

5 A Comprehensive Example: To Specify the Heterogeneous Distributed Multimedia Architecture Using OOADL

Modern multimedia systems normally are distributed systems which consist of many components and the communications between components are significant. Thus, software architecture approach is especially suitable for multimedia system design. Actually, many recent research topics in multimedia systems are concerning with the system frameworks and structures. To apply an ADL to construct the multimedia system framework will be an attractive application of the ADL. In the following sections, we'll apply our OOADL in specifying the framework of a sample multimedia system. Considering the space limitation of this paper, we'll intentionally omit the detailed Z specifications of the system, and only provide the structural specifications instead. The complete specification is contained in the APPENDIX C.

5.1 The architecture framework

The sample multimedia system used in this paper is based on the client-server paradigm. There are totally two kinds of servers in the system, namely, the global server and the regional server. There is only one global server in the system, while the total number of regional servers depend on the condition of the communication network and the services requested by the clients.

5.1.1 Regional Server

The regional servers are actually the central part of the system. One important aspect of the regional server is dynamic adaption. Actually, the regional server is an intelligent agent, it should monitor its own region and assess the current situations of this region, then determine whether to keep the current status or to make certain adaptions, i.e., degrade the original region further into one or more smaller groups as new regions, then duplicate the original regional server and launch the copies to those new groups, and finally gear these copies as the new regional server of those new groups to make the system functional. Certain criteria for adaption are applied here:
1. When the requested services in the original region increase drastically and make the load of the regional server become too heavy to handle by itself which lead to the sharp decreasement of efficiency and effectiveness. In such cases, the server needs to assess the situations based on the location of the clients and the different requests combinations from the clients.
2. When the regional server receives explicit requests from applications that several end users want to form a special purpose group and request for a server which can dedicate in serving for this group. In this case, the original server just need to send a copy as the response to those request.
3. When communication congestion occurs inside the region, the server also needs to analyze the network topology of this region via the regional table it keeps to determine whether adaption needs to be performed or not.

One possible solution to problem 1 is to check each client nodes of the over-loaded region, then split the original region into smaller regions with each resulted regions has a working load not exceeding the expected load of the regional server, and finally, setup a new regional server for each resulted regions. We'll omit problem 2 and problem 3 as they're not going to be used in our ADL specification.

5.1.2 Global Server

As we have seen above, the regional servers are capable of self-replication and dispersing. But this is only one direction of adaption, the other direction is coalescence. Consider the situation that there're several regional servers each with a slight load compare to the normal working capacity, which is inefficient as it will increase the control and communication overheads without fully utilize the ideal capacity of the regional server. Thus, it is necessary to provide some way to re-assess the overall system conditions and combine regional servers together by eliminating unnecessary ones. As this work needs to analyze the overall regional informations which the regional server itself is incapable of, it is necessary to introduce a global server into the system to accomplish this task. Besides that, consider the situation that one client of a region want to join a video conference held by several clients of another region, in which case a reallocation of the clients will be needed. The regional server itself cannot accomplish this work without query overall client-server informations at global level.

The most important part of the global server is a **server table** which will contain the following

information:

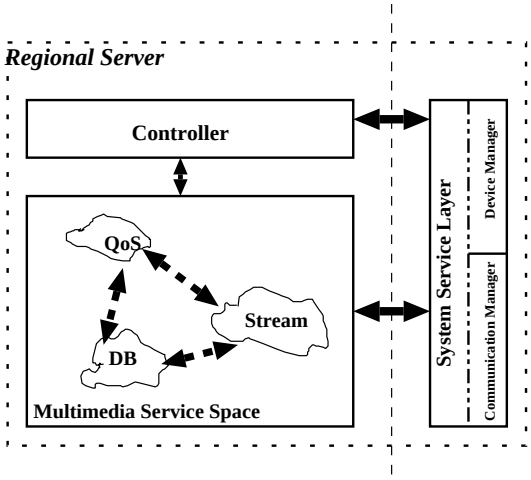
Server Table			
region_id	**preferred load**	**actual load**	$\cdots$
id_1	server_time_1	server_time_a	$\cdots$
id_2	server_time_2	server_time_b	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$

region_id is the unique identifier of each regional server, as one server manages one region, this identifier also is the unique indentifier of each region;
preferred load denotes the most preferrable working capacity of each regional server; while *actual load* denotes the actual working load reported by each regional server.

