An Introduction to the D-Bus Language Binding for ooRexx

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna
"The 2011 International Rexx Symposium", Oranjestad, Aruba, Dutch West-Indies
December 4th – 7th, 2011

Abstract. This article introduces the ooRexx language binding for the D-Bus interprocess communication. In order to do so it first introduces the D-Bus concepts and how they relate to each other. The dynamically typed ooRexx language possesses a C++ API that gets used together with a matching ooRexx package to create the language binding for the D-Bus. After introducing the core ooRexx class, "DBus", the abilities for creating ooRexx proxy objects for remote D-Bus service objects, ooRexx listeners for D-Bus signals and ooRexx D-Bus service objects get documented. ooRexx nutshell examples serve to demonstrate each of these abilities of the ooRexx D-Bus binding, which the reader can use to assess this particular language binding. ooRexx programmers should become able to understand the D-Bus concepts and put them to work for their purposes, after studying this article.

1 Introduction

D-Bus is an interprocess communication wire protocol that got originally developed by Red Hat [1]. Over the course of time the D-Bus development has been anchored with the Linux "free desktop" [2], [3] initiative, that develops open source technology that can be deployed on any Linux operating system.

D-Bus is being used on Linux systems for kernel programs to communicate with other parts of the Linux system and to emit signals that may be interesting for desktop application programmers, like gaining or losing access to devices and the like. These particular communications are taking place over a D-Bus "system" message daemon (dubbed "system bus") that gets started when Linux boots. By default the system bus is sheltered by rigorous security measurements that depend on configuration files that only administrators can change.

In addition, whenever a user logs into a Linux system and thereby starts a new session a separate D-Bus "session" message daemon (dubbed "session bus") gets started. This D-Bus allows any application in the user's session to interact with each other. It is e.g. possible to interact with editors and in effect remote control them via D-Bus, given that these editors supply a D-Bus interface.

In the meantime the D-Bus interprocess communication infrastructure has been ported to MacOSX and Windows. Having D-Bus use the TCP/IP communication infrastructure allows applications to communicate with each other independent of operating systems, programming languages or hardware infrastructures. This opens up interesting opportunities for businesses to create, mix and control their deployed IT infrastructure, possibly allowing managers to break out of lock-ins over

time.

There are language bindings for D-Bus for programming languages and libraries [4] like e.g. C, C++, GLib, Qt4, Python, Ruby, TcL, which allow programmers of such languages to simplify the creation and use of D-Bus applications.

This article is about a new language binding for D-Bus, enabling the easy to learn and yet powerful scripting language ooRexx to become a “first-class citizen” in the world of D-Bus interprocess communication, which is dubbed “dbusoorexx”. The article does not discuss the implementation of the language binding itself, although some ideas might be interesting for others, but rather addresses ooRexx programmers who may have never heard of D-Bus and who should be empowered to understand its concepts and become able to exploit the ooRexx language binding for their (and their companies) benefits.

The article is structured such that it gives first an overview of the D-Bus concepts, followed by the description of the ooRexx class “DBus” which represents a D-Bus connection and supplies all functionality necessary to send and receive D-Bus messages over the D-Bus connection. To ease programming D-Bus even more, the ooRexx D-Bus package includes ooRexx classes that allow proxying remote D-Bus objects, D-Bus listeners and D-Bus service objects that make it quite simple to create ooRexx services that others can exploit via D-Bus. The description of the available functionality is demonstrated with simple nutshell programs such that the ooRexx programmer can see for herself how easy implementing D-Bus interprocess communication has become with this language binding.

2 Overview of D-Bus (D-Bus Concepts and Terms)

D-Bus is a wire protocol that has been standardized by the `freedesktop.org` initiative and allows two processes to communicate with each other.

In addition the term “D-Bus” is also used for a *message management system* that usually runs as a “daemon” and which is able to serve as a *broker* of messages, being in the center of a star-shaped communication setup. On Linux there is by default a system wide message daemon, dubbed “system bus”, which allows system wide message communication. In addition Linux distributions create a “session bus” for each created user session, such that applications in a user session are able to exchange messages with each other.

This infrastructure can be used for example to interact with desktop applications, or to learn about system wide events by receiving signal messages from the system bus.

The specifications and the documentation of the application programming interfaces are freely available [5], [6]. For a partial list of current projects that are based on D-Bus, cf. [7].

2.1 D-Bus Transports

D-Bus makes it easy for a process to create a D-Bus server to which an application in another process can create a D-Bus connection. At the time of writing there were the following *TRANSPORTS* defined that could be used for setting up a D-Bus communication:

- Unix sockets, where the server and the client must reside on the same physical machine, address prefix "`unix:`",
- `launchd`, where the server and the client must reside on the same physical machine, address prefix "`launchd:`",
- nonce-TCP/IP sockets, where the server and the client must reside on the same physical machine, address prefix "`nonce-tcp:`", and
- TCP/IP sockets, where the server and the client can reside on different physical machines, address prefix "`tcp:`". This mode allows programs running on different hardware and operating systems to communicate with each other using D-Bus.

2.2 D-Bus Messages

One peer can send messages to another peer, which as a result may send a reply or an error message in turn. In addition it is possible to send signal messages, that never can cause a reply.

The message format is simple: it defines an interface name (comparable to a package name in Java), a member name (comparable to the name of a method or of a field in Java), and arguments that may be supplied with the message, as well as a return value, if any.

2.3 Interface and Member Names

An *INTERFACE* name can contain ASCII letter characters, an underscore (`_`), digits, and dots. It must not start with a digit or a dot and cannot exceed a total of 255 characters. An interface name must contain at least one dot, which separates the *ELEMENTS*. Interface names are mandatory for a *SIGNAL MESSAGE*, but optional for all other D-Bus message types (*CALL MESSAGE*, *REPLY MESSAGE*, *ERROR MESSAGE*).

A *MEMBER* name follows the same rules as an interface name, but must not contain a dot and must consist of at least one character.

2.4 Types of Values Transported with D-Bus Messages

Values that are transported or replied with a D-Bus message are strictly typed. Table 1 below depicts the types that may be used for these values as well as their single character abbreviation which may be used when creating a signature that defines the types of the transported values. A basic *TYPE* is any type indicator that is not representing a container type. A *CONTAINER TYPE* is either a structure, an array, a map (a dictionary) and a variant, which contains a signature and the matching value.

