Testing and Exploiting the ooRexx DBus Binding

An ooTest and Use Case Assessment

Author: Mag. Sebastian Margiol, MSc
Advisor: Univ-Prof. Mag. Dr. Rony G. Flatscher

Vienna, May 2015

Table of Contents

Illustration Index

Script Example Index

Table Index

Abstract

DBus is a powerful message-broker system that enables easy-to-use interprocess communication between different programs that might be written in different programming languages, run on different machines or even run on different operating systems. These powerful features demand a solid specification that exactly defines how a successful message transaction is carried out.

DBus is now an integrated part of almost every modern Linux distribution. It enables a programmer to programming-language independently orchestrate different programs, therefore offering features like broadcasting and receiving simple messages like signals, providing services and handle properties. Access to DBus is realized through a so called DBus-language-binding. A language-binding tries to bring DBus interaction in line with the concepts of the programming language and also enables to circumvent the strict object type definition, DBus demands. The application of DBus functionality with an appropriate binding should come as natural as possible. This certainly imposes a challenge for the language binding. This paper defines three aims. Firstly create comprehensive test-cases and make assertions by using the ooRexx's JUnit equivalent, ooTest. The goal is to cover each functionality with an appropriate sample. Secondly, draw attention to some specifics in the application of DBusooRexx by analyzing input and output of selected statements. And finally, test the ease of use of the language binding in a practical setting.

For the last part, nutshell examples are provided that are designed to be useful on their own, serve as templates for other applications and cover as many different concepts of DBus whilst demonstrating some powerful features. Due to the assessment with ooTest and the interplay with other DBus services some bugs were revealed, reported and already corrected by the author of DBusooRexx.

Kurzfassung

DBus ist ein mächtiges Nachrichten-Vermittlungs-System, dass es ermöglicht, eine Interprozess-Kommunikation zwischen unterschiedlichen Programmen herzustellen. Dabei ist es unerheblich, ob die Programme in unterschiedlichen Programmiersprachen implementiert wurden, auf unterschiedlichen Maschinen oder sogar auf unterschiedlichen Betriebssystemen ausgeführt werden. Diese Funktionalität benötigt eine solide Spezifikation die eine erfolgreiche Nachrichtenübermittlung exakt festlegt.
DBus ist mittlerweile ein fester Bestandteil beinaher jeder modernen Linux-Distribution. Es ermöglicht, programmiersprachenunabhängig Anwendungen zu orchestrieren. Dafür werden Funktionen wie das Senden und Empfangen von Signalen, die Bereitstellung von Funktionen und die Verwaltung von Attributen (Properties) zur Verfügung gestellt.
Zugriff auf die Funktionalität von DBus wird durch sogenannte Sprachanbindungen bewerkstelligt. Eine Sprachanbindung versucht, DBus-Nachrichten in Einklang mit Konzepten der Programmiersprache zu bringen und ermöglicht darüber hinaus die strikte Typisierung, die von DBus verlangt wird, zu umgehen. Die Anwendung von DBus-Funktionalität sollte mithilfe einer entsprechenden Sprachanbindung so natürlich wie möglich erscheinen. Dies stellt dementsprechend eine umfangreiche Herausforderung für die Sprachanbindung dar. Diese Arbeit verfolgt im wesentlichen drei Ziele: Erstens werden umfangreiche Testfälle definiert und mithilfe von ooRexxUnit, einem Äquivalent zu JUnit, untersucht. Ziel ist es dabei, jede Funktionalität mit einem entsprechenden Beispiel abzudecken. Zweitens werden einige Besonderheiten, die während der Testfälle aufgetreten

sind, näher untersucht, indem Eingabe und Ausgabewerte dargestellt werden. Zuguterletzt wird die Einfachkeit der Anwendung mit kurzen Beispiel-skripten demonstriert. Diese Skripte sind mit der Absicht erstellt wurden möglichst viele unterschiedliche Funktionen von DBus anzuwenden und für sich selbst sowie als Grundlage für andere Skripte nützlich sein. Durch die Überprüfung mithilfe von ooTest und der Interaktion mit anderen DBus-services wurden Fehler entdeckt, gemeldet und vom Autor von DBusooRexx, bereits behoben.

1. Introduction

In an earlier paper, ooRexx was already used to establish connections to the DBus. (Margiol, 2011) As there was no language-binding available for ooRexx, an indirect approach was chosen using BSF4ooRexx in order to make Java functionality available for ooRexx and consequently utilizing Java's DBus binding. Although this approach is feasible, it has certain drawbacks. Java and DBus are both strictly typed, that means every single object type has to be defined. In the case of more complicated object types like structs, it is necessary to create a Java class and compile it, just to provide the object-type which can then be filled with values. Although writing Java classes and compile them over command line instructions works, it contradicts the concept of scripting languages, especially for ooRexx, as taking care about an object type comes unnatural. This further hinders the easiness of usage. Another major drawback is that very much code is generated for simple tasks. A simple example demonstrated that using Java mapped DBus functionality for ooRexx needs three Java classes defined, each containing more instructions than the same example implemented in three lines of python code.[1]

It was already mentioned that ooRexx programmers should not abandon the advantages of DBus interaction, but if a direct binding is available, the Java approach can gladly be passed back. (Margiol, 2011) Professor Rony Flatscher (Vienna University of Economics), who already contributed much to the ooRexx community finally wrote a DBus language-binding for ooRexx, called DBusooRexx. (Flatscher, 2011a)

The new member of the language-binding family offers many powerful features. One of the most attracting features certainly is the object type “guessing” capability. A DBus object type can therefore be created as easily as if it is a regular ooRexx object type.

