16

An experience modelling telecommunications systems using ODP-DLcomp

B. Stepien, K. Farooqui, L. Logrippo
Telecommunications Software Engineering Research Group
Department of Computer Science, University of Ottawa
Ottawa, Ont. Canada, K1N 6N5
(bernard | farooqui | luigi)@csi.uottawa.ca

Abstract

The ODP-DLcomp language is intended to describe systems in the ODP computational model. It is object-oriented, implementation-independent and its formal semantics are based on LOTOS, thus it is a Formal Description Technique. It supports specification of interface and object templates, with appropriate creation and binding operations. An application of the language is demonstrated using the Plain Old Telephone System example.

Keywords

Open Distributed Processing, OO Languages, Specification, Telephony

1 INTRODUCTION

ODP-DLcomp was designed and implemented at GMD-FOKUS by Frank Koch [Koch94]. The first author participated in the group at the time.

The motivation behind this new language was to offer a more readable and compact alternative to the LOTOS description of the ODP computational model [Vog93] but also to offer a bridge between Formal Description Techniques and Object Oriented implementation languages.

In this paper, we show that ODP-DLcomp includes all the essential concepts for system definition in the ODP computational model. It is high-level and object-based. One of its important characteristics is that its semantics are defined in terms of the formal language LOTOS[BoBr87]. In fact, the definition is constructive and a prototype compiler from ODP-DLcomp into LOTOS is available. Thus, the language has all the essential characteristics required of a Formal Description Technique for ODP. Because of their capability of being compiled into a language that is at least partially executable, specifications written in ODP-DLcomp constitute a *prototype* of the system being specified. Activities such as validation and test case generation, possible from LOTOS specifications, therefore are also possible from ODP-DLcomp specifications.

This language is somewhat comparable to TINA ODL [TINAC94], however the latter is not formal, and does not offer the capability of behavior specification.

A brief introduction to the language is given, followed by an example of its use for the description of a Plain Old Telephone System.

E. Najm et al. (eds.), *Formal Methods for Open Object-based Distributed Systems*

2 BASIC CONCEPTS OF ODP COMPUTATIONAL MODEL

The ODP computational *viewpoint* provides the concepts needed to explain how services can be programmed in a form suitable for distribution [FLM95]. This viewpoint focuses on the organization of applications in architecturally conformant ways rather than on the mechanisms used to distribute or support the applications in the system. The ODP computational *model* is the abstract model to express the concepts of the computational viewpoint. It is a framework for describing the structure, specification and execution of the (components of the) distributed application on the distributed computing platform. It describes the coarse-grained structure of an application, i.e., the application components and their interaction at an abstract, system independent level. Each coarse-grained entity of a distributed application is represented by an object, called *computational object*, with a (set of) well defined interface(s), called *computational interface(s)*. The computational model defines *what* is required rather than *how* it is provided, the latter being the responsibility of the engineering model [FaLo95].

The basic elements of the computational model are: *computational object*, *computational interface*, *operation invocation* at a computational interface, *activities* that occur within a computational object, *environment constraints* on operation invocation, etc.

3 SPECIFYING TELEPHONE SYSTEMS USING ODP-DLCOMP

As an example of use of the language, the well known POTS (Plain Old Telephone System) [SL93] application is presented. First, the general architecture is discussed and it is shown how the concepts of objects and interfaces apply to this example. The flow of operation invocations is then discussed.

3.1 General architecture

The system is composed of four kinds of objects (Fig 1):the network object, the administrator object, the phone objects, the user objects. These objects communicate through five kinds of interfaces:

- the phone control interface (*phone_ctrl_if*) where user requests take place.
- the network control interface (*network_ctrl_if*) where phone requests take place.
- the administration control interface (*admin_ctrl_if*) where administration requests take place.
- the phone network control interface (*ph_ntw_ctrl_if*) where network requests to a phone take place.
- the network administration control interface (*network_admin_ctrl_if*) where administration requests to the network take place.

Once the objects and their interfaces are identified, we need to describe the initial configuration of the system and the dynamic object creations that take place.

Initially, the system is composed of three known categories of components. The network, the administrator and the users need to exist and need to know how to communicate. The phones, however, do not exist. They need to be installed by the administrator on requests from the users. At this time, the users will consult the administrator to associate themselves with a particular instance of a phone using the directory function.

