Graphic Visualization and Animation of LOTOS Execution Traces

Bernard Stepien and Luigi Logrippo

School of Information Technology and Engineering
University of Ottawa
Ottawa, Canada
(bernard | luigi)@site.uottawa.ca

Abstract. Two types of visualization and animation tools for LOTOS execution traces are presented: a translator from LOTOS traces to Message Sequence Charts and a graphic animator. These are made possible by enforcing specific LOTOS styles and by providing certain mappings between the elements of LOTOS actions and the elements of the graphic notation. The main application area considered is the visualization and animation of specifications of telephony systems. The use of these tools in software design is discussed briefly.

KEYWORDS: Telecommunications Software Engineering, Formal Description Techniques, LOTOS, Language of Temporal Ordering Specifications, MSC, Message Sequence Charts, Graphic Animation

1 Motivation

The LOTOS language [3][15] [21][29] is one of the standard Formal Description Techniques (FDTs) that have been used widely to specify many kinds of distributed systems, especially in the telecommunications area. The semantics of LOTOS is based on algebraic concepts and it can be used to describe systems at several levels of abstraction and in several ways, some of which may describe the constraints that determine the system behavior rather than the system architecture [30] [29][11]. This is an advantage because of various reasons: the language can be used in several phases of the software development cycle (from requirements to implementation [29] [1]); it provides flexibility to address a variety of applications; and it lends itself to many design and validation techniques. On the other hand this very quality has been an obstacle to industrial adoption, since the language has been perceived as difficult to learn and use, one of the reasons being the absence of generally valid correspondences between system concepts and specification concepts.

In papers that present LOTOS specifications, structural diagrams are often used [20][11]. This is useful in order to describe the components of a system and their external interaction points (the *gates*) but is not useful during the specification validation phase, neither to understand traces, nor to help in the exploration of alternative behaviors in a step-by-step approach. A graphic syntax for LOTOS, called G-LOTOS, was defined [5]. Graphical editors for it have been made available [6], as well as an animator called GLA (reported in 1994 by the group of the ETSI Telecommunications in Madrid but for which we have not found any published papers). Other interesting LOTOS animation and visualization tools have been reported in [28][32]. However all these approaches remain LOTOS-centered. They are not open to graphical notations and representations that are commonly used in telecommunications systems design.

Visualization techniques that are often used in the design and validation phases of telecommunications systems are Message Sequence Charts (MSC) and animation. These techniques are strongly connected by methods and tools to the language SDL [18]. They are impossible to use with the full generality of LOTOS because often, in specifications written in this language, one cannot identify components or messages. LOTOS processes may not represent physical (or even functional) components, LOTOS actions may not represent messages between components, and even when they do, these associations may not be obvious (this point is developed in Section 2.1).

It has become clear over the years that it is useful for certain purposes to harness the generality of LOTOS, and that this can be done by including particular information in LOTOS actions, as well as by the use of certain specification structures or 'styles'. This same idea has led to the definition of high-level languages (possibly graphically based) that can be translated into LOTOS [26] [27] , a thread that will not be pursued in this article.

We propose the use of a LOTOS style and a format of LOTOS actions from which notations and animation techniques that are understood by telecom specialists, such as MSC and user-oriented animation diagrams, can be generated. Two different graphic representations are addressed in this research. First, we show how LOTOS traces can be represented in the form of MSC and then we discuss graphic animation. The concepts used in the mapping between elements of LOTOS traces and elements of MSC are re-used for graphic animation. Although this paper is based on the 1989 LOTOS standard (which is still the only one for which good tools are available), the principles apply to the Extended LOTOS (E-LOTOS) language [16] [33]as well.

This work took place within the context of an industrial project to evaluate the use of FDTs for the specification of telephony services at the design stage. The graphic representation of execution traces and animation were an important part of the project. Since then, the tools described here have been used in other projects. This is discussed further in the last section.

2 Representing LOTOS Traces as Message Sequence Charts

The Message Sequence Chart (MSC) notation [17] is one of the best established notations for the design of telecommunications systems. After many years during which they were used informally by designers, they were standardized by the ITU. A similar notation in UML is the Interaction Diagrams [22]. These notations have differences which are being reconciled [14]. In this paper, we are not trying to be completely faithful to either notation (although the MSC textual notation produced by our tools is accepted by the Telelogic Tau tools). We use the term MSC to refer to a generic notation where parallel vertical lines represent communicating entities, and unidirectional arrows across, joining the vertical lines, represent named messages (or events) between the entities. Only one message exchange scenario (one sequence of message exchanges without choices) can be represented in one of our MSCs. Consistent with the semantics of process algebras, and LOTOS in particular, we assume that messages take a null time to travel from sender to receiver. It would be possible to relax this limitation and allow non-null arrival times, with the possibility of message loss and timeout. However this would require the definition, in the LOTOS specification, of an intermediate 'medium' process and more complicated mapping rules (than those that we introduce below). We have found that the simpler model proposed here is quite useful for practical applications.

2.1 LOTOS actions.

This section presents some characteristics of the LOTOS language that are important in order to understand the rest of the article. For information on this language, we refer to the LOTOS tutorials that have appeared in the literature [3][21] [29].

A LOTOS specification, when executed, produces *actions* made of a *gate name* and a number of *experiments* (sometimes also called *parameters*), such as:

gate_name !E_1 !E_2 . . . !E_n

where each E_1 is an abstract data type *value expression* that is a *ground term* [9] (in other words, we assume that there are no value identifiers and so all choices are resolved, which is reasonable both in terms of LOTOS theory and in terms of our application). A *trace* is a finite sequence of such actions.

A typical LOTOS action for the purpose of a database query might appear in a trace in this form:

```
AirFareDB ! getFare ! new_york ! paris ! 2000 ! dollars          (*)
```

We have here a gate *AirFareDB* that is an interaction point, and five experiments (parameters) that handle various aspects of this transaction: *getFare* is the requested operation, *new_york* and *paris* are the origin and the destination of the trip, *2000* is the amount of the fare and *dollars* is the currency required for the quote. In LOTOS, there are no requirements or significance regarding the order in which experiments appear in an action. The meaning of a value expression in an experiment can however be traced back (often not easily) to the abstract data type definitions.