Actually, besides to keep and update the **server table**, the only job of global server is to optimize the regional server allocation. This can be done by querying the **server table** and comparing the *preferred load* with *actual load*, if large discrepancy exists, then the global server should start relocation/coalition operations. The relocation/coalition operations can be accomplished just by sending a message to the regional servers to be eliminated and sending another message to the regional server which will take the responsibility of the regions belongs to the eliminated servers.

5.1.3 Functional Components of the Regional Server

The regional server is the central part of the sample multimedia system, besides provide the adaption and optimization mentioned in section 5.1.1, it's also capable of providing the basic multimedia services such as QoS service, multimedia inter- and intra-stream synchronization and multimedia database accessing functions, etc. The internal architecture of the regional server is like following:



From this figure, we can see there are three major parts inside the regional server. They are the

Controller, multimedia service space and system service manager. These three components actually can be viewed as layered relationship, that is, the controller layer will monitor and coordinate the actions of the multimedia service space layer, while multimedia service space layer will supply certain operation informations to the controller layer which will be used for the dynamic adaption between different regional servers. Meanwhile, both controller layer and multimedai service space layer need to communicate to the system service layer to get information from outside and to send instructions to clients.

Controller is the central part of the regional server, it has three major functions: fulfill dynamic adaption; coordinate the inter-relationship between the functional components inside the multimedia service space layer; collect and supply informations for the global server.

System Service Layer is a intermediate area which is used to hide the unnecessary low level implementation details from both the upper layer of regional server and the applications on the client site. As shown in the above Figure, there are two components inside the system service layer: the *device manager* and the *communication manager*. The major responsibility of the *device manager* is to interact with the system level operations such as I/O, device driver, format conversion, and so on. While the *communication manager* will handle the tasks such as provide basic network services (such as establish communication routes, select appropriate communication protocols, etc.); supply network informations (such as the congestion status) to the controller for dynamic adaption.

5.2 The specification

Similar as that of the simple example illustrated in section 3, the first step to construct the specification is starting from the top and define the whole system as an object as follows:

OBJECT: multimedia_system
A KIND OF: CLIENT_SERVER_SYSTEM
PARTS: global_server
regional_server
client

INTERNAL BEHAVIOR:
PRECOND: multimedia_system = ϕ
ACTION: Initialize
EXCEPTION: nil

ARCHITECTURAL BEHAVIOR: nil

Figure 8. Specification of the *multimedia_system*

In the above specification, we define the whole multimedia framework as an object with three components: the global_server, the regional_server and the client. As defined in the action part, the global_server is a kind of *PLATFORM*, while both of regional_server and client are *PLATFORM* set, this is because there should be only one global_server in the entire system while there might be a set of subsystems with regional_server and client act as components. The only work that belongs to the whole system is to complete the initialization operation, and thus only one action is specified in the internal behavior part—*Initialize*.

During the initialization procedure, we specify the component–global_server as the *server* of a *client_server* system "global", and the component–regional_server as the *client* of "global". After that, we specify that for every regional_server, it is the *server* of one of the subsystems of the set "regional", which should be different from each other and also different from "global". Finally, every element of the component–client are also distributed into those subsystems of "regional" and become the *client* of them. Thus, we constructed a two-level system with both levels are *client_server* style, and for all the subsystems in either level, they will inherit all the properties and operations defined in the CLIENT_SERVER_SYSTEM of section 4.