The full specification of the POTS system initialization is as follows:

```
system_initialization
    create_object network()
        return network_admin_control_if_id:ident, network_control_if_id1: ident,
               network_control_if_id2: ident ;
    create_object administrator(network_admin_control_if_id,network_control_if_id1,
                                network_control_if_id2)
        return admin_control_if_id: ident ;
    create_object user( admin_control_if_id )
        return client1_phone_if_id: ident ;
    create_object user( admin_control_if_id )
        return client2_phone_if_id: ident
```

3.2 Specification of the interfaces

The operations that can be performed at each interface, the semantics of their various terminations, and the orderings of these operations at this interface need to be identified. These operations are summarized in Fig.1. Each interface is represented by a box with a header containing the name of the interface and a body containing the list of available operations.

The following example describes the phone control interface where users invoke operations as requests to a given instance of a phone.

```
op_if_template phone_control_interface( )
    operation offHook( )
        termination tone_obtained( )
        termination tone_absent()
    operation onHook( )
        termination on_hook_performed( )
    ...
    behaviour
        choice(seq(offHook,dial,talk,onHook), seq(offHook,onHook))
end
```

Most operations are described as having only one termination. The *dial()* operation is an exception that illustrates the principle of multiple terminations. When a user dials a number, two things can happen. Either this user gets connected or she hears a busy signal. Consequently the dial operation has been specified as having two terminations: *connected*() and *busy*().

Another aspect of the termination statement is the parameter list as in the *createPhone()* operation of the administration control interface (*admin_ctrl_if*).

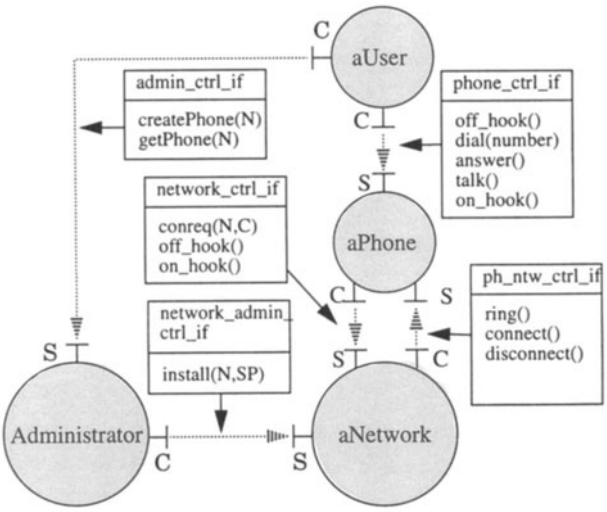

Figure 1 General architecture and interface model.

```
operation createPhone( aNumber: number )
    termination phone_created( phone_ctrl_if_id:ident)
```

The above operation returns the identifier *phone_ctrl_if_id* of the server interface of the newly created phone (denoted by S on Fig. 1). The user needs to perform an **associate** statement using this identifier in order to be able to communicate with this phone on this interface.

3.3 Specification of objects

The specification of objects consists in instantiating interfaces and describing the behavior for each operation at these interfaces.

Each object needs a number of attributes and instances of interfaces. The user, phone and administrator objects have a fixed number of attributes and interfaces as shown in Fig. 1. However the network, in order to communicate with a variable number of phones, needs a variable number of interfaces as shown in Fig. 2. For the purpose of this example, we have used two instances of phones with the corresponding interfaces.

The main role of the network is to perform connections. For this purpose, when a new connection request comes in, the network needs to remember where it came from, where it is going to and, since many operations are required to complete the connection, a connection state attribute. These attributes will contain references to interface instances.

The following is an example of the network attributes and interfaces requirements:

```
object_template network( )
    ...
    initialization
    ...
    instantiate phone_network_control_interface( ) client
        return phone_network_ctrl_if_id1 : ident ;
    instantiate phone_network_control_interface( ) client
        return phone_network_ctrl_if_id2 : ident ;
    instantiate network_control_interface( ) server
        return network_ctrl_if_id1 : ident ;
    instantiate network_control_interface( ) server
        return network_ctrl_if_id2 : ident ;
    instantiate network_admin_control_interface( ) server
        return network_admin_ctrl_if_id1 : ident
```

In the next sections, specific aspects of interface association mechanism of ODP-DLcomp and operation invocations are addressed.

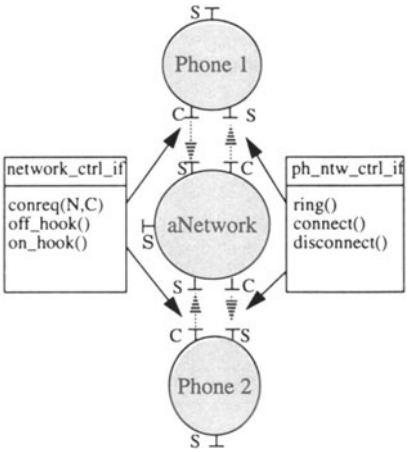

Figure 2 Multiple instances of interfaces of the network object.