The aim of this paper is not to provide an introduction into DBus concepts in general as there is already some excellent documentation available. The DBus specification is probably the most comprehensive documentation. (Pennington et al., 2014) The wiki page of freedesktop.org provides Links to many documents about general introduction to DBus, documents about the reference implementation as well as articles and tutorials about some language-bindings. The FAQ subpage provides explanation of how DBus differs from other IPC mechanisms like CORBA and DCOP. The introduction to the language-binding for ooRexx (Flatscher, 2011a) and the presentation slides (Flatscher, 2011b) are compulsory readings, if heading into DBusooRexx as well. A comprehensive book about Rexx and ooRexx can be recommended as well, especially if some aspects of the programming language are unclear and the programming reference of ooRexx does not provide sufficient information. (Flatscher, 2013)

This document is structured as following: In the next section everything is described that is necessary to setup the environment for using DBusooRexx. The section is divided into mandatory and optional installation packages, the latter are useful for creating own scripts, but need not necessarily be installed for using DBusooRexx. The next section copes ooTest, a toolset for creating automated testcases. The creation of comprehensive testcases is intended to be one of the main contributions of this work, therefore the testing framework is introduced and the creation of the resulting testgroup is described step by step. Some of the testgroup's testmethods are elaborated in more detail to point attention to specialties in the application of DBusooRexx and DBus (for example handling .nil values).

The last section of this document is dedicated to the usage of DBus in a practical setting,

1 DBusooRexx code for the same example would also be about three lines.

therefore usecases are defined and DBusooRexx solutions are provided. The intention of these scripts is not only to offer examples and templates, but also to demonstrate that small, yet feature-rich programs can easily be developed with the help of a powerful system like DBus and the appropriate “translator” DBusooRexx.

As this document contains some program code that might disrupt the readability, following syntax is defined: Every program's input or output is displayed with a mono-space font. That is also true for some longer program names. If the program code refers to an input or any other related DBus command, it is displayed with a yellow background. If the code snippet depicts an output or a result of an invocation, it is colored with gray background. See following examples: `input, plain and` **`bold`** and `output, plain and` **`bold`**. Script examples do additionally use colored keywords. (common programming language instructions for the given language.)

For the rest of this document the term DBusooRexx relates to the DBus language binding for ooRexx. The term testgroup refers to the ooTest file that contains all test cases.

The following section describes all components that have to be installed prior to access DBus via ooRexx and also suggests software that might probably be helpful during the development of DBus applications.

1.1 Mandatory Installation Packages

Being a long back and forth to settle for a unified inter-process-communication system on different Linux-distributions, DBus is now in use by any modern Linux distribution by default. Therefore it is very unlikely that DBus is not installed and already running if using a Linux system.[2] The most current version at the time of writing was dbus-1.8.6 which was released on 2 July 2014.[3] If not already installed, the Software repository will much likely offer an installation candidate for dbus per default. On a current Ubuntu version (14.04), DBus version 1.6.18-0ubuntu4.1 (trusty-update) is used. Although this version represents the legacy branch of DBus and is only supported for security fixes, it is common for stable distributions to provide this version.[4] DBus, being part of the `freedesktop.org` software project is dedicated to create interoperability between different (Linux/Unix) operating systems. MacOSX and Windows are supported as well what further enhances this claim. The Windows port is hardly documented, on the project homepage it is mentioned that two former projects (dbus4win and windbus) merged into the development branch of DBus.[5] You can obtain and compile the source code, use the binary from sourceforge[6] (latest update in April 2013), or use the binary provided through DBusooRexx. The DBusooRexx installation package ships support for all common operating systems in both, 32 and 64 Bitness. As the low-level implementation of DBUS requires a XML parser that is probably not readily available on Windows operating systems (expat)[7], Rony Flatscher also compiled and attached

2 Availability of a running DBus-session can be screened by issuing the command `dbus-monitor` in a shell.

3 Repository of all DBus releases so far: http://dbus.freedesktop.org/releases/dbus/, accessed on 2 September 2014.

4 Change-logs of any DBus version can be found at http://upstream-tracker.org/versions/dbus.html, accessed on 2 September 2014.

5 Information about Windows DBUS port: http://www.freedesktop.org/wiki/Software/dbus/, accessed on 2 September 2014.

6 Download location of DBUS binaries for Windows: http://sourceforge.net/projects/windbus/ , accessed on 2 September 2014.

7 Project Homepage of libexpat: http://www.libexpat.org (only 32-Bit version available), accessed on 2 September 2014.

it for 32-Bit (libexpat.dll) and for 64-Bit (expat.dll) Windows versions.[8] In contrast to Linux systems where DBus is started by default, Windows systems need DBus to be started manually. This can eighter be done by starting the daemon with the command `dbus-daemon.exe` and the switches `--session` and `--print-address` or by starting any application that requires DBus and therefore initializes it. (Slower operating systems might run into a timeout on the first try) If starting the DBusDemon manually following information is provided:

`tcp:host=localhost, port=8640,family=ip4,guid=07522f56431e30fca59f041b53f62a7c`.

This information indicates that DBus is running and is reachable. There are some switches available that allow to specify the connection furthermore. The command `dbus-daemon.exe --help` lists all available options.

Of course ooRexx is a mandatory installation candidate. Installation packages for different operating systems can be found on the ooRexx homepage.[9] At the time of writing the most current release on this page was version 4.2.0. The Open Object Rexx download section section[10] of the open-source-archive sourceforge.net provides different ooRexx versions and other ooRexx related projects as well.

The minimum requirement for DBusooRexx is ooRexx version 4.2.0. The release note indicates that version 4.2.0 is a new feature and bug fix release[11], you should therefore install this version or any higher (if available) for DBusooRexx. Pick the version that best matches your Linux distribution[12], respectively Windows version. Be careful that ooRexx is installed in the same bitness, the operating system has.

