An experience modelling telecommunications systems using ODP-DLcomp

Bernard Stepien, Kazi Farooqui, Luigi 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

0. 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][LoFH92]. 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.

1. 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 com-

putational *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.

2. An overview of the ODP-DLcomp language

The language supports the two following concepts of the ODP computational interface template:

- operation signature: the language allows the specification of multiple terminations along with the operation name.
- behavior: the language supports the specification of the behavior exhibited at the interfaces. It allows the specification of possible orderings of operations by using operators able to express sequence, choice and interleaving.

An ODP-DLcomp specification is made up of four components. We present its constructs via concrete examples. The following example shows the general structure of an ODP system specification. The keywords used for each section are as follows: **op_if_template** for the description of operational-interface templates, and **object_template** and **system_initalization** which are self-explanatory.

```
odp_system POTS_service ::=

        library Boolean, Integer, Set endlib

        type Integer is
            sorts Int
            opns
                0 :-> Int
        endtype
        ...
        op_if_template admin_control_interface( )
            ...
        end
        ...
        object_template administrator( server_network_cntrl_if_id: ident, ...)
            ...
        end
        ...

        system_initialization
            ...
end_spec
```

2.1 Specification of abstract data types

Abstract Data Types are currently described using ACT ONE as in LOTOS. A tutorial on this language can be found in [MRV91]. It is planned to include ASN.1 in the future.

2.2 Specification of interface template

This template defines the operations that can be performed over the interface with the number and type of formal parameters (sometimes called stimuli), and the possible terminations.

Each interface template has a name. For each operation we specify a name, the names and types of formal parameters and of terminations. An interface template can then be used by an object. It is the responsibility of the object to determine the role of each interface (client or server) and the activities included in an operation.

```
op_if_template admin_control_interface( )

        operation createPhone( aNumber: number )
            termination phone_created(phone_ctrl_if_id )
            termination error( err_msg: CrError )

        operation getPhone( aNumber: number )
            termination theNumberIs( aPhone: ident )

        behaviour

            choice(create_phone, getPhone)
end
```

2.3 Specification of an object template

There are four main sections in an object template. The following example illustrates the syntax to describe these main concepts.

```
object_template administrator( server_network_admin_cntrl_if_id, server_network_cntrl_if_id1,
                                network_control_if_id_2: ident)

        includ_if_tpl : admin_control_interface,
                        ...
        refer_object_tpl phone

        initialization

            let entry_1: phone_entry = nil ;
            ...
            instantiate admin_control_interface( ) server
                            return admin_ctrl_if_id : ident ;

            instantiate network_admin_control_interface( ) client
                            return network_admin_ctrl_if_id : ident ;

            associate network_admin_ctrl_if_id with server_network_cntrl_if_id
            ...
        behaviour

            ...
end
```

We now briefly review the components of this object template.First, the list of all interface templates that will be used by the object is declared.

```
includ_if_tpl : admin_control_interface,
                ...
```

Here *admin_control_interface* was defined in the interface template definitions section. Then we need to indicate the objects that will be used. As part of its activities, an object can instantiate other objects. This template defines which objects can be instantiated.

```
refer_object_tpl phone
```

An object needs to be initialized like in a *constructor* operation of an object as known in object-oriented literature [Booch94]. There are two kinds of elements that need to be declared and instantiated.

- Internal variables that correspond to attributes in traditional object-oriented terminology are declared here and assigned their initial values.
- Instances of interface templates are specified along with their roles within this object (client or server). There can be many instances of a given type of interface. For example a network object needs to communicate to various phones through individual instances of the same *phone_network* interface template. Each instance is identified by a unique identifier, which is obtained through the *return* construct.

```
initialization

    let entry_1: phone_entry = nil ;
    ...
    instantiate admin_control_interface( ) server
                    return admin_ctrl_if_id : ident ;

    instantiate network_admin_control_interface( ) client
                    return network_admin_ctrl_if_id : ident ;

    associate network_admin_ctrl_if_id with server_network_cntrl_if_id
    ...
```

Finally the behavior of an object is specified for each of its interfaces. The keywords **SERVER_IFACE** or **CLIENT_IFACE** are used to specify the role of the individual interface. For each interface template there is an interface description block that contains individual operations supported by the interface. The behavior of operation signatures is described as a block of activities.