Data Type	Type Indicator	Comment
array	`a`	If sequence of '`a`'s, then each '`a`' stands for one dimension, followed by the element type indicator of the array.
boolean	`b`	8-bit unsigned integer
byte	`y`	'`0`' (`false`) or '`1`' (`true`)
double	`d`	IEEE 754 double
int16	`n`	16-bit signed integer
int32	`i`	32-bit signed integer
in64	`x`	64-bit signed integer
objpath	`o`	Must start with a slash (`/`).
signature	`g`	May consist of type indicators only.
string	`s`	Must be encoded as UTF-8.[1]
uint16	`q`	16-bit unsigned integer
uint32	`u`	32-bit unsigned integer
uint64	`t`	64-bit unsigned integer
unix_fd	`h`	Available only, if using Unix socket transportation.
variant	`v`	A container type which includes the signature of the encoded value.
structure	`(…)`	A container type. Parentheses may contain any types.
map/dict	`a{s…}`	A container type. Map/dictionary, index is always a string, value can be of any type.

Table 1: D-Bus type indicators.

2.5 D-Bus Connection (“Bus”)

If a connection is established to a server, this connection is called a *BUS*, which is used to exchange any kind of messages. A *BUS NAME* follows the same rules as an interface name, but in addition may have a dash (`-`) and a colon (`:`). When running on a system or session message bus, then the D-Bus

1 If an ooRexx string contains non-US-ASCII characters, then the public routine `stringToUtf8(string)` can be used to encode the ooRexx string as an UTF-8 string.

daemon managing it, assigns a *UNIQUE BUS NAME* to each connection it accepts.[2, 3]

Each program is able to request a unique bus name[4] for itself, by requesting the desired name from the D-Bus broker (message daemon), which makes sure that any bus name is unique among all processes that are connected to it. As dynamically assigned unique bus names cannot be known in advance, requesting an arbitrary unique bus name and documenting that name in the public allows any other D-Bus process to contact the process that owns the published unique bus name.

The D-Bus broker is able to automatically start processes that are defined in so-called *SERVICE* files to supply that "well-known" bus name. These processes then will be able to service clients that seek to use its services.

2.6 D-Bus Message Daemon: Exchange of Messages

Having a D-Bus message daemon allows any two processes to send each other messages with the help of the D-Bus broker, which receives and forwards messages. Figure 1 below depicts four processes that communicate with each other with the D-Bus communication infrastructure, employing a D-Bus broker. `Process A`, `Process B` and the `Notifications` process all connect to the D-Bus broker process, which can also be addressed via its well-known bus name `org.freedesktop.DBus`. Each process is shown with its (arbitrary) unique bus name. In order to exchange messages, a process needs to tell the broker the destination address of the message, which is the unique bus name of the target process.

Figure 1 shows message exchanges (blue, dashed line with arrow heads) between `Process A` and the `Notifications` process which are relayed via the D-Bus broker process. The `Notifications` process possesses a unique bus name, `org.freedesktop.Notifications`, such that `Process A` just needs to know that unique bus name, in order to be able to address that process.

Service processes may supply many different services, where each service is conceptually represented by a (service) object. The D-Bus communication infrastructure provides the concept of an *OBJECT PATH* which can be used to identify the particular service in the service's process that the client wishes to communicate with.

2 It is possible to start as many message bus daemons as one sees fit by using the `dbus-daemon` program that gets distributed with D-Bus and is documented with the help of man pages on Linux.

3 Using a D-Bus daemon as a broker incurs, that it reserves the right to use a colon exclusively for itself to assign unique bus names, i.e. the broker would not accept bus name requests from its clients that contain a colon!

4 Synonyms for "unique bus name" in this article are "service name" and "well-known bus name".

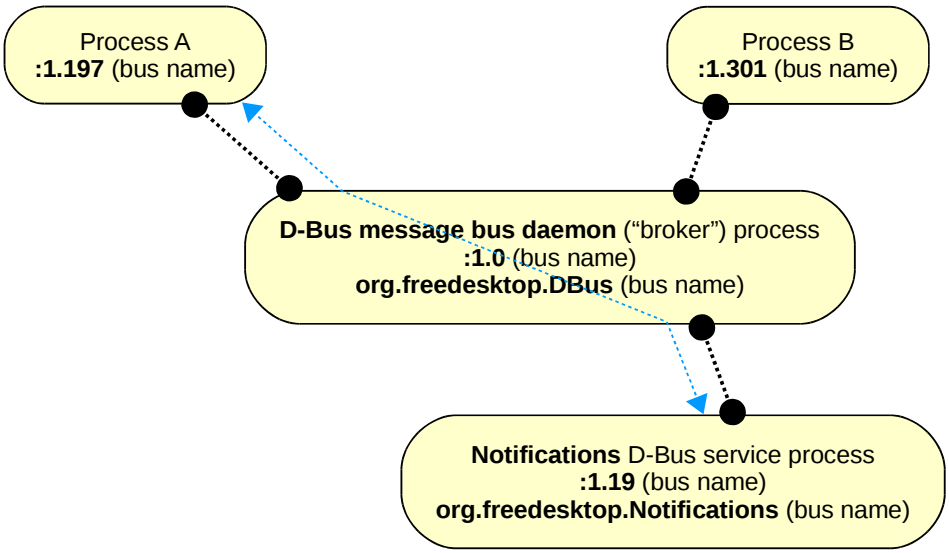

Figure 1: Processes communicating with and via the D-Bus message bus.

2.7 Object Path

An *OBJECT PATH* name can contain ASCII characters, an underscore (_), digits, and slashes. It must not be empty and start with a slash (`/`) and can be of any length. An object path must not contain two consecutive slashes or end with a slash, unless the object path name consists of a single slash only.

2.8 Discovering the Interfaces of Service Objects

By convention each D-Bus service object is able to submit an XML encoded string that documents all the published interface names with all of their published members. The interface name of this particular method is `org.freedesktop.DBus` and the member name is `Introspect`, which does not expect any arguments and returns a value of type string (`s`), cf. table 2.

Returns	Message Type	Member Name
`s`	*method*	**Introspect**( )

Table 2: Interface: `org.freedesktop.DBus.Introspectable`.

Members are either of type *METHOD*, *SIGNAL* or *PROPERTY*. The type of the arguments of methods, signals and properties are encoded with the D-Bus type indicators given in table 1 above. Methods that do not return any values will document this with an empty string as their signature. Signals, being one-way (broadcast[5]) messages may carry arguments, but will never return a value.

If a service object supports properties, then in order to get or set their values, some or all of the methods of the `org.freedesktop.DBus.Properties` interface as depicted in table 3 below will be

5 A signal will be broadcast by the D-Bus message daemon to any connected process, if such a process registered the interest for receiving signals.

made available[6].

Returns	Message Type	Member Name
v	*method*	**Get**(ss)
	method	**Set**(ssv)
a{sv}	*method*	**GetAll**(s)

Table 3: Interface org.freedesktop.DBus.Properties.