After finish the overall system specification, the next step should be the individual specification of the three components introduced in the *PARTS* declaration, i.e., the global_server, the regional_server and the client. Let's first look at the global_server. As stated in section 5.1, the global_server maintains a global *server_table* and the major work of the global_server will be done based on this table. So, we first specify the *server_table* as follows:

[SERVER_TIME]

region_id: PLATFORM,
prefer_load: SERVER_TIME,
actual_load: SERVER_TIME,

Server_Table

record: **F** {region_id, conn_list, prefer_load, actual_load}

$\forall rcd_x, rcd_y$: record $\bullet rcd_x$.region_id=rcd_y.region_id $\Rightarrow rcd_x = rcd_y$
get_prefer: region_id $\rightarrow$ prefer_load
get_actual: region_id $\rightarrow$ actual_load

After we got this *server_table*, we can specify the global_server as following with the major actions are modifying this table:

OBJECT: global_server
A PART OF: multimedia system
PARTS: nil

INTERNAL BEHAVIOR:

PRECOND: low_loading
ACTION: Optimize
EXCEPTION: unsatisfiable_optimization

PRECON: update_request
ACTION: update
EXCEPTION: update_before_register

PRECON: register_request
ACTION: Add_New
EXCEPTION: already_exist

PRECON: query_request
ACTION: Query_Info
EXCEPTION: nil

ARCHITECTURAL BEHAVIOR:

IMPORT PART:

IMPORT: info_update(region_id, actual_load, prefer_load)
PROVIDER: regional_server

IMPORT: add_new(region_id, actual_load, prefer_load)
PROVIDER: regional_server

IMPORT: query_info(region_id)
PROVIDER: regional_server

EXPORT PART:

EXPORT: query_result(rcd)
RECEIVER: regional_server

Figure 9. Specification of the *global_server*

Now let's examine the definition of the global_server. First of all, it is declared as *A PART OF: multimedia_system*, which matches the **PARTS** specification of the *multimedia_system*, and as global_server itself can no longer be decomposed into further structures, no *A KIND OF* relationship is needed for it, this is also reflected in the **PARTS** specification which contains nil. Then, we specified four kinds of internal behavior for the global_server:

Optimize takes the responsibility of the system optimization, i.e., to combine certain regional_servers when all of their works can be done by one regional_server instead of serveral. This action is triggered if there exists regional_server that its actual_load exceeds the threshold. The action is also quite straightforward: check all the other regional_serverss_x, if any of them meet the constraints "$get\_actual(s) + get\_actual(s_x) < get\_prefer(s) - offset$", then combine s_x into s, i.e., retract s_x from the system, transfer all the clients of s_x to s, and update the

Server_Table. The exception here states that if there's no satisfiable s_x, just leave the whole system unchanged.

Update will be triggered if there's data coming in at the IMPORT—info_update and the first argument is a valid region_id. The actions of "Update" is simply to refill the related field in the Server_Table based on the data supplied in the IMPORT. The exception handling associated with this action is when the first argument is not a valid region_id, i.e., it's not maintained in the current Server_Table, just treat it as a request for registering as a new regional_server and invoke the action "Add_New".

Add_New will be triggered if there's data coming in at the IMPORT—add_new and the first argument is not an already existing region_id in the current Server_Table. The actual operation just adds a new record with the related information of the new region_id into the Server_Table. The exception part of this action is the counterpart of "Update" action, i.e., for those requests with region_id already in the Server_Table (it's not a new regional server), we treat it as a request for updating the region's current status information.

Query_Info will be triggered if there's data coming in at the IMPORT—query_info and the first argument is a valid region_id of an existing record of the Server_Table. The operation performed here is to output the record associated with that region_id to the EXPORT—query_result. The exception handling associated is to output a nil result if the region_id being queried is not a valid one.

The next step is to specify the regional_server. Unlike the global_server which is unique in the whole system, the regional_server is a set of servers which have the same behavior but different environment, i.e., different clients set and topology, thus, we will first create a regional_server_frame, which acts just like the class in normal object-oriented paradigm, and make the instances of this regional_server_frame be the actual object in the entire system architecture. Thus, the actual definition of regional_server could be like:

OBJECT: regional_server
AN INSTANCE OF: regional_server_frame
PARTS: ···
INTERNAL BEHAVIOR:
⋮
ARCHITECTURAL BEHAVIOR: nil

Figure 10. Illustration of the usage of *AN INSTANCE OF* relationship

Each instance can define certain internal behaviors besides those already specified in the regional_server_frame, but for the architectural behavior, nothing more need to be done as all of them have been setup in the specification of regional_server_frame. Thus, it brings a lot of flexibility.