The most common activities are:

- object creation
- operation invocation on an instance of another interface
- instantiation of new interfaces
- object or interface deletion

An interface is used symmetrically by a client and server object. The activity of an operation is described in the server object while the evaluation of the termination is described in the client object.

```
behavior

    SERVER_IFACE admin_control_interface :
        {
            createPhone(aNumber: number )
                {
```

```
                create_object phone(aNumber, server_network_cntrl_if_id)
                    return phone_control_if_id: ident ;
                    ...
            }
        ...
    } ;

CLIENT_IFACE network_admin_control_interface:
    {
        install:
            term_case phone_installed( ) :
            {
                state_phone := installed ;
            }
        ...
    } ;
...
end
```

Summary of activities specification

Interface instantiation

```
instantiate <interfaceTemplateName> <role>
        return <interfaceIdentifier>: ident
```

The above construct instantiates an interface and returns an identifier.

Object creation

```
create_object network()
    return network_admin_control_if_id, network_control_if_id_1,
            network_control_if_id_2: ident ;
```

When an object of a given class is created, a list of instantiated interface identifiers is automatically created. These are of built-in type *ident* and can be referenced by interface instantiations *association* or operation invocation statements.

Object interfaces binding

```
associate network_ctrl_if_id with server_network_cntrl_if_id
```

This statement binds two instances of the same class of interface template.

Operation invocation

```
invoke called_phone_network_ctrl_if_id->ring()
    term_case rung() :
    {
    phone_status := rung
    } ;
```

The above construct enables a client to invoke an operation on a server interface specified by its identifier. The constructs allow the evaluation of multiple terminations.

Object deletion

An object or an interface can be destroyed by using the keywords *stop_obj* and

stop_interface along with the object and interface identifiers.

```
stop_obj phone
```

2.4 System initialization specification

The initial configuration of a system includes the definition and initialization of variables and the creation of new objects. However an object can also be created as an activity of operations at interfaces. Each instantiated object returns identifiers of its server interfaces. These identifiers can themselves be passed as arguments of another object creation as in the following example, where the administrator needs to know the identity of the network object server interfaces to communicate with them.

```
system_initialization

        let max_calls: Int = 0;

        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 ;
            ...
```

3. Specifying telephone systems using ODP-DLcomp

As an example of use of the language, the well known POTS (Plain Old Telephone System) [FaLS91] 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( )

        operation dial( phone_if : ident )
            termination dial_performed( )
            termination busy()

        operation talk( )
            termination talking( )


    behaviour

        choice(seq(offHook,dial,talk,onHook), seq(offHook,onHook))