2.9 Interfaces of the D-Bus Message Daemon

The D-Bus message daemon (also dubbed "broker" in this article) owns the bus name org.freedesktop.DBus[7] and supplies the interface org.freedesktop.DBus.Introspectable with the method member Introspect, returning a string that describes all published interfaces and their members. Table 4 below depicts the org.freedesktop.DBus interface and its members with their signatures.

Returns	Message Type	Member Name
	method	**AddMatch**(s)
ay	*method*	**GetAdtAuditSessionData**(s)
ay	*method*	**GetConnectionSELinuxSecurityContext**(s)
u	*method*	**GetConnectionUnixProcessID**(s)
u	*method*	**GetConnectionUnixUser**(s)
s	*method*	**GetId**( )
s	*method*	**GetNameOwner**(s)
s	*method*	**Hello**( )
as	*method*	**ListActivatableNames**( )
as	*method*	**ListNames**( )
as	*method*	**ListQueuedOwners**(s)
b	*method*	**NameHasOwner**(s)
u	*method*	**ReleaseName**(s)
	method	**ReloadConfig**( )
	method	**RemoveMatch**(s)
u	*method*	**RequestName**(su)
u	*method*	**StartServiceByName**(su)
	method	**UpdateActivationEnvironment**(a{ss})
	signal	**NameAcquired**(s)
	signal	**NameLost**(s)
	signal	**NameOwnerChanged**(sss)

Table 4: Interface org.freedesktop.DBus *on Ubuntu 11.04, 64-Bit.*

6 The first string argument denotes the interface name. The second string argument, if given, denotes the name of the affected property.

7 The object path for the D-Bus daemon/broker is defined to be "/org/freedesktop/DBus", but "/" works as well.

2.10 Private D-Bus Server

The D-Bus interprocess communication infrastructure allows programmers to use the infrastructure without the help of a D-Bus message daemon. Such servers are called *private D-Bus servers*. This allows for simple client/server applications where any process having a connection to the private D-Bus server can communicate with it using D-Bus messages.

3 ooRexx Language Binding for D-Bus

The scripting language “Open Object Rexx (ooRexx)” [8] is an object-oriented extension of the REXX scripting language, which was intentionally designed as a “human-oriented” language. [9] gives a brief overview of its history and its features. The language can be briefly characterized as:

- an interpreted language,
- having an easy, pseudo-code like syntax,
- being dynamically typed (REXX: “everything is a string”, ooRexx: “everything is an object”),
- caseless (everything outside of quotes gets uppercased by the interpreter before execution),
- possessing an explicit message operator, the tilde (~); left of the tilde is the receiving object, right of it the method name and, optionally (in round brackets) the arguments supplied with the message,
- drawing concepts from Smalltalk,
- possessing an easy to use, yet powerful C++ API.

ooRexx was originally developed by IBM which handed over its source-code to the non-profit special interest group “Rexx Language Association (RexxLA)” for opensourcing and further developing the language.

The author created a Java language binding for ooRexx, named “Bean Scripting Framework for ooRexx (BSF4ooRexx)”, which camouflages all of Java as if it was ooRexx [11]. Drawing from the experiences with creating that language binding over many years, the following goals should be achieved with the D-Bus language binding for ooRexx:

- where possible ooRexx semantics and mechanics should be put in place, such that D-Bus can be camouflaged as ooRexx,
- all features of D-Bus should be made available to ooRexx programmers,

- any strict typing should be hidden as much as possible from the ooRexx programmers, i.e. marshalling and unmarshalling should be as transparent and flexible as possible,
- if seen as necessary, additional utilities (routines, classes, programs) should be created, such that all of the D-Bus functionalities can be exploited in an easy to use manner by ooRexx programmers.

The code in the ooRexx package "`dbus.cls`" has to contain brief and understandable comments and where appropriate links to further relevant information.

After studying the D-Bus specification and some D-Bus language bindings the following design decisions were taken:

- map D-Bus messages directly to ooRexx messages, take advantage of the ooRexx built-in unknown mechanism[8]; this allows any ooRexx object to receive and process any D-Bus message,
- add by default a slot argument to any D-Bus message before forwarding it to an ooRexx object, which is an ooRexx dictionary (map) containing D-Bus message related information like message type, sender, signature and the like,
- implement marshalling and unmarshalling such that it is transparent to the ooRexx programmer; allow arguments to be optional (left-out) and supply safe default values in the marshalling code for them,
- create an ooRexx class that represents a D-Bus connection, named "`DBus`", which allows to establish a connection to the system and session daemons, as well as to D-Bus servers with a known address, to send and to receive messages,
- create an ooRexx class that serves as an ooRexx proxy for remote objects, camouflaging them to be ooRexx objects, named "`DBusProxyObject`",
- create an ooRexx class that allows local filtering of signals, named "`DBusListener`",
- create an ooRexx class that makes it easy to implement services in ooRexx, named "`DbusServiceObject`",
- support dynamic discovery of object paths for DBus clients,
- create an ooRexx class that makes it easy to establish a private D-Bus server, named

8 The ooRexx unknown mechanism allows to intercept messages and their arguments for which no methods are implemented in the receiver's class.

“DBusServer”,

- create support for creating and analyzing introspection data on-the-fly.

3.1 ooRexx Class “DBus”: Representing a DBus-Connection

To enable ooRexx to take advantage of the D-Bus interprocess communication the ooRexx class DBus got created in the ooRexx package named “dbus.cls”.

The class methods connect and new allow for creating new D-Bus connections by supplying an address to connect to. Due to the importance of the system and session message bus daemons it is possible to supply the pseudo addresses named “system” or “session” to establish a connection. In addition the class methods system and session can be used as well. Methods 1 below depicts the most important class methods.[9]

```
new( server_address | “system” | “session” )
connect(server_address | “system” | “session” )
system
session
```

Methods 1: The most important class methods of class "DBus" that return a connection object.

Each D-Bus connection will be serviced by an ooRexx controlled message loop, which runs on a separate thread, to receive and reply to messages, if listener or service objects are defined.

To send messages from ooRexx the instance method message must be used. Its first argument can be either “call” or “signal”, depending on the message type one wishes to send via the connection.

The method listener allows to add or remove an ooRexx listener object, as well as getting a list of currently registered ooRexx listener objects. An ooRexx listener object gets wrapped up and stored in a “DBusListener” object.

The method serviceObject allows to add or remove an ooRexx service object, as well as getting a list of currently registered ooRexx service objects.

The method busName allows to request and to release the ownership of a bus name.