The regional_server_frame is specified as follows:

OBJECT: regional_server_frame
A PART OF: multimedia_system
A KIND OF: LAYERED_SYSTEM
PARTS: controller
system_service_layer
multimedia_service_layer
INTERNAL BEHAVIOR:
PRECOND: ϕ
ACTION: Initialize
EXCEPTION: nil
ARCHITECTURAL BEHAVIOR: nil

Figure 11. Specification of the *regional_server_frame*

In the above specification, we declared that regional_server_frame is *A PART OF: multimedia_system*, thus, when a new instance of regional_server_frame is created like that in Figure 10, the **PARTS** declaration of *multimedia_system* will be matched. Note that in the specification of regional_server_frame, except for the "initialize" action, no concrete actions is defined in the internal behavior part and the architectural behavior is even declared as nil. This is because regional_server_frame itself acts as an abstract object here, and all the concrete operations will be done by the its three components: controller, system_service_layer and multimedia_service_layer. The purpose of doing so is first to provide a "container" to clone the three components together; and second is to provide the constraints on the three components as specified in "initialize" action.

Controller is the core part of the regional_server, and it will accomplish most of the concrete operations of dynamic adaption. For simplification purpose, in the following specification, we only illustrate one of the adaption cases here, the other adaption case can be specified similarly.

OBJECT: controller
A PART OF: regional_server_frame
PARTS: nil
INTERNAL BEHAVIOR:
PRECOND: overloaded
ACTION: Adaption_1
EXCEPTION: nil
⋮
ARCHITECTURAL BEHAVIOR:
IMPORT PART:
⋮
EXPORT PART:
EXPORT: new(id, actual, prefer)

RECEIVER: system_service_layer

⋮

Figure 12. Specification of the *controller*

The action "Adaption_1" of the **controller** is specified for the adaption case #1 given in section 5.1.1. It will be triggered whenever the actual_load of the regional_server exceeds the prefer_load. The meaning of the "Adaption_1" schema is: find out an seperation of all the clients of this regional_server which satisfies the actual_load of every client set of this seperation will not exceed the threshold as expressed by "($\forall$ set: client_set)(set.actual_load $\leq$ prefer_load - offset)", then, setup a function that maps every client_set to a new region_id as denoted by "newid=(λ s: client_set $\bullet$ region_id)", finally, send out the request "new" to the *system_service_layer* which will further transfer it as "add_new" request to the *global_server* and make each clients of the client set become the client of that new region_id.

Due to the space limitation of this paper, we won't be able to specify all the components in detail, but in order to make clear the connections between regional_server and global_server, we will specify part of the architectural behavior of the system_service_layer, which will be promoted as the architectural connections of the regional_server to the outside as implicated by the "Initialize" action of the regional_server_frame. The actions of system_service_layer will mostly consist of the operations that transfer import of *controller* and *global_server* to export of *global_server* and *controller*, or vice versa. For example, after receiving "new"("update_info") from *controller* in the IMPORT part, transfer it to the *global_server* as "add_new"("info_update") request through EXPORT part. Thus, match between *global_server* and *regional_server* can be insured.

OBJECT: system_service_layer
A PART OF: regional_server_frame
PARTS: communication_manager
device_manager
INTERNAL BEHAVIOR:
PRECOND: ···
ACTION: ···
EXCEPTION: ···
⋮
ARCHITECTURAL BEHAVIOR:
IMPORT PART:
IMPORT: query_result(rcd)
PROVIDER: global_server

IMPORT: new(id, actual, prefer)

PROVIDER: controller

⋮

EXPORT PART:

EXPORT: query_info(region_id)
RECEIVER: global_server

EXPORT: info_update(region_id, actual_load, prefer_load)
RECEIVER: global_server

EXPORT: add_new(region_id, actual_load, prefer_load)
RECEIVER: global_server

⋮

Figure 13. Partial specification of the *system_service_layer*

6 Conclusion

In this paper, we first presented an overall incentive for architecture description languages — the widely used box-and-line diagram and nature language like descriptions in designing software frameworks is too "flexible" to convey rich informations about system design. As a consequence of this, we designed our OOADL aimed at six goals: support for O-O approach, support for hierarchical refinement, support for broad class of styles, support for dynamic architecture construction, support for analysis and support for exception handling. We have given out the outlook of our OOADL in this paper and used a small example to illustate the usage of it. Further more, as an important feature of our OOADL, we explained the default architecture style and its purpose in detail, default styles extensively reflected the objected-oriented features: it brings to us the extensibility(override part of the default specifications or refine the default ones) and reusability(multiple instances can be created and used in system specification) via instanciation and inheritance of default styles. To see the practical applicability of this language, considering the close relationship between software architectures and multimedia system design, we used a sample multimedia system as a comprehensive example in this paper to demonstrate the power of our OOADL.