end
```

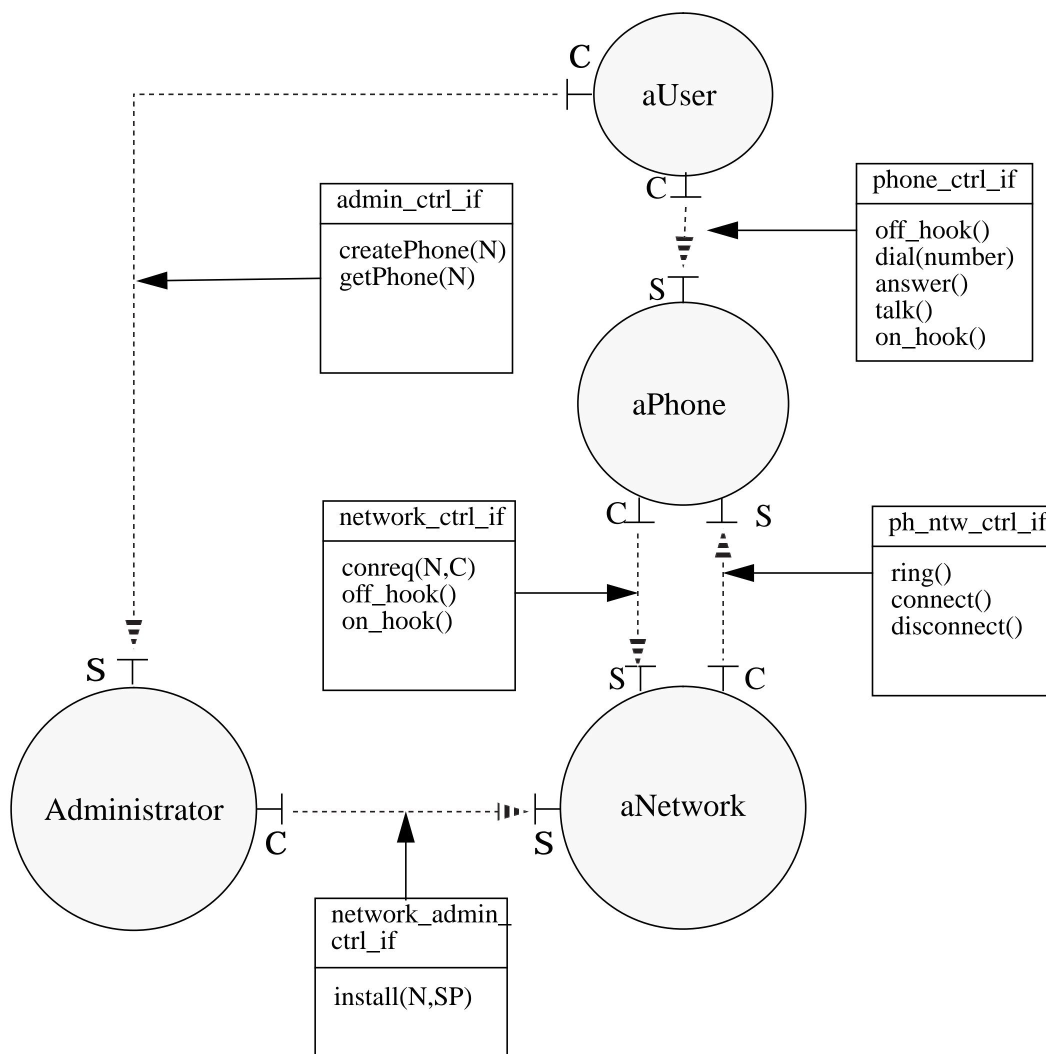

Fig.1 General architecture and interface model

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*).

```
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.

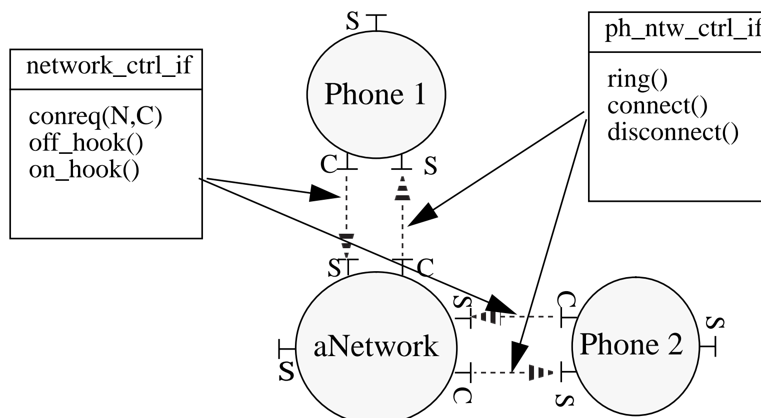

Fig.2 Multiple instances of interfaces of the network object

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

                let connection_state : state =nil ;
                let caller_control_interface:ident = nil;
                let called_control_interfac:ident = nil ;

                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.