The above action could result from the synchronization of two actions originating from processes representing client and server instances that could be specified in LOTOS as follows:

```
behavior
        AirFareClient[AirFareDB](bob, dollars)
        ||
        AirFareDatabase[AirFareDB]
where

process AirFareClient[AirFareDB](customer:name, currency:float):noexit:=

        AirFareDB ! getFare ! new_york ! paris ? Fare:float ! currency ; ...
endproc

process AirFareDatabase[AirFareDB]:noexit:=

        AirFareDB ! getFare ! new_york ! paris ! 2000 ! dollars ; ...
        []
        AirFareDB ! getFare ! new_york ! paris ! 2300 ! euros ; ...
        []
        AirFareDB ! getFare ! new_york ! london ! 1800 ! dollars ; ...
        []
        AirFareDB ! getFare ! new_york ! london ! 2070 ! euros ; ...
endproc
```

In this example we observe the matching of a request action on the client side with one of several actions on the server side that represent quotes for different routes and currencies. Note that what would be a two-message query/response sequence in SDL or MSC, or in an implementation, results in the single action (*) in LOTOS. Also, in (*) there is no indication that this action takes place between a client named bob and an air fare server. Further, since LOTOS semantics sees synchronization as symmetric, there is no indication of the direction of this action. In the LOTOS specification this information might be distributed very logically in various places, for example in process names or in abstract data type definitions. However each specification may do this in a different way and the person who wants to interpret a trace may need a major mental effort to recover this information.

Another noteworthy feature of LOTOS is that such atomic actions can involve any finite number of parallel processes, depending on the specification structure. This is called *multi-way synchronization*, and is often used to specify the combination of *constraints* in a system (hence the term *constraint-oriented style[30][25][10]*). In the telephony area there is a simple example of a three-way synchronization that can be used for specifying the busy behavior of a phone or circuit. In the example below, a *ring* action results from synchronization between two processes both offering a *ring* action, and the fact that a phone can be busy is specified by a third parallel behavior , with a predicate stating that the called phone should not be in the list of busy phones [10].

```
g ! num_2 ! ring  (* as offered by process phone *) ...
||
g ! num_2 ! ring  (* as offered by process switch*) ...
||
g ! num_2 ! ring [num_2 NotIn PhoneInUseList] ...
```

The third parallel action could be offered by a sub-process of the switch process or by some process representing a resource such as a Service Control Point, a database, etc. If this constraint is not satisfied there will be a deadlock, or another alternative will be chosen. While in the LOTOS style illustrated at the beginning of this section the processes correspond to components such as databases, client, etc., in the constraint-oriented style processes represent logical constraints and may have no physical or functional counterpart in terms of components or software modules.

In conclusion, actions in LOTOS do not need to contain the following information, often considered crucial in the design of distributed systems: the names of the components involved in the synchronization, the direction of information transfer, and the purpose of an experiment. By purpose of an experiment we mean what a particular experiment represents, in our case whether it is the name of a message, an instance name or a parameter. This may be one of the reasons why LOTOS, with all its expressive power, has not received significant industrial endorsement. Interestingly however, this generality is useful at the initial stages of design where it may be convenient to think in terms of actions that represent generic *responsibilities* to be refined later [7] [2].

2.2 The elements of an MSC

The ITU-T Z.120 standard for MSC includes the definition of a textual notation [17], of which we have retained the concepts of *instance head* and *message description*.

Instance heads are declarations that will result in vertical lines on the graphical representation. In ITU-T Z.120 terminology we speak of *instance* and *instance kind*. The name of the instance kind and the name of the instance will appear in that order on top of the vertical lines. In general, the latter is optional but it will not be so for us. The following example shows a kind of instance named *Client* for which there are two instances *bob* and *mary* and only one instance called *airlineServer* for the kind of instance *Server*:

```
bob: instancehead Client;
mary: instancehead Client;
airlineServer: instancehead Server;
```

The message description in MSC textual notation corresponding to the LOTOS action given at the beginning of the Section 2.1 (*) might be:

```
airlineServer: in getFare, 1 (new_york, paris, dollars) from bob;
bob : in Quote, 2 (new_york, paris, 2000, dollars) from airlineServer;
```

These two lines represent two messages, one for query and one for response, each of which is graphically represented by an arrow connecting two instances *bob* and *airlineServer* with the parameter values shown in parentheses. The first message is from bob to airlineServer, the name of the message is getFare, and the parameters are new_york, paris and dollars. The second message should be read similarly. 1 and 2 are message numbers.

2.3 Defining the mapping

The problem of establishing a mapping from elements of LOTOS traces to elements of the MSC notation is much more complex than the problem of establishing such a mapping between SDL and MSC. SDL describes communication between only two processes at a time. The communication consists in sending unidirectional *signals* from a process to another, and SDL signals have a pro-forma correspondence to MSC messages. Therefore, the mapping from SDL execution traces to MSC notation does not require user effort; in fact MSC is the normal representation of such traces. We have seen above that the type of communication that can be specified between LOTOS processes can be much more complex. Therefore, if it is desired to represent LOTOS traces in the form of MSC, the LOTOS style to be used, as well as the information to be included in the LOTOS actions, must be governed by the rules of MSC. In addition, the correspondence between LOTOS constructs and MSC constructs must be described by an appropriate mapping language. Three problems have to be addressed:

- expressing directionality in LOTOS actions, and assigning a direction to each of them
- establishing a correspondence between experiments in different positions of LOTOS actions and the MSC language elements of kinds and instances, message names, and parameter values
- defining a LOTOS specification style that is appropriate for the mapping

These problems will be discussed in the following sections.

2.3.1 Expressing directionality by gate names

Since the concept of directionality does not exist in LOTOS but is fundamental to MSC (all MSC messages are unidirectional), we need to find a convention to express directionality in LOTOS actions so that they can be unambiguously translated. This can be achieved by using one of the two basic components found in a LOTOS action, i.e. the gate or an experiment in the experiment list. We decided that the simplest and probably most expressive way to handle directionality is by using gate names constructed by using instance kind names linked by the symbol *_to_*. This means that a separate gate for each direction is necessary. For example, synchronization between instance kinds *client* and *server* will occur through the two gates *client_to_server* and *server_to_client*.