Methods 2 below depicts the most important instance methods.[10]

Code 1 below depicts an ooRexx program, that interacts with a D-Bus service named org.freedesktop.Notifications and uses it to notify the user. The documentation of the interfaces of this service as available on Ubuntu 11.04 can be studied in figure 8 on page 22, which also shows

9 A vertical bar (|) delimits alternatives.

10 Square brackets enclose optional arguments. If the name of an argument is followed by an equal sign, then the value right of the equal sign will be the default value, if that particular argument was left out. If string literals are given, only the uppercase letters need to be given.

the signatures of the arguments.

```
busName(“Hasowner”, busName)
busName(“REQuest”, busName [,flags])
busName(“RELease”, busName)
busName(“Uniquename”)

close

getObject(busName,objectPath)

listener(“Add”,object[,[interface=.nil[,signalName=.nil]])
listener(“Remove”,object[,[interface[,signalName]])
listener(“Getlisteners”)

match(“Add”, matchRule [,reportError=.false])
match(“Remove”, matchRule [,reportError=.false])

message(“Call”,busName,objectPath,interface=””,memberName,replySignature=””,argSignature=””[,arg…])
message(“Signal”, objectPath, interface, signalName, argSignature=”” [,arg…])

serviceObject(“Add”, objectPath, object)
serviceObject(“Remove”, objectPath, object)
serviceObject(“Getserviceobjects”)
```

Methods 2: The most important instance methods of class "DBus".

The ooRexx program in code 1 below requires the package (module) “dbus.cls” which makes the ooRexx D-Bus language binding available.[11] The ooRexx program first acquires a connection to the session message bus daemon, then defines the values for the arguments to supply to the connection's message method.[12]

```
conn=.dbus~session      /* get connection to the session message bus     */

    /* define message arguments */
busName       ="org.freedesktop.Notifications"
objectName    ="/org/freedesktop/Notifications"
interfaceName ="org.freedesktop.Notifications"
memberName    ="Notify"
replySignature="u"             /* uint32 */
argSignature ="susssasa{sv}i"  /* string,uint32,string,string,string,
                                  array of string,dict of variants,int32 */

id=conn~message("call",busName,objectName,interfaceName,memberName, -
                replySignature,argSignature,"An ooRexx App", ,       -
               "oorexx", "ooRexx Demo", "Hello, my beloved world!", , , -1)

::requires "dbus.cls"          /* get DBus support  */
```

Code 1: Using the `org.freedesktop.Notifications` *service from ooRexx.*

Running the program of code 1 above on Ubuntu 11.04 yields the notification window as depicted in figure 2.

11 If ooRexx directives are employed, they are led-in with two consecutive colons at the end of a Rexx program. The “::requires” directive demands from the interpreter to run another Rexx program (“package”) which sets up public routines and public classes a program depends on, before executing the program.

12 Please note that it is possible to leave out arguments altogether. The marshalling of the ooRexx D-Bus binding will take care of missing arguments and create matching “empty” values according to the signature: for boolean and numeric types the default value used will be 0, for strings the empty string “”, for containers empty arrays, for maps empty maps, and for variants the type 's' (string) with the value “” (empty string).

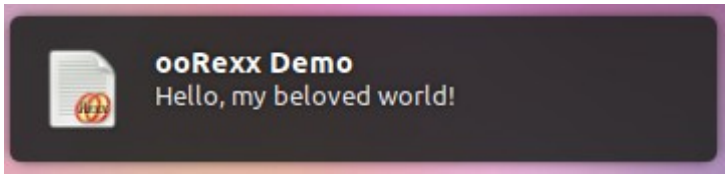

Figure 2: Notification window.

To use the connection's method `message` can be quite challenging, as it mandates that one is acquainted with defining D-Bus signatures. Nevertheless, this method allows to send D-Bus messages in which each argument can be explicitly set, allowing e.g. to use methods which have different casing[13] and the like.

Sometimes the documentation of D-Bus service objects requires variant values to be marshalled according to documented signatures. If there is a need to marshal variant arguments with a predefined signature, then the ooRexx programmer can take advantage of the public routine `DBus.Box(signature,value)`[14] that is part of the ooRexx language binding for D-Bus.

3.2 ooRexx Proxy for Remote Service Objects

Following the Rexx philosophy of "human orientation" it would be desirable, that ooRexx programmers need to supply signatures only if variants data types mandate them. This would mean that all arguments and return values need to be known by an ooRexx D-Bus proxy, such that it can always supply the correct signatures automatically.

Also, if interfacing with D-Bus properties it would be nice, if by default ooRexx programmers could treat them as if they were ooRexx attributes[15], rather than being forced to use the D-Bus `org.freedesktop.Properties` interface.

The ooRexx class `DBusProxyObject` supplies these camouflaging services. It will automatically introspect the remote object and remember all of its interface definitions. Later, when the ooRexx programmer sends messages to the remote D-Bus service object, no type information needs to be supplied as this is already known by the proxy object.

The instance method `getObject(busName,objectPath)` of the class `DBus` returns an ooRexx proxy object for the remote D-Bus service object which can be addressed as a normal ooRexx object.

Code 2 below is comparable in function to code 1 above. One can observe that there is no need for signatures anymore. Also, adhering to ooRexx caselessness, the method's name is not spelled out

13 According to the D-Bus specifications it should not be the case that members of a particular service's interface are spelled with the same characters, but with different case, although one can find such D-Bus services sometimes.

14 In ooRexx a dot is regarded to be a "normal" character, such that symbols may contain one ore more dots.

15 ooRexx attributes can be queried by merely sending the name of the attribute to the object, and can have values assigned to them with the following pattern: `"object~attributeName=newValue"`.

with the correct case, yet the correct method in the service object's interface will be addressed, utilizing the received introspect data “behind the curtains”.

```
        /* get access to remote object */
o=.dbus~session~getObject("org.freedesktop.Notifications","/org/freedesktop/Notifications")

id=o~notify("An ooRexx App", , "oorexx", "ooRexx Demo", "Hello, my beloved world!", , , -1)

::requires "dbus.cls"      /* get DBus support  */
```

Code 2: Using the `org.freedesktop.Notifications` *service via an ooRexx proxy.*

In the extremely rare case that a service object may have different interfaces in which members carry identical names, then the ooRexx proxy support might use the wrong interface to carry out the respective message call. For such situations there is a simple solution available: use the fully qualified name of the desired method in the remote service object, i.e. use the interface name and concatenate it with a dot with the desired member name. Code 3 below demonstrates this technique.

```
        /* get access to remote object */
o=.dbus~session~getObject("org.freedesktop.Notifications","/org/freedesktop/Notifications")

id=o~org.freedesktop.Notifications.Notify("An ooRexx App", , "oorexx", "ooRexx Demo", -
                                          "Hello, my beloved world!", , , -1)

::requires "dbus.cls"      /* get DBus support  */
```

Code 3: Using the `org.freedesktop.Notifications` *service via an ooRexx proxy, fully qualifying the desired method name.*

3.3 ooRexx Listener for D-Bus Signals

The ooRexx class `DBusListenerObject` is a wrapper class that stores the ooRexx object to which signal messages should get forwarded to (attribute `listenerObject`). In addition it allows for storing optionally an interface name (attribute `interface`) and/or a signal name (attribute `signalName`), which would get used as additional filters.