The next step is to install the ooRexx DBus binding, DBusooRexx[13]. The installation script is written in Rexx and is called `install_ooRexx_dbus.rex`. If any problem arise during the installation process, it might be useful to consult the troubleshooting section at the end of this document.

After installed correctly, DBus has to be made available for ooRexx by adding the `::requires DBUS.CLS` directive to a script.

The next installation package, BSF4ooRexx, is not a mandatory installation for DBusooRexx anymore. BSF4ooRexx enables to provide all Java functionality to ooRexx in an easy way, it is the language-binding to Java if using this analogy to DBusooRexx. An earlier version of DBusooRexx had dependencies on Java's UTF-8 conversion routine, which was accessed through BSF. The current version has a fall-back mode if the utf8-conversion routine is not available through Java. It is nonetheless recommend to install BSF4ooRexx as it provides useful features and one of the script examples presented later, will make use of it.

If all these installations have been conducted, the core setup that is sufficient to explore DBus with ooRexx is ready to be used.

8 Note that DBusooRexx for Windows was not tested extensively at the time of writing. Download location: http://wi.wu.ac.at/rgf/rexx/orx22/work/, accessed on 2 september 2014.

9 ooRexx Homepage: http://www.oorexx.org/download.html, accessed on 2 September 2014.

10 ooRexx Projects: http://sourceforge.net/projects/oorexx/, accessed on 2 September 2014.

11 Changes of ooRexx 4.2: http://sourceforge.net/projects/oorexx/files/oorexx/4.2.0/CHANGES.txt/download , Release Note ooRexx 4.2: http://sourceforge.net/projects/oorexx/files/oorexx/4.2.0/ReleaseNotes/download, accessed on 2 September 2014.

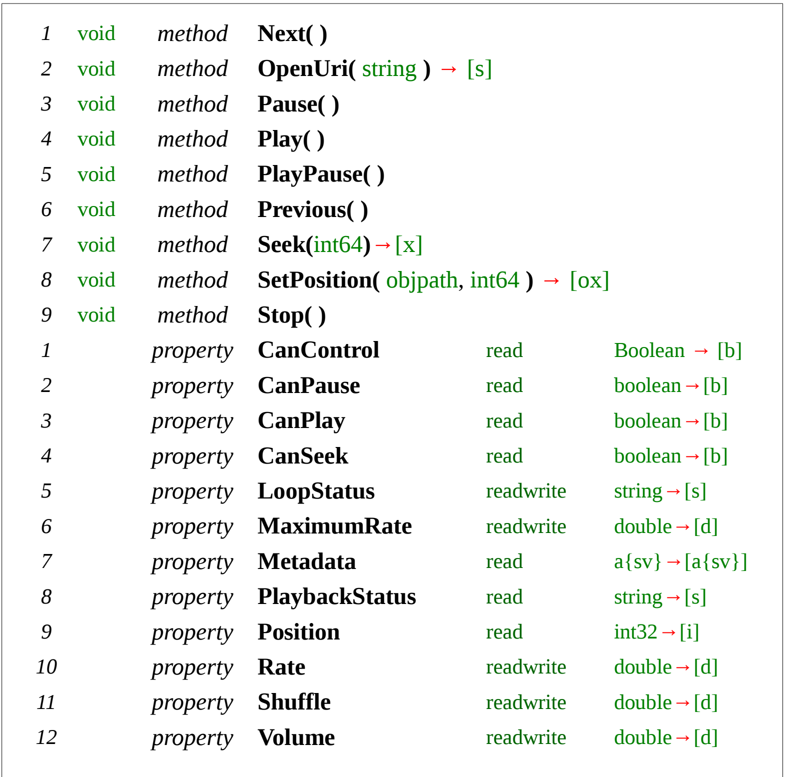
12 The most current version for 64-Bit Ubuntu systems at the time of writing is **`ooRexx-4.2.0-1.ubuntu1310.x86_64.deb`**. The package description denotes the minimum system version, therefore it perfectly fits for an Ubuntu14.04 installation.

13 DBusooRexx and its source-code are available on sourceforge: https://sourceforge.net/projects/bsf4oorexx/files/GA/sandbox/DBusooRexx/, respectively https://sourceforge.net/p/bsf4oorexx/code/HEAD/tree/sandbox/rgf/misc/DBusooRexx/, accessed on 2 September 2014.

1.2 Optional Installation Packages

If testcases are executed or implemented, ooRexxUnit is necessary as well. ooRexxUnit is an ooRexx equivalent to Java's popular JUnit testing framework.
At the time of writing, the most current version of ooRexxUnit was 4.2.0.Snapshot.06, which was updated last time February 2014.[14] Snapshots are packages (zip archives) that contain the testing framework for writing own testgroups and a set full of prepared testgroups, which can be used to test the ooRexx interpreter as well as documentation. It is essential to use a testing framework that is equivalent to the ooRexx version. (Miesfeld, 2009a). If the installation script of ooRexxUnit does not work as expected, try consulting the troubleshooting section at the end of this document.