The interface association mechanism

Static association

As already mentioned, at system initialization time, only the network, the administrator and the users exist. The association of their interfaces is straightforward since the objects involved

play only one role at a time in pairs, i.e. the user is always a client to the administrator server and the administrator is always a client to the network server. This simple relationship can be described as part of the object initializations. For example the user to administrator interface is described by the following objects:

```
object_template administrator( server_network_admin_cntrl_if_id,
                    server_network_cntrl_if_id1, server_network_cntrl_if_id2: ident)
    ...
    initialization
        instantiate admin_control_interface( ) server
                return admin_ctrl_if_id : ident ;
    ...
end
object_template user(server_admin_cntrl_if_id: ident )
    ...
    initialization
        instantiate admin_control_interface( ) client
                return admin_ctrl_if_id : ident ;
        associate admin_ctrl_if_id with server_admin_cntrl_if_id
    ...
end
```

The glue between these two object template instantiations is achieved by the following system initialization statements. One observes that the interface identifier *admin_control_if_id* returned from the instantiation of the administrator object is a parameter of the user object constructor:

```
system_initialization
    create_object administrator(network_admin_control_if_id,network_control_if_id1,
                                network_control_if_id2)
        return admin_control_if_id: ident ;
    create_object user( admin_control_if_id )
        return client1_phone_if_id: ident ;
```

Note that the identifier *admin_control_if_id* is given as returned by the created object *administrator*, and therefore it actualizes *admin_ctrl_if_id* defined in the template of the object. This identifier then is used as formal parameter of the created object *user*. As such it actualizes *server_admin_ctrl_if_id*.

Dynamic association

However, the associations between users and phones or phones and network are dynamic, i.e. they occur as results of operation invocation. The following excerpts of the user, administrator and phone objects will illustrate this concept.

```
object_template user(server_admin_cntrl_if_id: ident )
    ...
    initialization
        instantiate phone_control_interface( ) client
                return phone_ctrl_if_id : ident
        ...
    behavior
        ...
        CLIENT_IFACE admin_control_interface :
        {
            createPhone :
                term_case phoneCreated( aPhone_ctrl_if_id:ident ) :
                {
                    associate phone_ctrl_if_id with aPhone_ctrl_if_id
    ...
end

object_template administrator( server_network_admin_ctrl_if_id, server_network_cntrl_if_id1)
    ...
    initialization
        ...
        instantiate admin_control_interface( ) server
```

```
                return admin_ctrl_if_id : ident ;
            ...
        behaviour
            SERVER_IFACE admin_control_interface :
                {
                    createPhone(aNumber: number )
                    {
                        create_object phone( aNumber, server_network_cntrl_if_id1 )
                            return phone_control_if_id: ident
end
object_template phone( number: number, server_network_if_id: ident)
    ...
    initialization
        instantiate phone_control_interface( ) server
            return phone_ctrl_if_id : ident;
        ...
end
```

When a user requests a phone to be created by the administrator via the *createPhone* operation, the administrator creates the object and obtains in return an identifier *phone_control_if_id*. This identifier is created as a result of the instantiation of the *phone_control_interface* that returns the identifier *phone_ctrl_if_id*. Finally when the operation *createPhone* terminates successfully, it returns the result *phone_created* that carries the identifier *phone_control_if_id*. The latter is then used in the user object (*term_case phone_created(...)*) to perform the association between the client and server instances of the *phone_control_interface*. The associations between the instances of the network control interface and the phone network control interface, that were also unknown to both the phone and the network before the phone object was created, are not shown.