The method `listener` in the connection class `DBus` wraps an ooRexx listener object in an instance of the `DbusListenerObject` class. And if the arguments `interface` and/or `signalName` were supplied, will set these values as well.

Code 4 below depicts an ooRexx program, which connects to the session message bus daemon and adds an ooRexx listener object to that connection (method `listener`), which will start a message loop to run on a separate thread that looks for signal messages that then will get forwarded to the registered listener objects. The DBus broker is asked to forward all signal messages that are of type signal (cf. message `match` and the match rule argument `“type='signal'”`, cf. [5]). This way this simple listener can be used to discover all signal messages that are being broadcast via the session

```
signal on halt          -- intercept ctl-c (jump to label 'halt:' below)

conn=.dbus~session      -- get the "session" connection
conn~listener("add", .rexxSignalListener~new)   -- add the Rexx listener object
conn~match("add", "type='signal'", .true)       -- ask for any signal message

say "Hit enter to stop listener..."
parse pull answer       -- wait for pressing enter

halt:                   -- a label for a halt condition (ctl-c)
   say "closing connection."
   conn~close           -- close connection, stops message loop

::requires "dbus.cls"      -- get dbus support for ooRexx

::class RexxSignalListener -- just dump all signals/events we receive
::method unknown           -- this method intercepts all unknown messages
  use arg methName, methArgs

  slotDir=methArgs[methArgs~size]     -- last argument is slotDir
  say "-->" pp(slotDir~messageTypeName) pp(slotDir~interface) -
            pp(slotDir~member)", nrArgs="methArgs~items-1

  if methArgs~items>1 then
  do
     do i=1 to methArgs~items-1
        say "    argument #" i":" pp(methArgs[i])
     end
  end
  say "-"~copies(79)

::method NameOwnerChanged  -- demo how to directly intercept a signal
  use arg name, old, new, slotDir
  say "==> NameOwnerChanged:" "Name:" pp(name)", OldOwner:" pp(old) ))  -
                              ", NewOwner:" pp(new)
  say "-"~copies(79)

::routine pp      -- "pretty print": enclose string value with square brackets
  parse arg value
  return "["value"]"
```

Code 4: A simple ooRexx listener for D-Bus signal messages.

message bus daemon. Running the program in its own command line window on Ubuntu 11.04 may yield output as shown in figure 3.[16]

3.4 ooRexx D-Bus Service Object

Creating service objects for D-Bus is done by employing the method `serviceObject` of the connection class `DBus`, which expects an object path and an ooRexx object fulfilling the role of a service. A working ooRexx service object is shown in code 5 below, which makes two methods available: `Introspect` and `Multiply`.

16 The output reflects the following actions of the user after starting listening for all session D-Bus signals in a listener terminal window: switch to another terminal window, start a service, end the service immediately, switch back to the listener terminal window.

```
rony@rony-linux:/mnt/root_f/dbusoorexx/doc/code$ rexx signalListener.rex
Hit enter to stop listener...
--> [signal] [org.freedesktop.DBus] [NameAcquired], nrArgs=1
    argument # 1: [:1.481]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [0]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.matcher] [ActiveWindowChanged], nrArgs=2
    argument # 1: [/org/ayatana/bamf/window67113036]
    argument # 2: []
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [0]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.matcher] [ActiveApplicationChanged], nrArgs=2
    argument # 1: [/org/ayatana/bamf/application311805604]
    argument # 2: []
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [1]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.matcher] [ActiveWindowChanged], nrArgs=2
    argument # 1: []
    argument # 2: [/org/ayatana/bamf/window67127770]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [1]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.matcher] [ActiveApplicationChanged], nrArgs=2
    argument # 1: []
    argument # 2: [/org/ayatana/bamf/application311805604]
--------------------------------------------------------------------------------
==> NameOwnerChanged: Name: [:1.482], OldOwner: [], NewOwner: [:1.482]
--------------------------------------------------------------------------------
==> NameOwnerChanged: Name: [org.rexxla.oorexx.demo.Hello], OldOwner: [], NewOwner: [:1.482]
--------------------------------------------------------------------------------
==> NameOwnerChanged: Name: [org.rexxla.oorexx.demo.Hello], OldOwner: [:1.482], NewOwner: []
--------------------------------------------------------------------------------
==> NameOwnerChanged: Name: [:1.482], OldOwner: [:1.482], NewOwner: []
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [1]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.matcher] [ActiveWindowChanged], nrArgs=2
    argument # 1: [/org/ayatana/bamf/window67127770]
    argument # 2: [/org/ayatana/bamf/window67113036]
--------------------------------------------------------------------------------
--> [signal] [org.ayatana.bamf.view] [ActiveChanged], nrArgs=1
    argument # 1: [0]
--------------------------------------------------------------------------------

closing connection.
```

Figure 3: Possible output of the D-Bus signal listener program of code 4 above.

In order for D-Bus programs to be able to take advantage of the services in the ooRexx object, it is necessary to implement the method `Introspect`, which must return a string that contains an XML encoded documentation [12] of the available interfaces. This information will be analyzed by the clients and enables them to discover the features the service object makes available to them.

Also, the service object needs to inform the connection about the object path that addresses that

particular service object (“/org/rexxla/oorexx/demo/Hello”). Lastly, the clients need to be able to address the process in which the service object waits for them. To facilitate easy addressing the service object requests[17] a unique bus name (“service name”, “well-known bus name”) from the session message daemon, named “org.rexxla.oorexx.demo.Hello”.[18]