The next installation candidates are, though optional, essential to be productive during the implementation and testing process of services. They enable quick and easy interaction, in order to test the accessibility and functionality of any DBus service, own created or already available, namely DBus debuggers.
DBusooRexx ships a program that is able to create a beautiful and comprehensive summary of any DBus service, `dbusdoc.rex`. This program creates a HTML file that lists all methods, properties and signals of any DBus service passed as argument. (See Figure 1) This tool provides enough information about any DBus service, if however a graphical, interactive



```
1   void   method     Next( )
2   void   method     OpenUri( string ) → [s]
3   void   method     Pause( )
4   void   method     Play( )
5   void   method     PlayPause( )
6   void   method     Previous( )
7   void   method     Seek(int64)→[x]
8   void   method     SetPosition( objpath, int64 ) → [ox]
9   void   method     Stop( )
1          property   CanControl       read        Boolean → [b]
2          property   CanPause         read        boolean→[b]
3          property   CanPlay          read        boolean→[b]
4          property   CanSeek          read        boolean→[b]
5          property   LoopStatus       readwrite   string→[s]
6          property   MaximumRate      readwrite   double→[d]
7          property   Metadata         read        a{sv}→[a{sv}]
8          property   PlaybackStatus   read        string→[s]
9          property   Position         read        int32→[i]
10         property   Rate             readwrite   double→[d]
11         property   Shuffle          readwrite   double→[d]
12         property   Volume           readwrite   double→[d]
```

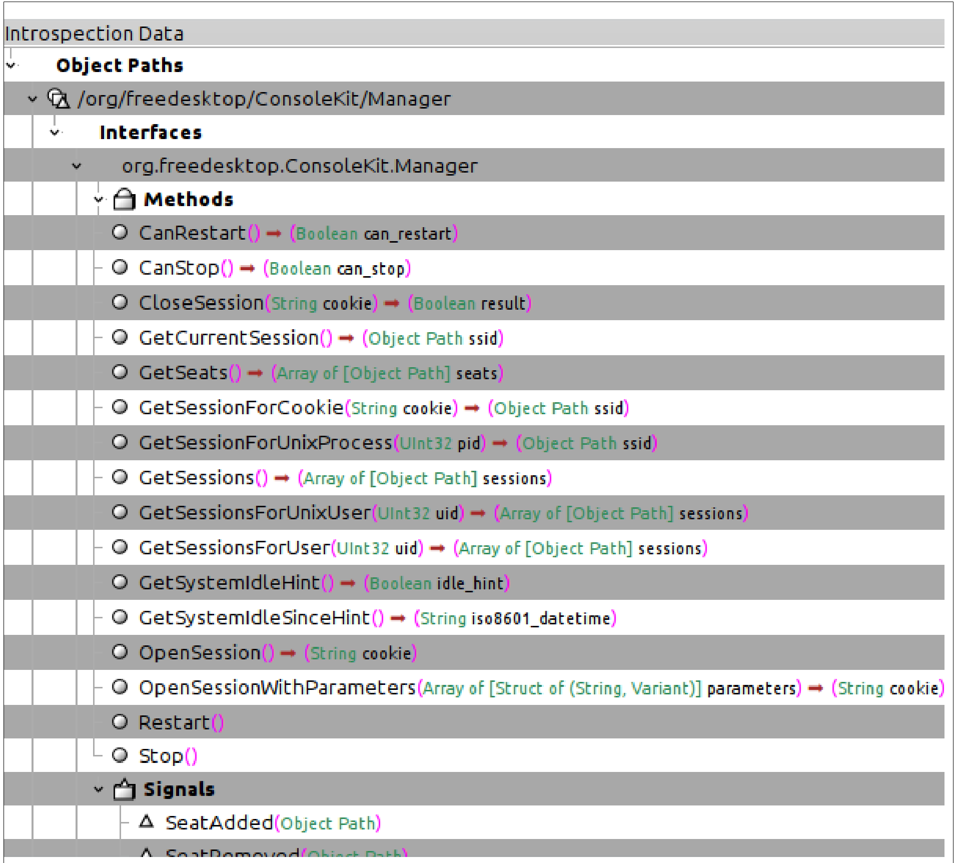
Figure 1: dbusdoc.rex output of mpris interface (excerpt)

representation is preferred, two tools are recommended.
The first one is called D-Feet[15] and is probably already available in the standard repository of

14 ooRexxUnit`s homepage: http://sourceforge.net/projects/oorexx/files/oorexxunit/, accessed on 2 September 2014.

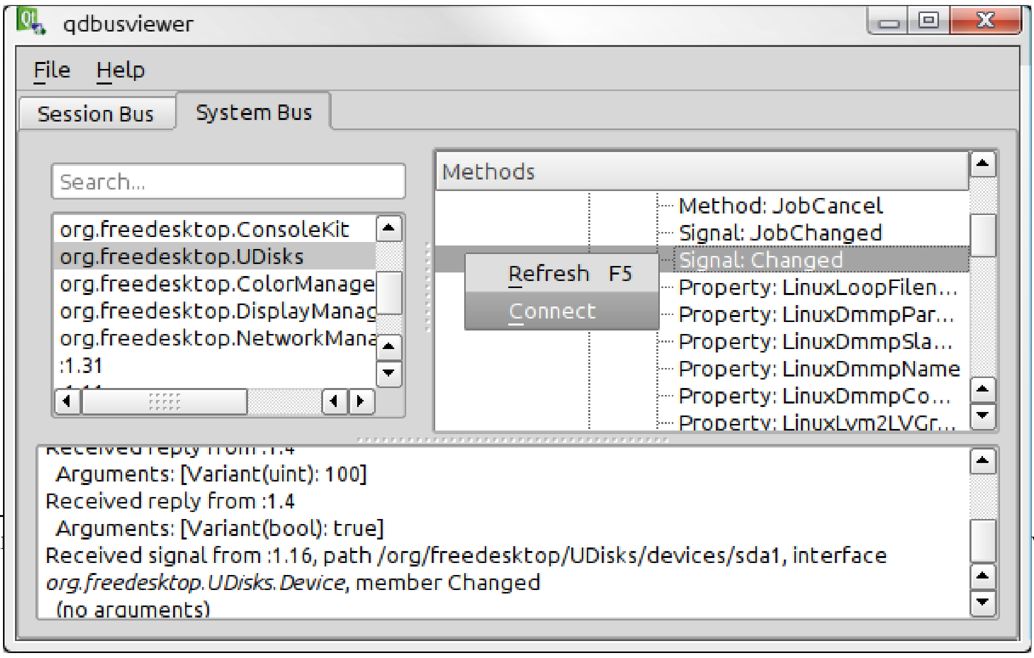
15 D-Feet's Homepage: https://live.gnome.org/DFeet/ (alternatively search for d-feet within your package manager.), accessed on 2 September 2014.