In this way, a single gate name can serve multiple instances of communicating entities as is often done in LOTOS specifications. We enable separation of concerns in the LOTOS action by using the gate name to convey information that we want to be fixed and invariable for each type of action: the communicating kinds of instances and the direction of the messages between them. Instance identifiers, message names and parameter information that are more variable in number and content are placed in the experiment lists. This convention has been inspired by the *gate splitting* concept found in the LOTOS literature [4][13]. This will become clearer when we present the mapping language in the next section.

2.3.2 Establishing a correspondence between MSC and LOTOS action concepts

Next, we discuss the appropriate representation in LOTOS experiments, and the translation into MSC format, of structured data units, such as message contents. There are at least two ways to represent structured data units in LOTOS. One way consists in combining everything in a single value expression, such as:

```
connection_request(phone_a, switch_x, phone_b)
```

Using this form for our purpose might however require complex mappings to deeply nested operations in the value expressions. A more practical method consists in using single atomic constants that are distributed among several experiments where they can be easily located by the transformation procedure as in the following example:

```
phone_to_switch ! phone_a ! switch_245 ! connection_request ! phone_b
```

Another problem is caused by the fact that MSC identifiers are character strings, and they must be constructed by using LOTOS value expressions. For example the value:

```
succ(succ(0))
```

is not valid according to MSC syntax and thus must be transformed into the following string, which is valid:

```
succ_succ_0
```

Also, character strings obtained from different LOTOS experiments must sometimes be merged into one character string in the corresponding MSC. This is shown in the example of Figure 1. *Action templates* define the mapping between elements of LOTOS actions and elements of the MSC notation. Each action template is identified by two indicators: a gate and a message name. This is because the same message could be exchanged between different instances, also different messages could have different experiment structures by the positioning of the experiments or by their total number. The third piece of information in an action template is the mapping between experiments and the MSC language elements of message, instances and parameters. This mapping is specified as a list of tuples indicating the experiment's position in the action, and its function that can be one of four kinds: *message*, *parameter*, *origin instance* and *destination instance*. The message and the parameters are represented as value expressions in given positions in the experiment list, but, as mentioned above, the origin and destination instances can be composed by concatenating the value expressions of several experiments.

The mappings need to be specified using some notation. Initially we had created our own notation but later we decided to use XML [31]. This choice has been motivated by the fact that this notation is well-known and has good tools, namely for visual editors and parsers. Also it is adequate because the required mappings are simple in structure.

The mappings are expressed using three XML main tags, *gate*, *message* and *experiments_mapping*. These address the three main components of an MSC message:

- the instance kinds involved, and the direction of the message that are obtained from the gate name

- the origin and destination instances names and the name of the message that are obtained from the experiment list

- the parameters of the message that are also obtained from the experiment list.

The *position* attribute of the mapping tag indicates which experiment in the LOTOS action maps to which MSC element. Note that several different messages on the same gate can share the same experiment mapping structure.

Consider a LOTOS action representing a connection request message between a phone and a switch. Suppose that this action contains the name of the originating entity in the first and second experiment, the name of the destination entity in the third experiment, the name of the message in the fourth experiment, and the value of the parameter (destination number) in the fifth experiment, e.g.

```
phone_to_switch !613 !2301111 !switch_245 !connection_request !5625800
```

An action template for this action can be represented in XML as follows:

```
<action_template>
        <gate> phone_to_switch </gate>
        <messages>
                <message> connection_request </message>
        </messages>
        <experiments_mapping>
                <mapping position="1" function="OrigInstance"/>
                <mapping position="2" function="OrigInstance"/>
                <mapping position="3" function="DestInstance"/>
                <mapping position="4" function="Message"/>
                <mapping position="5" function="Parameter"/>
        </experiments_mapping>
</action_template>
```

This would produce the MSC in figure 1.

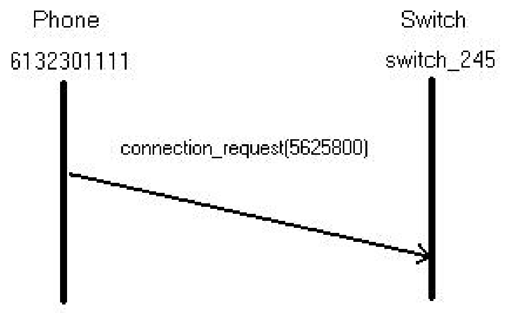


Figure 1 Sample MSC

Note that the keyword *message* appears twice in the above definition. The first occurrence as a tag name gives the name of the message to which this action template refers, the second occurrence as a function indicator gives the position of the message name in the experiment list.

Note also that, since we use XML that is an external language to LOTOS, all ADT names in the LOTOS specification must be global.

2.3.3 LOTOS specification constraints

There are two style constraints to consider, and they relate to multiple messages in a single action and multi-way synchronization.

2.3.3.1 Multiple messages in a single LOTOS action

We have seen in Section 2.1 that it is possible in LOTOS to express powerful actions that specify simultaneous data transfers in different directions as the result of an atomic event. Such actions cannot be expressed in MSCs and therefore they cannot be used in actions that appear in traces if it is desired to represent such traces as MSCs. They must be decomposed into sequences of simpler actions.

For example, the busy phone detection described in Section 2.1 could be specified instead as follows:

```
        switch_to_phone ! num_2 ! ring  (* as offered by process phone*)
|[switch_to_phone]||
        (* as offered by process switch *)
        switch_to_database ! get_status ! num_2 ;
        database_to_switch ! num_2 ! line_free ;
        switch_to_phone ! num_2 ! ring
|[switch_to_database, database_to_switch]|
        (* as offered by the database process *)
        switch_to_database ! get_status ?phone_num:number ;
        database_to_switch  !  phone_num  !  line_free  [Phone_Num  NotIn
        PhoneInUseList]
```

And again to handle a busy line, we need to offer alternate sequences of actions in the Switch and Database processes, yielding the following trace:

```
database_to_switch ! num_2 ! busy_line ;
switch_to_phone ! num_2 ! busy_signal
```

Note that LOTOS specifications written in this style are closer to the implementation level than those written in the more concise style presented in Section 2.1. They are more easily translated into an implementation language. In a LOTOS-based software development process, it may be necessary to accommodate several LOTOS styles, including the ones that we have presented, in order to be able to go from the initial to the later stages of software development. *Equivalence-preserving style transformations* that can be used in this process have been discussed in [4][12].