APPENDIX A

Formal Grammar of OOADL

< OOADL > ::=
    OBJECT: < component-name >
        < oo-features >
        < composition >
        INTERNAL BEHAVIOR: < behavior >
        ARCHITECTURAL BEHAVIOR: < I/O >

< component-name > ::=
    < identifier >

< oo-features > ::=
    < Abstract-Relation-Type > : < object-name > | ϵ

< Abstract-Relation-Type > ::=
    A KIND OF | **A PART OF** | **AN INSTANCE OF**

< object-name > ::= identifier | < OOADL >

< composition > ::=
    PARTS: (< object-name >)$^+$ | ϵ

< behavior > ::=
    (< action-triplet >)$^+$ | nil

< action-triplet > ::=
    PRECOND: < Z-specification >
    ACTION: < Z-specification >
    EXCEPTION: < Z-specification >

< I/O > ::=
    < import > | < export > | < import > < export > | ϵ

< import > ::=
    IMPORT PART: (< import-pair >)$^+$

< import-pair > ::=
    IMPORT: < message >
    PROVIDER: < object-name >

< message > ::=
    < identifier > (< argument >*)

< argument > ::=

< identifier >

< export > ::=
EXPORT PART: (< export-pair >)$^{+}$

< export-pair > ::=
EXPORT: < message >
RECEIVER: < object-name >

APPENDIX B — Default Architecture Styles

Constraints of Object-Oriented System:

1. The only components of this style are objects;
2. Each object maintains its data members (ADTs) and a set of operations on these data members (methods), it also provides an interface (set of messages) for outside;
3. The messages appear in the interface should be a subset of available methods of the object;
4. There's no "useless" data members, i.e., every data member must appear in some methods;
5. Objects communicate with each other through message passing, only messages appeared in the interface will be accepted, (we assume undefined messages sent to an object will be simply neglected without any action).

Z Specification of Object-Oriented Style:

[DATA NAME]

ADT: $\mathbf{P}$ DATA

Object

$$
\begin{array}{l}
\text{data\_member: } \mathbf{P}\ \text{ADT} \\
\text{method: } \mathbf{P}\ \text{NAME} \times \text{ADT} \rightarrow \text{ADT} \\
\text{interface: } \mathbf{P}\ \text{NAME} \times \text{ADT} \\
\hline
\text{data\_member} \neq \phi \Rightarrow \text{method} \neq \phi \\
\forall\ \text{x: data\_member} \bullet \exists\ \text{m: method} \bullet \text{x} \in \mathit{dom}\ \text{m} \vee \text{x} \in \mathit{ran}\ \text{m} \\
\text{interface} \subseteq \mathit{dom}\ \text{method}
\end{array}
$$

method_execution[name, arg]

$$
\begin{array}{l}
\Delta\ \text{Object} \\
\hline
\text{method'} = \text{method} \\
\text{interface'} = \text{interface} \\
\exists\ \text{m: method} \mid (\text{name, arg}) \in \mathit{dom}\ \text{m} \bullet \\
\theta\ \text{data\_member'} = \theta((\text{data\_member} \setminus (\mathit{ran}\ \text{m} \cap \text{data\_member})) \\
\qquad\qquad \cup\ (\text{m(name, arg)} \rhd \text{data\_member}))
\end{array}
$$

Object_behavior

$$
\begin{array}{l}
\text{obj: Object} \\
\text{msg: Object} \times (\text{NAME} \times \text{ADT}) \rightarrow \text{ADT} \\
\hline
((\text{receiver, (mth\_name, arg)}) \in \text{msg}) \wedge (\text{receiver=obj}) \\
\Rightarrow (\text{mth\_name, arg}) \in \text{obj.interface} \bullet \text{method\_execution(mth\_name, arg)}
\end{array}
$$