your Linux distribution, the second one is called qdbusviewer and is shipped with the qt4-dev-tools package[16]. Both have their strengths. D-Feet is very clearly represented, indicates expected object-types and return values for a service call and allows to invoke services and pass arguments. (see Figure 2 on next page) D-Feet also allows to establish a connection to other buses than session and system-bus.
qdbusviewer offers an additional feature. It is possible to connect to signals and watch for their appearance on the on the built-in output panel. (see Figure 3) This is handy if you are interested in signals from a specific application, but do not know how exactly the signal looks like. This program allows to connect to any signal and look if and when it appears.



```
Introspection Data
Object Paths
  /org/freedesktop/ConsoleKit/Manager
    Interfaces
      org.freedesktop.ConsoleKit.Manager
        Methods
          CanRestart() ➡ (Boolean can_restart)
          CanStop() ➡ (Boolean can_stop)
          CloseSession(String cookie) ➡ (Boolean result)
          GetCurrentSession() ➡ (Object Path ssid)
          GetSeats() ➡ (Array of [Object Path] seats)
          GetSessionForCookie(String cookie) ➡ (Object Path ssid)
          GetSessionForUnixProcess(UInt32 pid) ➡ (Object Path ssid)
          GetSessions() ➡ (Array of [Object Path] sessions)
          GetSessionsForUnixUser(UInt32 uid) ➡ (Array of [Object Path] sessions)
          GetSessionsForUser(UInt32 uid) ➡ (Array of [Object Path] sessions)
          GetSystemIdleHint() ➡ (Boolean idle_hint)
          GetSystemIdleSinceHint() ➡ (String iso8601_datetime)
          OpenSession() ➡ (String cookie)
          OpenSessionWithParameters(Array of [Struct of (String, Variant)] parameters) ➡ (String cookie)
          Restart()
          Stop()
        Signals
          SeatAdded(Object Path)
```

Figure 2: D-Feet's view on DBus services



```
org.freedesktop.ConsoleKit
org.freedesktop.UDisks
org.freedesktop.ColorManage
org.freedesktop.DisplayManag
org.freedesktop.NetworkMana
:1.31
```

```
Method: JobCancel
Signal: JobChanged
Signal: Changed
Property: LinuxLoopFilen...
Property: LinuxDmmpPar...
Property: LinuxDmmpSla...
Property: LinuxDmmpName
Property: LinuxDmmpCo...
Property: LinuxLvm2LVGr...
```

```
 Arguments: [Variant(uint): 100]
Received reply from :1.4
 Arguments: [Variant(bool): true]
Received signal from :1.16, path /org/freedesktop/UDisks/devices/sda1, interface
org.freedesktop.UDisks.Device, member Changed
 (no arguments)
```

Figure 3: qdbusviewer's signal handling capabilities

16 Sea viewer.

These two debuggers as well as dbusdoc.rex are used for the first assessment of a DBus service. Another interesting tool in this context is called dbus-send.[17] It is a command-line tool that allows to interact with DBus services and is probably useful if a simple method call has to be effected, as no setup is involved. In the case of other debuggers are failing to connect to DBusooRexx services, this tool will be used to assess the availability of services as well in order to exclude the debuggers as possible source of failures. A nice tool that can be used to get a visual representation of DBus traffic is called Bustle.[18] The last section of this document presents an excerpt of DBus traffic, captured during the execution of the final testgroup. Bustle allows to filter specific bus-names, enabling a very well structured view on all traffic from and to the DBus service.
The two debuggers with the graphical user interface and Bustle are regrettably not available for the Windows and Mac operating systems, but dbus-monitor and dbus-send can be used for that purpose as substitutes.

Although it is possible to code ooRexx in any given text editor, it might be handy to use an integrated development environment. Eclipse is well known for its extensibility and plug-in handling. RexxDev[19] is an Eclipse development plug-in, that can be used for syntax highlighting, code completion and debugging. RexxDev tries to be compatible with popular Rexx language interpreters, therefore supporting ooRexx code as well. Another plugin for this purpose can be found at[20]. If colorized syntax code need to be exported, using eclipse offers the most easy to use support. Whilst other text editors with syntax highlighting export the source code as HTML, eclipse needs no special export instruction. It is possible to use colorized source code with common copy and paste instructions.[21]
I nonetheless prefer to use kate, the text-editor shipped with KDE. Although ooRexx is not supported, regular Rexx syntax highlighting is sufficient enough to visually group the code and the built-in console window allows to start scripts within the text-editor in an easy way.

17 dbus-send documentation: dbus.freedesktop.org/doc/dbus-send.1.html. accessed on 2 September 2014.
18 Bustle download location: http://www.willthompson.co.uk/bustle/, accessed on 2 September 2014.
19 Information about the RexxDev eclipse plugin is available at: http://rexxdev.sourceforge.net/. Although this project was updated the last time in 2008, the plugin was tested to work. accessed on 2 September 2014.
20 Rexx plugin for Eclipse: http://sourceforge.net/projects/rexxeclipse/ accessed on 2 September 2014.
21 eclipse exports text in Rich-Text-Format as default.