2.3.3.2 Handling LOTOS multi-way synchronization

Initially, one may think that multi-way synchronization cannot be represented directly in MSCs. This is because multi-way synchronization could be interpreted in some cases as a series of multiple simultaneous exchanges of messages between more than two entities. In these cases the approach is the same as the one discussed in the previous section (using a different specification structure). Otherwise, multi-way synchronization does not need to be eliminated. It is necessary to include in the actions all the elements that we have seen, and the same action structure will be used across all parallel processes. The mapping to MSC will only see the final compound action, which can be the result of multi-way synchronization, since at the trace level there is no indication of this fact. Note also that in the constraint oriented style, when there are several constraint processes in parallel, often only one process is active at a time for a particular set of values and all others do silent moves. For example in the busy phone example discussed previously the following three-way synchronization can be interpreted as a communication between a switch and a phone and thus uses a gate name that includes only these two entities:

```
switch_to_phone ! num_2 ! ring     (* as offered by process phone *) ...
||
switch_to_phone ! num_2 ! ring     (* as offered by process switch*) ...
||
switch_to_phone ! num_2 ! ring [num_2 NotIn PhoneInUseList] ...
          (* a constraint that is in essence part of the switch logic *)
```

3 Graphic Animation

The basic MSC notation we have discussed cannot represent choices in a single MSC. However the MSC notation of reference [17] is capable of representing alternative scenarios. In LOTOS, alternative scenarios will appear as choices in a behavior expression or alternative branches in a Labeled Transition System [3]. But such choices can result from expansion of any operator, including parallel composition. Waiting for a solution to this problem (and for the other reasons mentioned below), animation remains an attractive choice.

Animation is a complementary technique that shows the system's behavior in a number of scenarios. It can show graphic icons to represent kinds of instances, the state of an instance, and alternative scenarios. In animation, the user has the option to observe one message at a time and to see the flow of messages as they occur between visual icons that are idealized representations of the real world.

Figure 2 shows an excerpt of a typical call animation sequence for POTS (Plain Old Telephone System). Titles have been added manually to improve understanding.

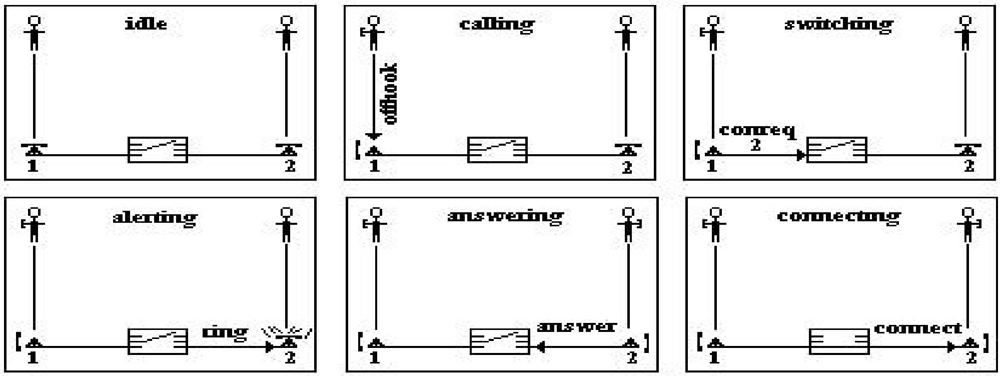


Figure 2 POTS call animation sequence

3.1 Animation of LOTOS traces

For graphic animation, we can re-use all the concepts developed so far for MSCs. The basic raw material, LOTOS actions and traces of LOTOS actions, are the same in both cases. We keep the concepts of unidirectionality of messages, instance and instance kind, however instances will be shown as graphical icons rather than as vertical lines. These icons can change as the actions follow each other. We will then say that the state of the instance has changed. Labels that before were applied to vertical lines are now applied to icons. Arrows that in MSCs were between vertical lines are now arrows between icons, but we will only see one of them at a time.

The main difference between MSC and animation is that MSCs provide a visual overview of message exchanges, while animation provides only one message exchange at a time, with the additional capability of dynamic exploration of alternatives that is not possible in MSCs.

Animation can be used in conjunction with a variety of LOTOS validation and simulation techniques. We have step-by-step exploration and automated state space exploration techniques that include random walk, reachability analysis, and others. In each of these techniques the user will identify certain traces to demonstrate specific scenarios or examples of behavior, for example behaviors that correspond to certain functionalities or reach certain states. In this paper, we will mostly discuss animation of traces in a context of step-by-step execution but of course the same principles can be used for traces obtained by other methods of execution.

We use the same set of icons for the various instances of a given kind. The instances are identified with labels carrying the instance's name placed above them. For example two instances of Bob and Jim would be represented by two icons representing a person with their respective names placed on top of the icon. The specific icon can change, representing a person in different situations (different states) as the trace evolves.

According to the conventions for LOTOS actions described above, gate names are composed using the names of origin and destination instance kinds with the string *_to_* between them. When an action is added to a trace, the names of origin and destination instance kinds are recovered to determine the icons to be used. For each instance kind we thus provide a list of mappings for each different message. If a mapping is missing for a particular message, the icon is left in its current state.

In principle, we would like the specific icons used after each message to represent the state of the resulting LOTOS specification. Unfortunately, this is not easy in LOTOS because in this language there are no state identifiers: the current state is the current behavior expression. It may be possible to gather enough information from this to guide the change of icons. However in order to do this, the graphic simulator would have to access the behavior expression, and this would increase its complexity. How to gather relevant state information from a behavior expression is not obvious, and probably a solution would involve imposing further constraints on the specification style (for example, the state-oriented style of [30]). We have chosen a simpler solution, involving only information available in the actions. An illustration of the resulting problem can be found again in the example of a telephone system. When a user performs the *dial digit* operation, this can be represented by an icon of someone punching keys on a phone. Then there would be some timeout and some further actions performed by the network that will not involve the user. In LOTOS, these would be hidden actions. By using our method, during this hidden sequence, the user's image of punching digit keys stays. In a better system, after the timeout, the user would be represented waiting with hands away from the dial.

The description of state representation in XML is as in the following example:

```
<icon_state_definition>
        <instance_kind name="User" gate_portion="user"/>
        <state_mappings>
                <state_mapping message="offhook" icon="offhook_man"/>
                <state_mapping message="onhook"  icon="idle_user"/>
        </state_mappings>
  </icon_state_definition>
```

We have here state transition definitions for icons of the instance kind called *user*. They say that an *offhook* message will cause the icon to become *offhook_man*, while an *onhook* message will cause the icon to change to *idle_user*.