Figure 4 below shows the output of running the server in code 5 below with the service object servicing one client request and hitting the enter key thereafter to stop the server.

```
busName       ="org.rexxla.oorexx.demo.Hello"
objectPath    ="/org/rexxla/oorexx/demo/Hello"

conn=.dbus~session            -- get the session bus

signal on halt                -- intercept ctl-c (jump to label "halt:")

res=conn~busName("request", busName)   -- request a bus name
if res<>.dbus.dir~primaryOwner & res<>.dbus.dir~alreadyOwner then -- o.k., wait for clients?
do
   say "res="res "problem with requesting the bus name" busName", aborting ..."
   exit -1
end

service=.HelloRexxService~new -- create an instance of the Rexx service
conn~serviceObject("add", objectPath, service)  -- add service object to connection

say "Hit enter to stop server..."
parse pull answer

halt:                   -- a ctl-c causes a jump to this label
   say "closing connection."
   conn~close           -- close connection, stops message loop


::requires "dbus.cls"         -- get dbus support for ooRexx

::class HelloRexxService

::method Introspect        /* return the introspection data for this service object */
 return '<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"'   -
        '"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">      '   -
        '<node>                                                               '   -
        '  <interface name="org.freedesktop.DBus.Introspectable">             '   -
        '    <method name="Introspect">                                       '   -
        '      <arg name="data" direction="out" type="s"/>                    '   -
        '    </method>                                                        '   -
        '  </interface>                                                       '   -
        '  <interface name="org.rexxla.oorexx.demo.Hello">                    '   -
        '    <method name="Multiply">                                         '   -
        '      <arg name="number1" direction="in"  type="d"/>                 '   -
        '      <arg name="number2" direction="in"  type="d"/>                 '   -
        '      <arg name="result"  direction="out" type="d"/>                 '   -
        '    </method>                                                        '   -
        '  </interface>                                                       '   -
        '</node>                                                              '

::method Multiply       -- implementation of "orx.rexxla.oorexx.demo.Hello.Multiply"
   use arg number1, number2, slotDir
   say "Multiply-request received: sender-bus=["slotDir~sender"]" "at=["slotDir~dateTime"]"
   return number1*number2
```

Code 5: ooRexx object used as a D-Bus service object.

```
rony@rony-linux:/mnt/root_f/dbusoorexx/doc/code$ rexx service.rex
Hit enter to stop server...
Multiply-request received: sender-bus=[:1.495] at=[2011-11-29T18:56:17.958842]


closing connection.
```

Figure 4: Output of running the D-Bus server in code after servicing one client request.

A client being brokered by a D-Bus message daemon needs the following information up-front to successfully get in contact and interact with a service object:

- the service name (well-known bus name), and
- the object path.

```
conn=.dbus~connect("session") -- connect to the "session" bus

busName       ="org.rexxla.oorexx.demo.Hello"
objectPath    ="/org/rexxla/oorexx/demo/Hello"

o=conn~getObject(busName, objectPath)     -- get the DBus object
num1=12.345
num2=10.01
say num1"*"num2"="o~multiply(num1,num2)   -- use the multiply method

::requires "dbus.cls"         -- get dbus support for ooRexx
```

Code 6: ooRexx client program that is able to interact with the ooRexx service object in code 5 above and code 7 below.

Code 6 above depicts the client code that interacts with the service object. The `getObject` method of class `DBus` will cause the introspection of the service object for its interfaces and have them set up "behind the curtain", such that one is immediately able to interact with the service object by sending it messages. The output of running the client code in a client terminal window is shown in figure 5 below.

```
rony@rony-linux:/mnt/root_f/dbusoorexx/doc/code$ rexx client.rex
12.345*10.01=123.57345
```

Figure 5: Output of running the ooRexx client program in code 6 above serviced either by the service object in code 5 or code 7.

Code 7 below shows a variant of the service object program as depicted in code 5 above. This variant subclasses the `DBusServerObject` class. Instead of supplying an `Introspect` method of its own, it forwards its introspect data[19] to the superclass in its `init` constructor method, which allows the superclass to answer `Introspect` messages.

In addition this program demonstrates how an ooRexx programmer can support discovery of the

19 It is possible to supply a path to a file instead that contains the introspect data.

```
busName       ="org.rexxla.oorexx.demo.Hello"
objectPath    ="/org/rexxla/oorexx/demo/Hello"

conn=.dbus~session            -- get the session bus

signal on halt                -- intercept ctl-c (jump to label "halt:")

res=conn~busName("request", busName)   -- request a bus name
if res<>.dbus.dir~primaryOwner & res<>.dbus.dir~alreadyOwner then -- o.k., wait for clients?
do
   say "res="res "problem with requesting the bus name" busName", aborting ..."
   exit -1
end

service=.HelloRexxService~new -- create an instance of the Rexx service
conn~serviceObject("add", objectPath, service)  -- add service object to connection

.IDBusPathMaker~publishAllServiceObjects(conn)  -- allow discovery of serviced object paths

say "Hit enter to stop server..."
parse pull answer

halt:                   -- a ctl-c causes a jump to this label
   say "closing connection."
   conn~close           -- close connection, stops message loop


::requires "dbus.cls"      -- get dbus support for ooRexx

::class HelloRexxService subclass DBusServiceObject

::method init              -- constructor
   -- define introspect data; instead one could supply the full path to an introspect file
 idata= '<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"' -
        '"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">      ' -
        '<node>                                                               ' -
        '  <interface name="org.freedesktop.DBus.Introspectable">             ' -
        '    <method name="Introspect">                                       ' -
        '      <arg name="data" direction="out" type="s"/>                    ' -
        '    </method>                                                        ' -
        '  </interface>                                                       ' -
        '  <interface name="org.rexxla.oorexx.demo.Hello">                    ' -
        '    <method name="Multiply">                                         ' -
        '      <arg name="number1" direction="in"  type="d"/>                 ' -
        '      <arg name="number2" direction="in"  type="d"/>                 ' -
        '      <arg name="result"  direction="out" type="d"/>                 ' -
        '    </method>                                                        ' -
        '  </interface>                                                       ' -
        '</node>                                                              '
  self~init:super(idata)   -- let superclass (DBusServiceObject) initialize

::method Multiply          -- implementation of "orx.rexxla.oorexx.demo.Hello.Multiply"
   use arg number1, number2, slotDir
   say "Multiply-request received: sender-bus=["slotDir~sender"]" "at=["slotDir~dateTime"]"
   return number1*number2
```

Code 7: ooRexx service object subclassing `DBusServiceObject`.

registered object paths, by employing a utility class, `IDBusPathMaker`, from the ooRexx D-Bus language binding: it is sufficient to send the message `publishAllServiceObjects(conn)` to the `IDBusPathMaker` class, which will make all registered service object paths researchable[20].

20 Depending on the D-Bus service there may be multiple object paths available. In order to discover them, the D-Bus service will add node-information when it returns the introspect data, which can be exploited to find all published object paths.

3.5 Private D-Bus Server

The D-Bus communication infrastructure can be used without a message bus daemon serving as a broker of D-Bus messages and manager of the D-Bus setup. The ooRexx language binding for D-Bus includes the class `DBusServer` which can be used to establish a private server. Methods 3 below shows the most important methods of that ooRexx class.

```
new( server_address [,[defaultService] [,[defaultListener] [,allowAnonymous=.false]]] )

allowAnonymous  /* attribute */
connections     /* attribute (get only), returns an array of open client connections */
defaultService  /* attribute */
serverAddress   /* returns server's address with GUID */
shutdown
startup
```

Methods 3: The most important methods of class "DBusServer".