2. Implement Automated Testing with ooTest

It seems reasonable on the first sight to write a short test to check if the newly programmed functionality works as expected and if everything works well, implement the next functionality and write a new test. But errors can occur nonetheless, what if a newly developed service somehow interferes with another code which was not part of the test.
Therefore it is very valuable to have an exhaustive set of different testcases available, that can be carried out all together without additional effort every time a change on the code was undertaken or a new developed service is to be tested thoroughly.
Although developing testcases is of utterly importance, creating comprehensive testcases is a very difficult task as many different aspects have to be taken into account, comprising the interplay of already proved-to-work modules as well. Therefore it is advisable to start from the most basic assertions and fill up assertion per assertion within the testcase, ideally at the time of writing the code that needs to be tested.
For a comprehensive test it is necessary to automate the testing procedure, thus enabling to test thousands of assertions within seconds. This is the primary purpose of using ooTest.
The ooTest framework sits on top of the ooRexxUnit and was slightly adapted by Mark Miesfeld. ooTest extends the functionality of ooRexxUnit in order to cope some special need in testing ooRexx. Therefore most ooTest classes are enhanced subclasses of the original framework. (Miesfeld, 2009b) The ooRexx version follows the same concepts as the archetype, the JUnit framework. There are eight different methods predefined that are used to test results of function calls or states. (see Figure 4)
The ooTest package also contains all tests written so far that are used by ooRexx developers, thus ooTest do not only serve as testing framework, but also as collection of many various tests. (Miesfeld, 2009a) Therefore using ooTest optimally results in a give and take. If testcases are comprehensive and useful for the community, they might possibly be incorporated into the collection of test cases within the framework.

```
assertEquals(expected, actual, [msg])
assertNotEquals(expected, actual,[msg])
assertNull(actual,[msg])
assertNotNull(actual, [msg])
assertSame(expected, actual,[msg])
assertNotSame(expected, actual,[msg])
assertTrue(actual,[msg])
assertFalse(actual,[msg])
```

Figure 4: Assertion methods of ooTest

The logic behind creating tests with ooTest is straight forward. A programmer knows prior to a function call what the result must look like. This expected value then gets compared with the actual result of an invocation, to test whether they are consistent. The optional `msg` argument can be used to provide further information about the assertion. Using this argument is especially necessary if tests are intended to be useful for other persons as well, or if there is a more extensive explanation necessary why a certain result is expected. A simple example for a necessity to give further explanation would be an addition where for some strange reasons another result is expected as mathematics would impose. This optional message will only be displayed if the assertion fails, therefore giving an explanation why a certain value was expected is helpful to identify why the actual value differs. (Miesfeld, 2009b)
Assertions are grouped into test methods, which themselves are grouped into a testgroup. Therefore a testgroup contains multiple test methods, which could carry out multiple assertions. Although using this framework seems easy and straight forward, there are some

things that have to be considered anyway. The chapter error treatment covers one of these specialties.
The functionality of most of the assertion methods is easily understandable. Some of them are even interchangeable. If for example the test is about a value to be true, all three **`assertEquals`**, **`assertSame`** and **`assertTrue`** can be used. Of course **`assertNotEquals`** and **`assertNotSame`** would also produce correct assertions if the value to be expected is changed to false, designing testcases that way would be confusing though. Even if the programmer is sure about only having true and false as available values, these assertions would fail if .nil or an empty string is asserted.
The only pair of assertions that might need further explanation is **`assertEquals`** and **`assertSame`**. Whilst **`assertSame`** tests whether two values are exactly the same, **`assertEquals`** asserts whether the values are loosely equal. For example 1 and 1 are strictly equal, whilst `“dog “` and `“dog”` are, although looking the same, not exactly the same object. Therefore an error is thrown if asserted with **`assertSame`** because of the additional space character. For these kind of assertion, **`assertEquals`** is used, which asserts the two strings as loosely equal. If collections are compared, **`assertEquals`** needs to be used as well, as array “a” and array “b” are probably not exactly the same object, but might nonetheless contain same values and therefore, for a general understanding, are equal arrays (Miesfeld, 2009b).
Apart from the simple logic behind the usage of assertion methods, creating a comprehensive set of tests is a difficult task.

2.1 Creating a Testgroup with ooTest

The author of ooTest describes that the most easy approach for the first usage of ooTest is to adapt an existing template file. This scripts defines a class that subclasses ooTestCase and contains some example test methods. The simplest approach is to change assertions within the test methods and rename everything according own needs.
The created file has to be started via an additional rexx file (`rexx testOORexx.rex -R [directory] -f [filename]`.) The required file `testOORexx.rex` is to be found in the installation directory of ooTest and drives the automated execution of testcases by starting the test suite.(Miesfeld, 2009b) This script is responsible for making all involved programs available for the execution of the tests.[22] The parameters allow to specify a directory where all testgroups are executed (flag `-R`) or to run a single file, specified through the `-f` flag.
Although the file `testOORexx.rex` is intended to provide all requirements for the test execution. The author personally do not find it handy to always search for that file. Running the testgroup through this script complicates the usage much more than eases it. The author therefore decided to make the required files permanently available by manually copying `OOREXXUNIT.CLS` and `ooTest.frm` to the directory `/usr/bin/`.[23] Another possibility is to include its location in the path variable, or make an entry in the file `/etc/environment` to make it permanently available. As all required programs are now retrievable on this path, the testgroup can be launched with `rexx DBUS.testGroup` directly from any path it was edited, instead of using the command `rexx testOORexx.rex -R dbus -f DBUS` and use the full path of `testOORexx`.