Hence the following LOTOS action will cause the instance *user1* to be shown as a *offhook_man* icon:

```
user_to_phone ! user_1 ! phone_1 ! offhook
```

For this action we first extract the two instance kind names *user* and *phone* from the gate name *user_to_phone*. Using the message name *offhook* we search the icon state definitions for both instance kinds *user* and *phone* as both icons could change. This is illustrated in Figure 3 where we can observe the difference between the icons for an *offhook* message on the left and the icons for an idle user and phone on the right.

3.2 Animation with a LOTOS interpreter

In this part, we consider the graphic representation of traces interactively obtained by a step-by-step interpreter. We have used the LOLA (LOTOS Laboratory) tool from University of Madrid [24], but other tools having similar functionalities could have been used. We have added two features (in addition to the graphic animation already discussed in Section 3.1):

- presentation of LOTOS actions in a user-friendly stylized English-like format
- graphic animation of alternatives

3.2.1 Presentation of LOTOS actions in stylized English format

Alternative actions are displayed in a list box as in the GUI version of LOLA, however they are presented in an MSC-oriented 'stylized English' format using *from* and *to,* kind and instance names, messages and parameters.

For example the following LOTOS action:

```
phone_to_switch ! connnection_request ! num_1 ! switch_25 ! 123
```

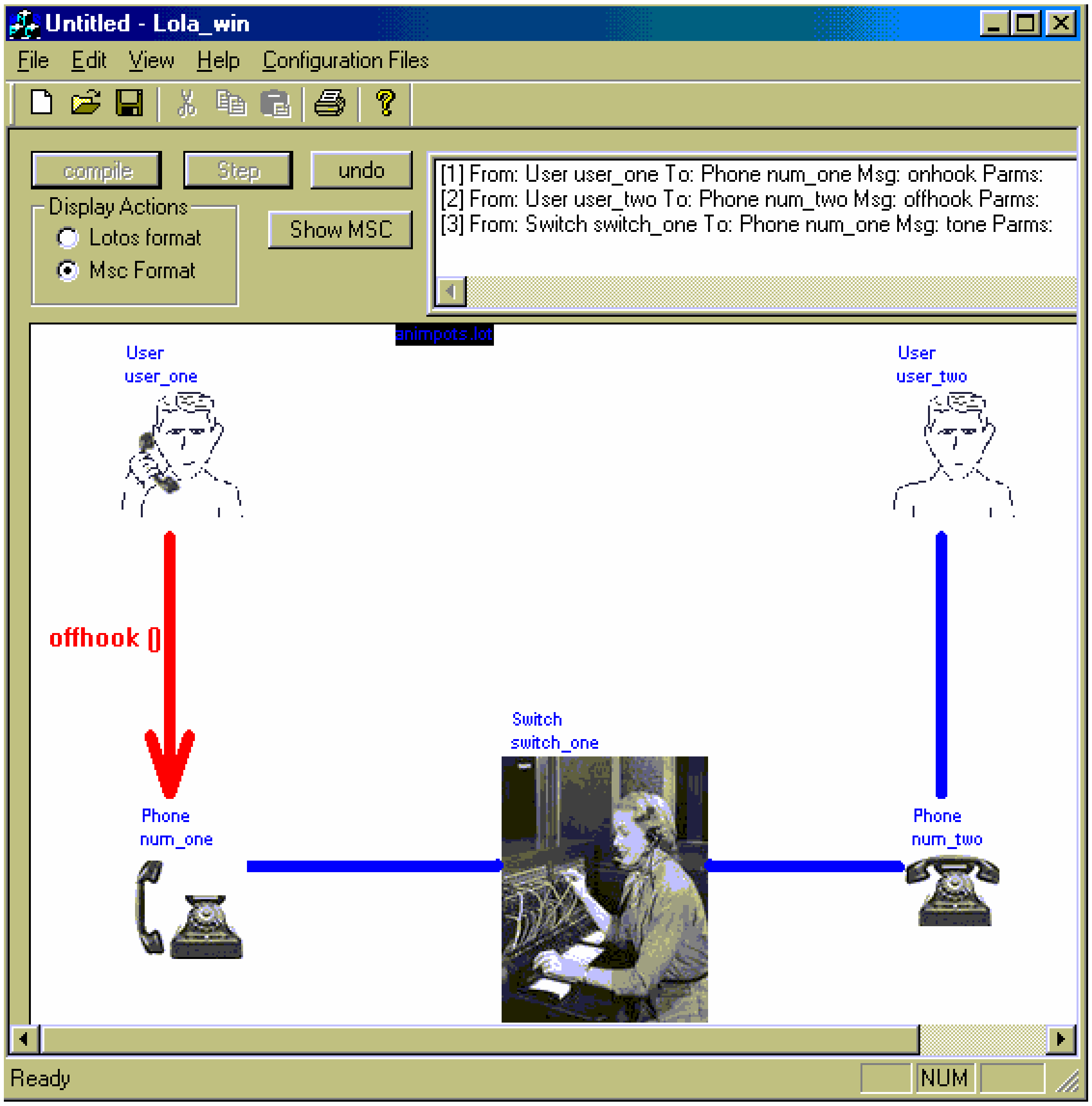
appears in a less cryptic form as:

Msg: connection_request **from** phone num_1 **to** switch switch_25 with **parameters**:123

This allows users that are not LOTOS experts to follow the exploration of the state space.

3.2.2 Graphic representation of alternatives

While MSCs represent traces that result from the execution of a LOTOS specification, in step-by-step execution one obtains two results: the trace of actions that have been executed and have led to the current state, and a list of next possible moves (actions) that can be executed from the current state. It is interesting for a user to explore the second result visually. It would be difficult to represent graphically several possible action derivations on the same screen, however we can at least allow the user to see the graphic display of one of them at a time. In order to distinguish them, we show them in a different color. A single click merely displays the result of a specific action without executing it and consequently not moving to the next state. After possibly attempting some of these actions, the user can double click on the action she wishes to execute and move on to the next state thus obtaining the next set of possible actions.