Constraints of Layered System:

1. The entire system is composed of layers and communications between layers;
2. #communication = #layer - 1;
3. Bi-direction communication between layers;
4. Layers are strictly sequenced;
5. One layer has at most two communications inside one layered system;
6. Each layered system can have one connection(I/O) to outside at its top, or one connection(I/O) to outside at its bottom, or both, or none, such connection should be considered as the entire system's communication to outside instead of the highest or lowest layer's behavior, i.e., though such connections attached to the highest or lowest layer topologically, they are indeed promoted to the entire layered system level;
7. Hierarchical structure is also available, i.e., each layer can take part in more than one layered system, but within each system, constraint #2 should still be conformed to.

Z Specification of Layered System:

[FUNCTIONAL_COMPONENT]

layered_component

layer: **F** FUNCTIONAL_COMPONENT
comm: **F** FUNCTIONAL_COMPONENT $\leftrightarrow$ FUNCTIONAL_COMPONENT
top, bottom: FUNCTIONAL_COMPONENT $\leftrightarrow$ PLATFORM

---

dom comm = layer
ran comm = layer
#comm = #layer - 1

layered_behavior

l_sys: layered_component
order: FUNCTIONAL_COMPONENT $\rightarrow N_1$

$\forall$ l: l_sys.layer $\bullet$ order($first$ l_sys.top) $\geq$ order(l) $\wedge$
order($first$ l_sys.bottom) $\leq$ order(l)

$\forall l_1, l_2$: l_sys.layer $\bullet$ order(l_1)=order(l_2) $\Rightarrow l_1 = l_2$
$\forall$ c: l_sys.comm $\bullet$ order($first$ c)=order($second$ c)+1 $\vee$
order($first$ bottom)=0 $\wedge$ order($first$ top)=#l_sys.layer-1
$\forall$ l: l_sys.layer $\bullet$ bag dom(l_sys.comm) $\sharp$ l = bag ran(l_sys.comm) $\sharp$ l = 1

Default operations of layered system such as add_as_layer[l, pos] and retract[l, pos] can also be defined using Z specification (which are omitted here). And thus, layered system can be specified as:

LAYERED_SYSTEM

layered_component
layered_behavior

---

add_as_layer[l,pos]
retract[l, pos]

APPENDIX C — The complete OOADL specification of the sample multimedia system.

• The specification of the top level system

OBJECT: multimedia_system
A KIND OF: CLIENT_SERVER_SYSTEM
PARTS: global_server
regional_server
client

INTERNAL BEHAVIOR:
PRECOND: multimedia_system = ϕ
ACTION:

Initialize

global: CLIENT_SERVER_SYSTEM
regional: **F** client_server_component
global_server: PLATFORM
regional_server, client: **F** PLATFORM

---

global_server = global.server
regional_server $\subseteq$ global.client
$\forall$ rs: regional_server | $\exists_1$ r: regional • rs=r.server
$\forall$ rs1,rs2: regional_server | $\exists$ r: regional $\wedge$ (rs1=r.server $\wedge$ rs2=r.server) • rs1=rs2
$\forall$ c: client | $\exists_1$ r: regional • c $\in$ r.client
regional_server=ϕ, client=ϕ

EXCEPTION: nil

ARCHITECTURAL BEHAVIOR: nil

• Specification of the *global_server*

OBJECT: global_server
A PART OF: multimedia system
PARTS: nil

INTERNAL BEHAVIOR:
PRECOND: low_loading: { $\exists$ s: region_id | get_actual(s) < get_prefer(s)-offset }
ACTION:

Optimize

Δ multimedia_system
Δ Server_Table

$\exists$ s_x: region_id | $s_x \neq$ s $\wedge$
get_actual(s_x)+get_actual(s) $<$ get_prefer(s)-offset
$\bullet$($\forall$ c: s_x.client) s.register_as_client[c] $\wedge$
multimedia_system.retract[s_x] $\wedge$
get_list'(s)=get_list(s) $\cup$ get_list(s_x) $\wedge$
get_prefer'(s)=get_prefer(s) $\wedge$
get_actual'(s)=get_actual(s)+get_actual(s_x) $\wedge$
record' = record $\setminus$ {s_x, get_list(s_x), get_prefer(s_x), get_actual(s_x)}