As most of the tests need a counterpart that is specially designed for that task, a client-server architecture was implemented. The testmethods and all assertions are incorporated into the client part. The server is used to respond to service calls and additionally emits signals, the client can connect to. Therefore the client is dependent on a running testserver. It is not very

22 The dependency for the test is the program **`ooRexxUnit.cls`**.
23 More details to be found in the troubleshooting section at the end of the document.

handy to manually start the testserver each time before the testgroup is executed and close it after all tests have been completed, this is not the intention of automated tests. A much better solution is to have the testgroup manage its dependencies (in this case the availability of the testserver) by itself. ooTest provides two methods that are directly dedicated to these requirements. A `setUp` and a `tearDown` method. These methods are incorporated into the class ooTestSuite.

The `setUp` method is always called first and is therefore ideally suited to organize all requirements for the testclient. In this usecase the intention is to start the required counterpart, the testserver. There are different solutions to this problem. ooRexx can easily interact with the operating system. Issuing unknown instructions for the ooRexx interpreter let it pass the command to its environment. The rexx instruction `say address()` returns the environment to which commands are passed. (Ashley et al., 2009, p 376) As expected the answer of this call is `bash` on a Linux system. Any command not processed otherwise from ooRexx is sent there. Adding the line `"rexx ooTestDBusServer.rexx &"` to the `setUp` method starts the testserver. The control operator `&` instructs the shell to execute the command in a sub-shell and not wait until the process is finished.[24] If the command is issued without this control operator, the script would stick in the setup method and wait until the server quits, which wont ever happen as the server is programmed to keep alive until its method quit is called.

There is also another very interesting possibility to start services. It is possible to start a program over DBus by defining a service file. (This method will be demonstrated in the sub-chapter “Start Service by Name”). Although very elegant, this method was intentionally omitted for the testgroup as it requires administrative privileges to setup the service file, what might impose an (unnecessary) additional burden.

The counterpart of the `setUp` method is the `tearDown` method, called at the end of the script and designed for cleaning up everything that was created during the execution of the test methods, or within the `setUp` method. In this usecase the `tearDown` method is used to terminate the testserver and close all connections established to DBus.

As the testserver was started over the command line with the described control operator, the ooRexx scripts process number has to be queried if the process is intended to be terminated with the `kill` command[25]. This approach unnecessarily involves command line operations and is omitted as actually there is a direct connection available to the testserver over DBus. It is much easier to politely ask the testserver to shut down as it knows best what connections to DBus have been established and need to get closed and whether there are other cleanup instructions outstanding. This behavior was realized by a local variable called `.dbus.shutdown` The serverscript starts all services and keeps them alive until this variable is changed to `.true`.

```
do while \.dbus.shutdown
call syssleep 4
end

halt:
   conn~close        --  stop message loop
   exit -1
```

As every testservice implements a method called Shutdown which changes the value of `.dbus.shutdown`:

```
::method Shutdown
  dbus.shutdown = .true
```

24 More information can be acquired with `man bash`.

25 The shell command `man kill` gives more details.

If this method is called, the server script closes all other service objects and closes its connection to DBus. As the **tearDown** method of the testgroup is the last called method within a Testsuite, this Shutdown method is called first to shut down all services, followed by closing instructions for the testgroup's own DBus connection.
In the first attempt only the **setUp** and **tearDown** methods were implemented and a testrun showed that none of them got invoked. Looking deeper into the code of ooTest revealed that the script has to contain at least one **::method** test defined in order to execute.
Adding one test method to the adapted ooTest class, enabled the script to execute as expected. The testserver is started in the **setUp** method, the test method is invoked, its assertions carried out and finally the **tearDown** method shuts down the testserver and closes all DBus connections.
But upon a second testmethod is added, the **setUp** and **tearDown** methods are called every time before and after every testmethod execution. As the expected behavior is to start the server, execute all testmethods and close it in the end, the script was discarded and another class of ooTest was used.
The class **ooTestSuite** is perfectly fits the described usecase. A subclass of ooTestSuite was created containing the **setUp** and **tearDown** methods and the testmethods with their assertions are implemented as a subclass of **ooTestCase**.
This setup now enables to execute all tests by only issuing one command: **rexx DBUS.testGroup**.[26] This is much handier than navigating trough the file system and taking care that everything was started and closed correctly. The newly created subclass of ooTestCase is now ready for being enhanced by adding testmethods one by one.

2.2 Designing Test Methods

The eradication of the test methods does not follow their placement within the testgroup file. Tests are not carried out one after another as implemented, but executed alphabetically[27]. It is therefore important how tests are named if a sequence is desired. Given the syntactical restraints that every test method has to be defined with **::method** test[name**]**, the name firstly has to be meaningful and of course unique within the script and secondly, the alphabetical placement of the first character after the word test in relevance to other test methods has to be considered. For easier readability it was decided to use an underscore after the method test, (**::method** test_integer32) but any allowed character does the job as well. One may argue that defining test methods and keep the alphabet in mind is not that difficult, but if a meaningful test name has to be defined, such character restrains can become unnecessarily complicated. Especially if the number of testmethods within a testgroup is extended over time. One possibility is to use a sorting character after the word test (**e.g.** ::method test_a_integer32 and **::method** test_b_array) but this is neither beautiful nor very useful, as the last used sorting character has to be retrieved every time a new test method is implemented. Another possibility would be using numbers, but the last number has to be retrieved as well if a new test method is implemented. Therefore the best solution is to not to demand any sequence within the script.

In order to structure the testgroup logically, the tests will start with basic assertions in order to assess whether all DBus object types are processed correctly.
As distributing, receiving and processing messages is one of the main tasks of DBus, it comes logically to start with assessing them first. Upon the basics were tested thoroughly, meaning

26 The file DBUStestServer.rexx and the introspection file Testservice.xml have to be placed in the same path as the testgroup.