[1] From: User user_one To: Phone num_one Msg: onhook Parms:
[2] From: User user_two To: Phone num_two Msg: offhook Parms:
[3] From: Switch switch_one To: Phone num_one Msg: tone Parms:

Figure 3 Sample screen for the animation tool

4 Animation with interactive devices

Finally, a useful feature is to allow the designer of a system to interact with representations of real devices with buttons, displays, etc. with which it is possible to interact. The device is displayed in a separate window that can be moved around. This requires addressing problems such as: placement information, mapping graphical elements to LOTOS action elements, and LOTOS specification style.

In this part of the work, we were directly inspired by [19], where the specification language is SDL, although similar graphic simulators have been reported elsewhere in the literature.

4.1 Virtual Terminal description

A virtual terminal has a name that corresponds to the instance kind name it is associated with, and a list of element descriptions (for example, description of buttons) that contains both placement information and the essential mapping with LOTOS actions. Upon creation, a virtual terminal is associated with the name of its instance.

The following example of a virtual terminal definition applies to a *phone* that has an *instance name* of *num_1*. Then, we have a list of virtual device element definitions, where each element has a *placement information* part and a *device element to LOTOS mapping* part:

```
<virtual_terminal_definition>
        <instance_kind name=”phone”/>
        <instance name=”num_1”/>
        <virtual_device_element_definitions>
                <virtual_device_element_definition>
                        <placement_information>
                                …. (see definitions in subsequent sections)
                        </placement_information>
                        <device_element_to_lotos_mapping>
                                …(see definitions in subsequent sections)
                        </device_element_to_lotos_mapping>
                </ virtual_device_element_definition>
                …
        </ virtual_device_element_definitions>
</virtual_terminal_definition>
```

The placement definitions are the coordinates, the labels that appear on buttons, the types of graphical objects (button, text area, etc...) and possibly icons. A typical example in XML follows:

```
<placement_information>
        <coordinates X="10" Y="10" Height="30" Width="40"/>
        <label name=”5”/>
        <graphic_object kind=”button”/>
        <icon name="five.bmp"/>
</placement_information>
```

The most important part is the mapping of device elements to LOTOS actions. It is necessary to determine which action in the list of possible action offerings in the list box corresponds to which button on the virtual device. This can be achieved by mapping graphic information back to LOTOS, thus performing the reverse operation than the one of mapping LOTOS concepts to graphic concepts. The following example shows the structure of a *device element to lotos mapping* for a button, where the gate name is *user_to_phone*, the message name is *dial_digit* and the parameter is 1. Clearly, it represents the button 1 on the virtual device.

```
<device_element_to_lotos_mapping>
        <gate_name="user_to_phone"/>
        <message name="dial_digit"/>
```

```
                <parameters>
                        <parameter>1</parameter>
                </parameters>
        </device_element_to_lotos_mapping>
```

The selection of a LOTOS action in the list of next possible actions is done in four steps:

- Find the corresponding device element description in the XML list
- Using the device element description, find the corresponding action template using the gate name and the message name.
- Transform the LOTOS action into its MSC representation using the action template, thus obtaining the instance names of the origin and destination.
- Find the appropriate action by matching the device element values of gate name, instance name, message name and parameter values with those found in the MSC representation of the LOTOS action.

The action found is then executed with the LOLA tool.

When button 1 on the virtual device for a phone that has the instance name *num_1* is clicked, the above steps will compose the following action:

```
user_to_phone ! bob ! num_1 ! dial_digit  ! 1
```

Note that the description of only one of the two instances involved in communication needs to be provided because a device is only assigned to one of them. The same mapping is used when the tool is used in the reverse direction, with the virtual device driven by LOTOS actions as happens in the case of ring or call display messages.

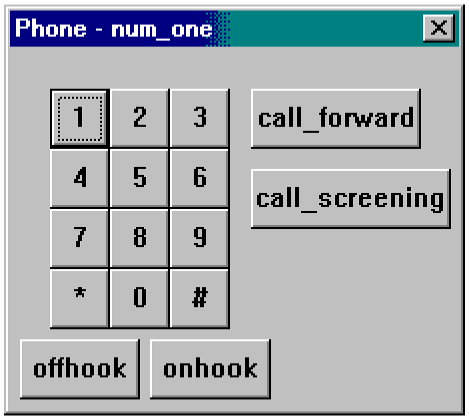


Figure 4 The interactive keypad

4.2 LOTOS specification issues

LOTOS behaviors are the result of synchronization between processes. At the highest level, there is an invisible process that is called the *environment*. The interactive device plays the role of the environment of the specification. An explicit description of both the device's behavior and of what interacts with it (e.g. a user) is required. For example, a telephone system is usually represented by a parallel composition between a phone and a switch, with the environment playing the role of the user. With interactive devices, we need also to specify the behavior of the user, leading to:

```
        user[user_to_phone, phone_to_user](bob, num_1)
|[user_to_phone]|
        phone[user_to_phone, phone_to_user, phone_to_network, network_to_phone](num_1)
|[phone_to_network]|
        switch[phone_to_network]
```

When specifying interactive devices by means of LOTOS processes, one cannot use the LOTOS variable declaration construct to bind values for actions specifying interaction with the environment such as:

```
gate_name ? phone_number:natural
```

Instead, we must specify one separate action for each device element such as a button. For example the actions of the process *user* of a virtual telephone can be represented by the following process that specifies each possible action that a user can perform with a phone set, namely an *offhook*, pressing a specific button to select a digit, and placing the set on hook (we use below a trivial relabeling of gates where u_to_p corresponds to user_to_phone, etc.):

```
process user[u_to_p, p_to_u](U:user, N:number):noexit:=
        u_to_p ! U ! N ! offHook ; user[u_to_p, p_to_u](U, N)
        []
        u_to_p ! U ! N ! dial ! digit_one ; user[u_to_p, p_to_u](U, N)
        []
        u_to_p ! U ! N ! dial ! digit_two ; user[u_to_p, p_to_u](U, N)
        []
                ... for each digit or special key such as * and #, etc...
        []
        u_to_p ! U ! N ! onHook ; user[u_to_p, p_to_u](U, N)
        []
        u_to_p ! U ! N ! talk ; user[u_to_p, p_to_u](U, N)
endproc
```