EXCEPTION:

unsatisfiable_optimization

Δ multimedia_system
Δ Server_Table

$\forall$ s_x: regional_server | $s_x \neq$ s $\wedge$
get_actual(s)+get_actual(s_x) $\geq$ get_prefer(s)-offset
$\bullet$ multimedia_system'=multimedia_system $\wedge$
Server_Table' =Server_Table

PRECON:

update_request

$\exists$ in: IMPORT_PART | in.IMPORT=info_update(region_id, actual_load, prefer_load) $\wedge$
content(in.IMPORT) $\neq \phi \wedge$
region_id $\in$ Server_Table.record.region_id

ACTION:

Update[region_id, actual_load, prefer_load]

Δ Global_Table

---

get_list'(region_id) = get_list(region_id)
get_perfer'(region_id) = prefer_load
get_actual'(region_id) = actual_load

EXCEPTION: update_before_register
region_id $\notin$ Server_Table.record.region_id $\bullet$ Add_New[region_id, actual_load, prefer_load]

PRECON:

register_request

$\exists$ in: IMPORT_PART | in.IMPORT=add_new(region_id, actual_load, prefer_load) $\wedge$
content(in.IMPORT) $\neq \phi \wedge$
region_id $\notin$ Server_Table.record.region_id

ACTION:

Add_New[region_id, actual_load, prefer_load]

Δ Global_Table

record'=record $\cup$ {region_id, ϕ, prefer_load, actual_load}
get_actual'=get_actual $\cup$ {region_id $\mapsto$ actual_load}
get_prefer'=get_prefer $\cup$ {region_id $\mapsto$ prefer_load}

EXCEPTION: already_exist
region_id $\in$ Server_Table.record.region_id $\bullet$ Update[region_id, actual_load, prefer_load]

PRECON:

query_request

$\exists$ in: IMPORT_PART | in.IMPORT=query_info(region_id) $\wedge$
content(in.IMPORT) $\neq \phi \wedge$
region_id $\in$ Server_Table.record.region_id

ACTION:

Query_Info

Δ global_server

$\exists$ out: EXPORT_PART | out.EXPORT=query_result(rcd)
$\bullet$ $\exists$ r $\in$ Server_Table.record $\wedge$ r.region_id=region_id
$\Rightarrow$ out'.EXPORT=query_result(r)

EXCEPTION: out.EXPORT=query_result(nil)

ARCHITECTURAL BEHAVIOR:

IMPORT PART:

IMPORT: info_update(region_id, actual_load, prefer_load)
PROVIDER: regional_server

IMPORT: add_new(region_id, actual_load, prefer_load)
PROVIDER: regional_server

IMPORT: query_info(region_id)
PROVIDER: regional_server

EXPORT PART:
EXPORT: query_result(rcd)
RECEIVER: regional_server

The regional_server_frame specification

OBJECT: regional_server_frame
A PART OF: multimedia_system
A KIND OF: LAYERED_SYSTEM
PARTS: controller
system_service_layer
multimedia_service_layer
INTERNAL BEHAVIOR:
PRECOND: ϕ
ACTION:

Initialize

l_cs, l_ms, l_cm: LAYERED_SYSTEM

---

controller, multimedia_service_layer ∈ l_cm.layer ∧
order(controller)=1 ∧ order(multimedia_service_layer)=0 ∧
top=nil ∧ bottom=nil

controller, system_service_layer ∈ l_cs.layer ∧
order(controller)=1 ∧ order(system_service_layer)=0 ∧
top=nil ∧ system_service_layer=*first* bottom ∧
∀ port: {system_service_layer.IMPORT ∪ system_service_layer.EXPORT} |
port ∈ {query_info(region_id), info_update(region_id, actual_load, prefer_load),
query_result(rcd), add_new(region_id, actual_load, prefer_load)}
• port *promoted*

multimedia_service_layer, system_service_layer ∈ l_ms.layer ∧
order(multimedia_service_layer)=1 ∧ order(system_service_layer)=0 ∧
top=nil ∧ bottom=nil