The private D-Bus server does no message relaying/brokering, nor does it assign unique names to bus connections and the like. The `org.freedesktop.DBus` interfaces with their functionalities are simply not available in this scenario.[21] Nevertheless it may be helpful to quickly create client/server applications using the D-Bus interprocess communication infrastructure for exchanging messages. If using the `tcp` transport then even processes on different computers, running on different operating systems etc. can communicate with the private D-Bus server.

When creating an instance of the `DBusServer` class one needs to supply a D-Bus supported address[22] to which clients can connect to. It is advisable to supply a default service object (optional), which gets assigned to each connected client by default to serve its requests. If the application mandates for it, one is also able to supply a default listener object that gets assigned to each accepted client connection.[23]

Code 8 below denotes a private D-Bus server implementation, employing a `tcp` transport, which allows D-Bus clients to be served across the Internet. In this particular case the host is defined to be `"localhost"`, such that only client processes on the same computer can interact with it.[24] Figure 7 below shows the output, after the server serviced one client request and got shutdown thereafter. As can be seen from the output the sender-bus name of the received message call is empty, which is to be expected, if using a private D-Bus server as there is no broker (message daemon) that would assign unique bus names and servicing requests for bus names from clients.

21 If one wishes to set up message daemons/brokers with private addresses then this is possible by utilizing the program `dbus-daemon` which is documented with man pages on Linux.

22 The D-Bus specification defines the possible formats for addresses, cf. [5].

23 It is possible to supply the same ooRexx object as a service and as a listener object. Unknown messages will then have to be handled with a common `UNKNOWN` method. This also makes it possible to communicate via attributes being used as control variables in ooRexx `GUARD` keyword statements.

24 Replacing `"localhost"` with an IP address or a known host name allows the interaction of different computers.

```
   -- set up address and define default service to be added to client connections
address="tcp:host=localhost,port=23000,family=ipv4;"
defaultService =.HelloRexxService~new
server =.DBUSServer~new(address,defaultService)
server~startup          -- start up the private server

say "server~serverAddress:" pp2(server~serverAddress)

signal on halt          -- intercept ctl-c (jump to label "halt:")

say "Hit enter to stop server..."
parse pull answer

halt:                   -- a ctl-c causes a jump to this label
   say "shutdown server, closing all connections to clients..."
   server~shutdown      -- close connection, stops message loop


::requires "dbus.cls"   -- get dbus support for ooRexx

::class HelloRexxService subclass DBusServiceObject

::method init           -- constructor
   -- define introspect data; instead one could supply the full path to an introspect file
 idata= '<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"'  -
        '"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">      '  -
        '<node>                                                               '  -
        '  <interface name="org.freedesktop.DBus.Introspectable">             '  -
        '    <method name="Introspect">                                       '  -
        '      <arg name="data" direction="out" type="s"/>                    '  -
        '    </method>                                                        '  -
        '  </interface>                                                       '  -
        '  <interface name="org.rexxla.oorexx.demo.Hello">                    '  -
        '    <method name="Multiply">                                         '  -
        '      <arg name="number1" direction="in"  type="d"/>                 '  -
        '      <arg name="number2" direction="in"  type="d"/>                 '  -
        '      <arg name="result"  direction="out" type="d"/>                 '  -
        '    </method>                                                        '  -
        '  </interface>                                                       '  -
        '</node>                                                              '
  self~init:super(idata)   -- let superclass (DBusServiceObject) initialize

::method Multiply          -- implementation of "orx.rexxla.oorexx.demo.Hello.Multiply"
   use arg number1, number2, slotDir
   say "Multiply-request received: sender-bus=["slotDir~sender"]" "at=["slotDir~dateTime"]"
   return number1*number2
```

Code 8: A private D-Bus server.

```
rony@rony-linux:/mnt/root_f/dbusoorexx/doc/code$ rexx privateService.rex
server~serverAddress: [tcp:host=localhost,port=23000,family=ipv4,guid=e56ccba92af36bfa265adb890006a754]
Hit enter to stop server...
Multiply-request received: sender-bus=[] at=[2011-11-29T19:22:11.091993]

shutdown server, closing all connections to clients...
```

Figure 6: Output of running code 8 above.

Code 9 below shows the client code, that connects to the private server using the address that the server listens to. Figure 7 below shows the output of running the client in a terminal window.

```
address="tcp:host=localhost,port=23000,family=ipv4;"
conn=.dbus~connect(address)    -- connect to given address

busName       ="org.rexxla.oorexx.demo.Hello"
objectPath    ="/org/rexxla/oorexx/demo/Hello"
o=conn~getObject(busName, objectPath)     -- get the DBus object

num1=12.345
num2=10.01
say num1"*"num2"="o~multiply(num1,num2)   -- use the multiply method

::requires "dbus.cls"         -- get dbus support for ooRexx
```

Code 9: D-Bus private client.

```
rony@rony-linux:/mnt/root_f/dbusoorexx/doc/code$ rexx privateClient.rex
12.345*10.01=123.57345
```

Figure 7: Output of running code 9 above.

3.6 D-Bus On-the-Fly-Documentation (“dbusdoc.rex”)

While researching D-Bus and the documentation of the D-Bus interfaces of D-Bus service objects it turned out that it was rather hard to locate suitable documentation. As a result it was very difficult, if not impossible at times, to interact with running D-Bus services as for the programmer the interfaces and signatures were not known.

On Linux there are many D-Bus services that are running and startable for the system and session message bus daemons, but there is no overview and documentation of the object paths and interfaces these offer.

In order to overcome this situation, which hinders programmers to experiment with D-Bus services, a little utility got created while developing the ooRexx D-Bus language binding, an ooRexx program named “`dbusdoc.rex`”. It is part of the distribution of the ooRexx D-Bus language binding.

This command line tool lists all available service names[25], if no argument is given, and finally gives its syntax: `dbusdoc.rex [session|system] [service.bus.name]`. If the first argument is omitted then the session message daemon is addressed by default.[26]

This on-the-fly documentation tool will create a HTML rendering of the found interface definitions and stores them as a file. The tool uses object path discovery, introspects each interface and documents the found artefacts in a style, that is hoped to be easily legible and comprehensible.[27]

25 “service name” and “bus name” are used as synonyms.
26 It may be the case that one is not allowed to send messages to the system message daemon for security reasons, which will inhibit the on-the-fly-documentation of system service names.
27 A popular interactive D-Bus debugger on Linux is called "D-Feet", cf. [13].