The corresponding process *phone* that receives the interactions from the user can use the expected input construct (? variable declaration) for the digits because this single action will match against each digit offered by the user process and will result in a list of twelve offered actions at the appropriate state during execution:

```
process phone[u_to_p, p_to_u, p_to_n, n_to_p](N:number, S:switch):noexit:=
        u_to_p ? U:user ! N ! offHook ;
        (
                n_to_p ! S ! N ! tone ;
                u_to_p ! U ! N ! dial ? CalledNumber:number ;
                p_to_n ! N ! S ! conreq ! CalledNumber ;
```

```
                        n_to_p ! S ! N ! connect ;
                        u_to_p ! U ! N ! talk ;
                                phone [u_to_p, p_to_u, p_to_n, n_to_p](N,S)
                        [> u_to_p ! U ! N ! onHook ;
                                phone [u_to_p, p_to_u, p_to_n, n_to_p](N,S)
                )
        endproc
```

However, it becomes clear that the keypad device shown in Fig. 4 does not correspond to the LOTOS process *phone* as one would expect because this process has only one single action for the *dial* message. This device can be thought of as describing the actions resulting from interactions between the *user* and the *phone* processes that in this case are governed by the actions of the user that presses keys on the keypad. One could also specify the phone with the same structure as the *user* process but this would be redundant because the final behavior would be equivalent.

The above example shows only a simplified version of dialing. A more appropriate specification would be in terms of a recursive digit collector with a termination feature such as a timeout or a function key such as the pound sign “#”.

For convenience, the user of the tool can elect to use either the actions list box or the virtual device to interact with the system. The virtual device can be obtained initially by double clicking on an object instance icon.

5 Applications and conclusions

We have presented two types of visualization and animation tools for LOTOS specifications: a translator from LOTOS traces to Message Sequence Charts and a graphic animator. These are made possible by enforcing a specific LOTOS style and by specifying mappings between LOTOS actions and the elements of the graphic notation. We have shown that the mappings needed for both visualization techniques are based on the same simple information model.

The first tool is called LOTOS2MSC, and has been used in several projects, starting with one in collaboration with the company Mitel, dealing with the analysis of Internet telephony features. In this project, we maintained two specifications for the same system, one in SDL and the other in LOTOS. Both specifications were obtained from Use Case Maps [7][2]. We wanted the two specifications to be trace-equivalent. The main use of LOTOS2MSC was to execute a LOTOS specification, obtain a MSC textual file, and then feed this one to an SDL specification for the same system that would either accept it or not. We have also developed a reverse tool, MSC2LOTOS, that takes a file in MSC textual notation and produces a LOTOS trace. This tool was used in the reverse direction, to check whether an SDL trace was also a legal trace for a LOTOS specification of the same system. Traces not accepted were printed in graphical MSC format for visual analysis. The reason why we wanted to have two trace-equivalent specifications of the same system are beyond the scope of this paper. It has to do with the fact that we used the LOTOS and SDL languages and toolkits in parallel, each for its strengths. LOTOS and

SDL also emphasize different aspects of a system, and using them both helps to increase confidence in the design.

The animation tool is useful in order to demonstrate the behavior of the system being designed to a person who is involved in the design from the user's point of view. We have developed a whole methodology of using these tools, however its discussion is beyond the scope of this paper. The main idea is that LOTOS is useful at the initial stages of the design process. LOTOS can express the behavior of a system at these stages in terms of sets of traces of abstract actions, and the tools are used to visualize the traces. An unfortunate point is that, as it was noted earlier, the LOTOS style that is forced by the need of including all the information required for visualization is of the kind that one would usually find towards the later stages of software development. The use of this style leads to other advantages however. For example, because of the fact that it gives explicitly some important information that is often hidden in LOTOS specifications, it makes the specification highly readable and easily transformable into an implementation.

Although this topic cannot be developed in this article, it is worth mentioning that the conversion of LOTOS traces to MSC has a benefit beyond graphic representation: it can lead to the generation of test cases using TTCN code generation tools that take MSCs as input [23].

Our experiences have led us to the conclusion that these tools, together with the associated methodology and the LOTOS specification style that they require, have a role towards increased practical usefulness and acceptance of LOTOS or related formal methods in telecommunications software development. Work on further developing this methodology, as well as on tools to support it, is ongoing in our research group.

Acknowledgments

This work was the result of much discussion in our research group: we would like to thank Daniel Amyot, Leila Charfi, Nicolas Gorse, Jacques Sincennes, as well as Tom Gray of Mitel for their feedback. The anonymous referees provided also a number of useful suggestions. This research was funded in part by Communications and Information Technology Ontario, the Natural Sciences and Engineering Research Council of Canada, and Mitel Corporation. This research was started while the first author was a guest scientist at GMD-FOKUS in Berlin.

References

[1] Amyot, D., Andrade, R., Logrippo, L., Sincennes, J., Yi, Z. Formal methods for mobility standards. Proc. of the 1999 IEEE Emerging Technologies Symposium on Wireless Communications and Systems. Richardson, TX, 1999

[2] Amyot, D., Logrippo, L. Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System. Computer Communications 23 (2000) 1135-1157.

[3] Bolognesi, T., Brinksma, E. Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59, also in: van

Eijk, P.H.J., Vissers,C.A., Diaz, M. (eds) The Formal Description Technique LOTOS. North Holland, 1989, 23-73.

[4] Bolognesi, T., De Frutos, D., Langerak, R., Latella, D. Correctness Preserving Transformations for the Early Phases of Software Development. In: Bolognesi, T., v.d. Lagemaat, J., Visser, C. LOTOSpere: Software Development with LOTOS. Kluwer, 1995, 161-180.

[5] Bolognesi, T., Najm, E., Tilanus, P.A.J. G-LOTOS: a graphical language for concurrent systems. Computer Networks and ISDN Systems 26 (1994), 1101-1127. Also appeared in different version in: Bolognesi, T., v.d. Lagemaat, J., Visser, C. LOTOSpere: Software Development with LOTOS. Kluwer, 1995, 391-450.

[6] Bolognesi, T., Hagsang, O., Latella, D., Pehrson, B. The definition of a graphical G-LOTOS editor using the meta-tool LOGGIE. Computer Networks and ISDN Systems. North-Holland, 22:61-77, 1991.

[7] Buhr. R. Use Case Maps as Architectural Entities for Complex Systems. IEEE Trans. on SE 24(12) 1998, 1131-1155.