EXCEPTION: nil

ARCHITECTURAL BEHAVIOR: nil

The specification of the controller

OBJECT: controller

A PART OF: regional_server_frame
PARTS: nil
INTERNAL BEHAVIOR:

PRECOND:

$\exists$ in: IMPORT_PART, out: EXPORT_PART |
in.IMPORT=get_query_result(rcd) $\wedge$ out.EXPORT=send_query_info(id)
$\wedge$ in.IMPORT $\circ$ out.EXPORT $\Rightarrow$ rcd.actual_load $\neq$ actual_load

ACTION:

Δ controller

$\exists$ out: EXPORT_PART
| out.EXPORT=update_info(id, actual, prefer)
• out'.EXPORT=update_info(region_id, actual_load, prefer_load)

EXCEPTION: nil

PRECOND: overloaded
actual_load > prefer_load

ACTION:

Adaption_1

Δ controller

$\exists$ client_set: **F** {$\exists$ r: multimedia_system.regional
| regional_server=r.server • c $\in$ r.client}
| ($\forall$ set: client_set) (set.actual_load $\leq$ prefer_load-offset)
• ((newid=(λ s: client_set • region_id))
| $\exists$ out $\in$ EXPORT_PART $\wedge$ out.EXPORT=new(region_id, actual_load, prefer_load)
• out'.EXPORT=new(newid(set), set.actual_load, prefer_load) $\wedge$
($\forall$ c: set) (c $\in$ newid(set).client))

EXCEPTION: nil

ARCHITECTURAL BEHAVIOR:

IMPORT PART:

IMPORT: get_query_result(rcd)
PROVIDER: system_service_layer

EXPORT PART:

EXPORT: send_query_info(id)
RECEIVER: system_service_layer

EXPORT: update_info(id, actual, prefer)
RECEIVER: system_service_layer

EXPORT: new(id, actual, prefer)
RECEIVER: system_service_layer

References

[1] Robert Allen and David Garlan: "The Wright Architectural Specification Language", Carnegie Mellon University, SCS Technical Report, to appear.

[2] Paul C. Clements: "A Survey of Architecture Description Languages", *Eighth International Workshop on Software Specification and Design*, Paderborn, Germany, March 1996.

[3] David Garlan, Robert Monroe and David Wile: "ACME: An Architectural Interchange Language", Technical Report CMU-CS-95-219, School of Computer Science, Carnegie Mellon University.

[4] David Garlan: "An Introduction to the Aesop System", July 1995, http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps

[5] Kuang Xu: "Comparative Study of Deadlock-free Property on Software Architectures Specified Using OOADL and WRIGHT", master thesis, EECS department, University of Illinois at Chicago, 1997.

[6] David C. Luckham and James Vera: "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, Vol.21, No.9, September 1995.

[7] J.Magee and J.Kramer: "Dynamic Structure in Software Architectures", *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp3-pp14, San Francisco, CA, October 1996.

[8] N.Medvidovic, R.N.Taylor and E.J.Whitehead,Jr: "Formal Modeling of Software Architectures at Multiple Levels of Abstraction", *Proceedings of the California Software Symposium 1996*, pp28-pp40, Los Angeles, CA, April 1996.

[9] Mary Shaw and David Garlan: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall Publishing, 1996.

[10] M.Shaw, R.DeLine, D.Klein, T.Ross, D.Young and G.Zelesnik: "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, April 1995.

[11] J.M.Spivey: *The Z Notation: A Reference Manual, Second Edition*, Prentice Hall International (UK) Ltd, 1992.

[12] Jeffrey J.P. Tsai and Thomas J. Welgert: *Knowledge-Based Software Development For Real-Time Distributed Systems*, Series on Software Engineering and Knowledge Enginerring - Vol. 1.

[13] Jeffrey J.P. Tsai, "Knowledge-Based Software Architecture," IEEE Trans. on Knowledge and Data Enginerring. Vol. 11, No.1, Jan. 1999.

[14] Mary Shaw, R. DeLine, D. Klein, T. Ross, D. Young and G.Zelesnik: "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, April 1995.