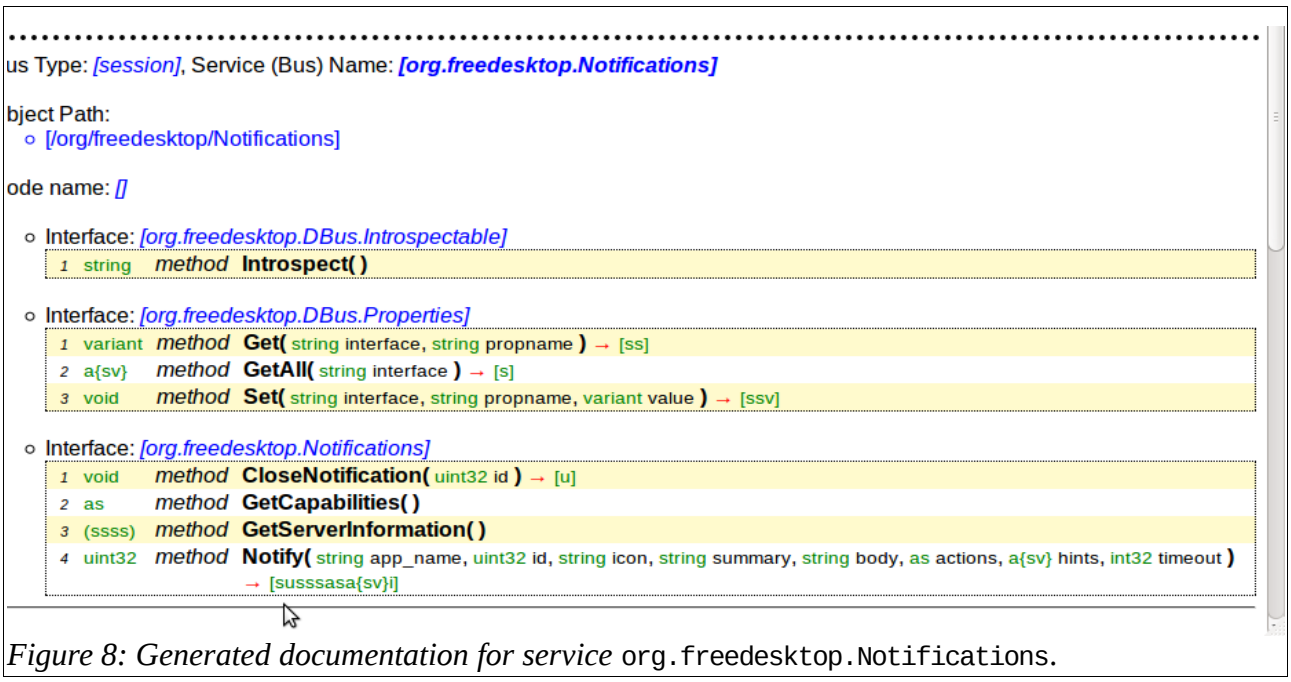

Figure 8: Generated documentation for service `org.freedesktop.Notifications`.

In order to minimize the documentation of the D-Bus services, the tool determines which object paths of a specific service return the same interface definitions and groups these object paths in alphabetical order to ease locating them and saving space for the interface documentation. In general all interface names as well as their members are sorted alphabetically within (each group of object paths) to ease locating them.

Figure 8 above displays the on-the-fly documentation created on Ubuntu 11.04 for the service name `org.freedesktop.Notifications` that was used to create the programs in code 1 (p. 11) and code 2 (p. 13) above. Please note that it is possible that running this documentation tool on different Linux distributions analyzing the same service names may yield different documentation, if the services document their interfaces differently.[28]

4 Roundup and Outlook

This article introduced the ooRexx language bindings for D-Bus, aimed at ooRexx programmers who may have never worked with D-Bus. Therefore the general D-Bus concepts got introduced, which then got exploited with the ooRexx language binding.

In order for ooRexx programmers to get acquainted quickly with the infrastructure, fully functional nutshell ooRexx examples are given and briefly explained. They are meant to serve as prototypes, which should make it easy for ooRexx programmers to take immediate advantage of it. As a result ooRexx programmers can now participate in D-Bus communications and create D-Bus client/server

28 It may even be the case that on different platforms the same services implement different sets of interfaces and members.

applications that take advantage of all of D-Bus' interprocess communication infrastructure.

A full documentation and discussion of the ooRexx classes and public routines that are part of the language binding and made available via the ooRexx package "dbus.cls" should be created. Depending on the interest from the ooRexx and/or D-Bus communities also the architecture and applied strategies for the ooRexx language binding could be documented (beyond the comments in the source code) or presented in order to make it available to a broader discussion.

5 Acknowledgements

The author wishes to thank the following D-Bus developers who helped by answering implementation questions on the D-Bus developer mailing list while developing the language binding from scratch (in alphabetical order):

- Mike Gorse,
- Simon McVittie, and
- Havoc Pennington.

6 References

[1] "D-Bus", Wikipedia (as of 2011-11-29): http://en.wikipedia.org/w/index.php?title=D-Bus&oldid=445173447

[2] Homepage of "freedesktop.org" (as of 2011-11-29): http://www.freedesktop.org

[3] Homepage of the D-Bus project (as of 2011-11-29): http://dbus.freedesktop.org

[4] Overview of D-Bus language bindings (as of 2011-11-29): http://www.freedesktop.org/wiki/Software/DBusBindings

[5] Specifications of D-Bus (as of 2011-11-29): http://dbus.freedesktop.org/doc/dbus-specification.html

[6] Low-level API documentation of D-Bus (as of 2011-11-29): http://dbus.freedesktop.org/doc/api/html/index.html

[7] List of some D-Bus projects (as of 2011-11-29): http://www.freedesktop.org/wiki/Software/DbusProjects

[8] Homepage of "Open Object Rexx (ooRexx)" (as of 2011-11-29): http://www.ooRexx.org

[9] Flatscher R.G.: "Resurrecting Rexx, Introducing Object Rexx", ECOOP Workshop on "Revival of Dynamic Languages (RDL)", Nantes 2006. URL (as of 2011-11-29): http://wi.wu.ac.at/rgf/rexx/misc/ecoop06/ECOOP2006_RDL_Workshop_Flatscher_Paper.

pdf

[10] Homepage of the “Rexx Language Association (RexxLA)” (as of 2011-11-29): http://www.RexxLA.org

[11] Homepage of the Java language binding for ooRexx (“BSF4ooRexx”, as of 2011-11-29): http://sourceforge.net/projects/bsf4oorexx/

[12] Document type definition (DTD) for D-Bus introspect text (as of 2011-11-29): http://standards.freedesktop.org/dbus/1.0/introspect.dtd

[13] Homepage of D-Feet, a D-Bus debugger (as of 2011-11-29): http://live.gnome.org/DFeet/