[8] DOM - Document Object Model (DOM) level 3 Core Specification W3C Working Draft 14 January 2002, http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020114/

[9] Ehrig, H., Mahr, B. Fundamentals of Algebraic Specifications I, Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer, 1985.

[10] Faci, M., Logrippo, L., Stepien, B. Formal Specification of Telephone Systems in LOTOS: The Constraint Oriented Approach in Computer Networks and ISDN systems , 1990

[11] Faci, M., Logrippo, L., Stepien, B. Structural Models for Specifying Telephone Systems. Computer Networks and ISDN Systems 29 (1997) 501-528.

[12] Fantechi, A., Mekhanet, B., Najm, E., Cunha, P., Queiroz, J. Correctness-Preserving Transformations for the Late Phases of Software Development. In: Bolognesi, T., v.d. Lagemaat, J., Visser, C. LOTOSpere: Software Development with LOTOS. Kluwer, 1995, 181-199.

[13] Giannotti, F., Latella, D. Gate Splitting in LOTOS Specifications Using Abstract Interpretation. Science of Computer Programming 23 (2-3): 127-149 (1994)

[14] Haugen, O. From MSC-2000 to UML 2.0 – The Future of Sequence Diagrams. In: R. Reed and J. Reed (eds.). SDL 2001 Meeting UML. Springer 2001, 38-51.

[15] ISO – Information Processing Systems. Open Systems Interconnection. LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observation Behavior, ISO 8807, 1989 (E. Brinksma, ed.).

[16] ISO/IEC 15437:2001 Information technology -- Enhancements to LOTOS (E-LOTOS) (J. Quemada, ed.).

[17] ITU-T. Recommendation Z.120: Message Sequence Chart (MSC-2000). Geneva, 2000.

[18] ITU-T. Recommendation Z.100: Specification and Description Language (SDL). Geneva, 2000.

[19] Kimbler, K., Verhaard, L., Sobirk, D., Tallinger, T., Hagenfeldt, C., Stolt, M. Simulated IN CS2 Infrastructure for Service Animation and Validation" Proc. of 2000 IEEE Intelligent Network Workshop, May 2000, Cape Town.

[20] Kremer, H., Lagemaat, J.v.d., Rennoch, A., Scollo, G. Protocol Design Using LOTOS. In: Diaz, M. and Groz, R. (eds.) Formal Description Techniques, V. (North-Holland, 1993) 231-246.

[21] Logrippo, L., Faci, M., Haj-Hussein, M. An Introduction to LOTOS: Learning by Examples. Computer Networks and ISDN Systems 23 (5) (1992) 325-342. Errata in 25 (1992) 99-100.

[22] Object Management Group. Unified Modeling Language (UML), version 1.4, 2001. Available from www.omg.org

[23] Probert, R.L., Ural, H., Williams, A.V. Rapid generation of functional tests using MSCs, SDL and TTCN. J. of Computer Communications, 24 (2001) 374-393.

[24] Quemada, J., Pavon, S., Fernandez, A. Transforming LOTOS Specifications with LOLA: the Parameterized Expansion. In: K.J. Turner (ed.), Formal Description Techniques 88. North-Holland, 1988.

[25] Turner, K.J. Constraint-Oriented Style in LOTOS. Proc. British Computer Society Workshop on Formal Methods in Standards. British Computer Society, 1988, 1-13.

[26] Turner, K.J., McClenaghan, A. Visual animation of LOTOS using SOLVE. In Hogrefe, D., Leue, S. (eds.), Formal Description Techniques VII, Chapman-Hall, London, 1995, 283-285.

[27] Turner, K., Realising Architectural Feature Descriptions Using LOTOS. Réseaux et sytèmes répartis (RSR-CP) 12/2000, 145-187.

[28] Turner, K. The N-Body Problem in LOTOS. In H. Bowman, editor, Proc. Formal Methods Elsewhere, pages 1-23, Electronic Notes in Theoretical Computer Science, volume 43, Elsevier Science, Amsterdam, Netherlands, June 2001.

[29] van Eijk, P.H.J., Vissers,C.A., Diaz, M. (eds) The Formal Description Technique LOTOS. North Holland, 1989.

[30] Visser, C.A., Scollo, G., van Sinderen, M.A., Brinksma, E. Specification Style in Distributed Systems Design and Verification. Theoretical Computer Science 89 (1991) 179-206.

[31] XML - Extensible Markup Language 1.0 W3C Recommendation 6 October 2000 http://www.w3.org/TR/2000/REC-xml-20001006

[32] Yasumoto, K, Higashino, T., Abe, K., Matsuura, T., Taniguchi, K. A LOTOS Compiler generating Multi-threaded Object Codes. In: G.v. Bochmann, R. Dssouli, O. Rafiq. Formal Description Techniques VIII, North-Holland, 1996, 271-286.

[33] Zambenedetti, L., Almeida, M.J. E-DART: A Specification Environment for the E-LOTOS Formal Technique. Proc. of the 4th Workshop on Parallel and Distributed Real-Time Systems, Orlando, Florida, 1998.

Bernard Stepien holds a BA and an MA in Economics from the Université de Montpellier (France), Subsequently, he carried out research in Transportation Science with the Montreal Transportation Commission and worked as an economist for Bell Canada. He has been a private consultant in computer and telecommunications applications since

1975 with the Canadian Government, Bell Canada, Nortel Networks, GMD-FOKUS in Berlin and Telcordia Technologies in New Jersey. Besides his ongoing research on LOTOS at University of Ottawa, he is carrying out research on TTCN-3, XML and automated software generation.

[A photo can be found in Computer Networks 29 (1997), 528]

Luigi Logrippo received a degree in law from the University of Rome (Italy) in 1961, and in the same year he started a career in computing. He worked for several computer companies and in 1969 he obtained a MSc in Computer Science from the University of Manitoba, which was followed by a PhD in Computer Science from the University of Waterloo in 1974. He was with the University of Ottawa for 29 years, where he was Chair of the Computer Science Department for 7 years. In 2002 he has moved to the Université du Québec à Hull, Département d'informatique. His interest area is formal methods (especially those based on process algebra and logic) and their applications in in the design of telecommunications systems. He worked for many years on LOTOS, its applications, and its extensions. Currently he is exploring the areas of internet telephony and mobile telephony, with reference to features and policies.

[A photo can be found in Computer Networks 34 (2000), 703]

.