Specifying the behavior as operation invocations

Busy signal specification

In POTS a phone is busy when its state is not idle. *idle* is the initial state, when the phone object is created or when an on hook operation has been invoked.In order to specify the busy signal a phone object needs a state attribute and two possible terminations of its *ring()* operation. The state variable is updated as different operations are performed on this object. For example, an off hook operation changes the state variable from *idle* to *off_hook*. This is shown in the following example:

```
object_template phone( number: number, server_network_if_id: ident)
    ...
    initialization
        let state : state_sort = idle ;
        ...
    behavior
        SERVER_IFACE phone_network_control_interface :
        {
            ring( )
            {
                if state eq idle then
                {
                    state := rung ;
                    terminate rung()
                }
                else
                    terminate busy()
            ...
end
```

The ring operation has an *if-then-else* statement that controls the nature of the termination value *rung()* or *busy()*. In the first case the phone object's state is changed to *rung* and in the second case it remains unchanged, because this phone is already engaged in another

connection hence its state must stay unchanged. On the network client side the termination case statement is used to control the next transition. The above termination cases are driven by the network's behavior where the return values are determined by the outcome of a ring() operation invocation:

```
invoke called_phone_network_ctrl_if_id->ring()
term_case rung() :
{
    terminate ring_back()
}
term_case busy() :
{
    terminate busy_tone()
}
```

Call termination specification

The requirement is that when any one of the phones involved in a connection goes on hook, the network should stop performing the normal sequence of operations and start disconnecting the phones involved.

The phone objects have two different behaviors depending on whether they are the call termination initiator or the passive disconnected party. In the first case the sequence of actions is to place the phone on hook and invoke a disconnection to the network, in the second case it is to receive a disconnection from the network and then perform an on hook. Consequently, an *onHook* operation at the phone control interface (invoked by a user) has to be processed differently in the two cases. In the first case, the phone invokes a disconnection to the network only if it was not already disconnected. In this case it receives back a *disconnected()* termination from the network, and the state is returned to *idle*. But if a phone becomes disconnected due to a passive termination it goes back directly to *idle* state.

```
object_template phone( number: number, server_network_if_id: ident)
...
initialization
    let state : state_sort = idle ;
    let dialed_number: number = nil ;
    ...
behavior
    SERVER_IFACE phone_control_interface :
    {
        onHook()
        {
                if state ne disconnected then
                {
                    invoke network_ctrl_if_id->onHook()
                        term_case disconnected() :
                        {
                            state := idle
                        }
                }
                else
                    state := idle ;
                terminate on_hook_performed()
        }
    ...
    };
```

4 CONCLUSIONS

It has been shown that the language ODP-DLcomp provides support for the formal specification of Open Distributed Processing concepts of object template, interface template (including multiple terminations), operation signature, object behavior, and others. These

capabilities were demonstrated by using a Plain Old Telephone System example. One of the main assets of the language is the visibility of interfaces which make it possible to specify the links between objects. On the other hand, the language's weakness resides in the expression of behavior. The C-like operation invocation seems to be an ad hoc solution, while it appears that the TINA-C recommendation of specifying behavior using LOTOS or SDL is a better solution. Further work will deal with improving the language constructs, given the insight provided by this experience.

5 ACKNOWLEDGMENTS

We would like to thank Jan de Meer of GMD-FOKUS for giving us the opportunity to work on the implementation of this language, especially on the exploration of visual animation and programming. This research was partially supported by a grant from Motorola ARRC and a Going Global grant from the Ministry of External Affairs.

6 REFERENCES

[BoBr87] Bolognesi, B., and Brinksma, E. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59.

[Booch94] Booch, G. Object Oriented Design with Applications, Benjamin/Cummings, 1994.

[FaLo95] Farooqui, K., and Logrippo, L. Architecture for Open Distributed Software Systems. In: A. Zomaya (ed.) Parallel And Distributed Computing Handbook. McGraw-Hill, 1996. Chapter 11, 303-329.

[FLM95] Farooqui, K., Logrippo, L., deMeer, J. The ISO Reference Model for Open Distributed Processing: an introduction in Computer Networks and ISDN Systems 27 (1995) 1215-1229

[Koch94] Koch. F. Spezifizierung offener Verteilter Systeme aus Sicht des ODP Computational Viewpoint, Gesellschaft fuer Mathematik und Datenverarbeitung, GMD-Studien Nr. 243, October 1994.

[SL93] Stepien, B., and Logrippo, L. Status-Oriented Telephone Service Specification. In: T.Rus and C.Rattray (eds.) *Theories and Experiences for Real-Time System Development*. AMAST Series in Computing, Vol. 2, World Scientific, 1994, 265-286.

[TINAC94] Kitson, B., Leydekkers, P., Mercouroff, N., Ruano, F, et al. TINA Object Definition Language (TINA-ODL) Manual, TINA Consortium, 1995.

[Vog93] A.Vogel On ODP'sArchitectural Semantics using LOTOS: In: J.de Meer, B. Mahr, O. Spaniol(Editors): Proceedings of the International Conference on Open Distributed Processing. Berlin, September 1993