3.3.1 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,
                                                                    server_network_cntrl_if_id2: ident)
    ...
    initialization
        ...
        instantiate admin_control_interface( ) server
                return admin_ctrl_if_id : ident ;
        ...
    behaviour

        SERVER_IFACE admin_control_interface :
            {
                createPhone(aNumber: number )
                {
                    if aNumber eq num1 then
                        create_object phone( aNumber, server_network_cntrl_if_id1 )
                            return phone_control_if_id: ident
                    else
                        ... ;
                    ...
                    terminate phone_created(phone_control_if_id)
            }
            ...
end

object_template phone( number: number, server_network_if_id: ident)
    ...
    initialization

        instantiate phone_control_interface( ) server
                return phone_ctrl_if_id : ident;
        ...
    behavior
        ...
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.

3.3.2 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.

So far, we have described the operation *ring* from the server side. We now show how it is invoked by the client network. The following is an excerpt of the network template.

```
object_template network( )
        ...
        behaviour

                SERVER_IFACE network_control_interface :
                    {
                        conreq(caller_number, called_number: number)
                           {
                            if called_number eq num1 then
                                   caller_phone_network_ctrl_if_id := phone_network_ctrl_if_id1
                            else
```

```
                                        called_phone_network_ctrl_if_id := phone_network_ctrl_if_id2 ;

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

Observe that when a connection request operation is invoked by a phone, the network will attempt to ring the requested number. In order to do this it is necessary to determine first the appropriate instance of the phone network control interface. Then, once the ring operation has been invoked on this selected interface, there are two cases: on the termination case *rung*, the network delivers a ring back to the phone, while in the termination case *busy*, it delivers a 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()
            }
        ...
        };

        SERVER_IFACE phone_network_control_interface :
        {
            disconnect()
            {
                state := disconnected
            }
            ...
        };
        ...
    end
```

On the network object side a call termination occurs when the disconnection operation is invoked on the network object. This operation changes the connection state attribute maintained by the network to *disconnected*, returns the termination *disconnected()* to the invoking phone and invokes a disconnection request on the called phone network interface if it was already involved. Due to multi-threading, it is possible that the network was already in the process of invoking other operations as part of a connection establishment procedure. In this case, any operation invocation should be guarded by a test on the connection state variable.

```
    object_template network( )
        ...
        initialization
                ...
        behaviour

            SERVER_IFACE network_control_interface :
                {
                    conreq(caller_number, called_number: number)
                      {
                        if called_number eq num1 then
                            called_control_interface := phone_network_ctrl_if_id1
                        else
                            called_control_interface := phone_network_ctrl_if_id2 ;

                        if connection_state ne disconnected then
                        {
                            invoke called_phone_network_ctrl_if_id->ring()
                                term_case rung() :
                                {
                                    connection_state := ringing
                                    terminate conreq_performed()
                                } ;
                                term_case busy():
                                {
                                    connection_state := aborting
                                    invoke caller_control_interface->disconnect()
                                        term_case disconnected():
                                        {
                                            connection_state := disconnected
                                        }
                                }
                        }
                      };
```

```
                disconnect_request()
                  {
                    if connection_state ne disconnected then
                    {
                        invoke called_phone_network_ctrl_if_id->disconnect()
                            term_case disconnected() :
                            {
                                connection_state:= disconnected
                            }
                    }
                    terminate disconnected()
                  };
                ...
                };
            ...
end
```

The reader should note the handling of disconnection collisions. The disconnect operation of a phone is invoked only if the connection has not already been interrupted by an on hook operation of the called party. This is the case where both parties go on hook at the same time without any disconnection from the other side.

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.

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.

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.

[FaLS91] Faci, M. , Logrippo, L., and Stepien, B. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach, Computer Networks and ISDN Systems 21 (1991), 52- 67.

[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.
[Koch94]Koch. F. Spezifizierung offener Verteilter Systeme aus Sicht des ODP Computational Viewpoint, Gesellschaft fuer Mathematik und Datenverarbeitung, GMD-Studien Nr. 243, October 1994.
[MRV91] deMeer, J., Roth, R., and Vuong, S. Introduction to Algebraic Specifications Based on the Language ACT ONE. Computer Networks and ISDN Systems, 23 (1992) 362-392.
[LoFH92] Logrippo, L., Faci, M. and Haj-Hussein, M. An Introduction to LOTOS: Learning by Examples, Computer Networks and ISDN Systems, 23 (1992) 325-342.
[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
[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