27 e.g the method test_arrays is executed prior to test_integer, although test_integer was defined a few hundreds lines before test_arrays within the testgroup file.

that every combination was tested including tests that could or should produce errors, the functionality of DBusooRexx has to be assessed. This incorporates all features that were introduced by the author of the DBus ooRexx binding, which need not necessarily be common for other DBus language-bindings as well. As ooRexx intends to be an easy understandable programming language, the author of the binding follows this concept by implementing some features that further ease the use of DBus like automatic marshalling, syntax checks for introspection data and others that are going to be introduced and tested within the following chapters.

The following functionalities introduced with DBusooRexx are going to be incorporated into the testgroup:

> *“map D-Bus messages directly to ooRexx messages, take advantage of the ooRexx built-in unknown mechanism; this allows any ooRexx object to receive and process any D-Bus message” (Flatscher, 2011a, p 9)*

Mapping DBus messages to ooRexx messages means that DBus messages can be processed as easy as if they were ooRexx messages, that also implies using the same syntax for instance. The testgroup contains many different DBus instructions that do not differ much from regular ooRexx code. The testgroup also contains a service that provides a method called unknown, which is used to return all DBus objects that were passed over to it. Although not designed for that usage, using the unknown method allows to successfully omit defining introspection data a priori.

> *“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.” (Flatscher, 2011a, p 9)*

The testgroup contains some test methods that assess the availability of this slotDir, as well as its nonexistence upon a flag is used to refrain DBusooRexx from creating it.[28]
The testgroup will also inspect information transported within this directory and assert both, the sender and the receiver's slotDir.

> *“...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,...” (Flatscher, 2011a, p 9)*

This is another very interesting function and also not common to all bindings. This approach very much eases the interaction with DBus as for an ooRexx programmer, all DBus invocations are coming as easy as interacting with common ooRexx objects. There is no need to deeper understand what exactly is done behind the curtain. Sometimes even a simple service invocation needs a complicated object type to be created and forwarded. This will be tested by issuing messages to some service objects to be known to be available on every system, the tests envisage to subsequently omit values in order to test automatic completion of expected values.

> *“...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*

28 Following instruction disables the SlotDir: `.dbus~session~makeReplySlotDir=.false`

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",..."
(Flatscher, 2011b, p 9)

The application of the `DBusProxyObject` allows to interact with DBus services as if they were regular ooRexx objects. Within the testgroup every call is effected over a direct message call as well as over this proxy object. The latter considerably eases the usage of DBus as the familiar syntax: object, tilde, instruction (`dbusobject~doSomething()`) can be used as if DBus objects were regular ooRexx objects.

"...create an ooRexx class that allows local filtering of signals, named "DBusListener",..." (Flatscher, 2011b, p 9)

Listening for signals and react appropriately upon their arrival is also a very important feature of DBus. Both emitting as well as receiving signals is tested within the testgroup. As the testgroup consists of a client-server architecture, signal emission is tested upon startup of the script, as the server informs the client that it is available by emitting the signal "Ready". The client, on the other side, listens for this signal whilst detaining execution of its assertions until the signal successfully arrives. Listening to signals and reacting upon their arrival will also be used within the example scripts 31 and 34 on page 63f.

"...create an ooRexx class that makes it easy to implement services in ooRexx, named "DBusServiceObject"..."
"...create an ooRexx class that makes it easy to establish a private D-Bus server, named "DBusServer",..." (Flatscher, 2011b, p 9f)

Of course all of these classes designed to further ease the use of DBus are deployed within the testgroup or within the example scripts. DBusServiceObject is used as counterpart of the testgroup, containing all methods, the testgroup calls. There are already two predefined addresses available per default, a DBus service can connect to, named session bus and system bus.
The DBusServer additionally allows to use different ways of transporting messages, these approaches are assessed as well. It is possible to choose amongst Unix sockets, launchd, nonce-TCP/IP sockets and TCP/IP sockets. The latter ones even enable programs on different machines to interact over DBus and a network.
The class DBusServer is very handy as it enables to create a client-server architecture with ease. Although a client-server architecture can easily be realized with a DBus Service object as well, DBusServer allows to specify the accessibility to its services more precisely and additionally it can be used without a message bus daemon that serves as broker of DBus messages. DBusServer does not interact with the class `org.freedesktop.DBus`, therefore no unique bus name is assigned.

"...create support for creating and analyzing introspection data on-the-fly."
(Flatscher, 2011b, p 10)

The testgroup contains four different methods to provide introspection data, creating it on the fly is one of them. This is a very interesting feature to circumvent struggling around with xml introspection data, which will be further described in the section "3.2.1 Providing

Introspection Data - DBusServiceObject“.

2.3 Error Treatment

Sometimes it is known that a certain message call will produce errors, but although it seems reasonable not to invoke method calls that will produce errors, it is nonetheless necessary to assert their occurrence. If errors are not thrown, the occurrence of unexpected behavior in the further execution of the script is often only a question of time. For example if a value of an integer is too high for its representation[29] and no error is thrown, the user does not know whether the value was simply ignored or even worser, manipulated (e.g. cropped) by the script.
For the implementation of error assessments some things have to be considered. First of all, if the error is fatal it might possibly tear down the testscript and other involved programs, such as Java as well. Such errors should be documented but not incorporated into testcases as the repetition of their execution would not make much sense. Minor errors can be caught with their associated error code (Miesfeld, 2009b). The following example demonstrates this process, a service was instructed to return the value it receives. In order to invoke an error, the service was called, without passing over any argument.

```
Test:   TEST_DBUSOBJECTS_STRINGS_DIRECT
Class:  DBUS.testGroup
File:   /home/zerkop/MasterThesis/snipplets/DBUS.testGroup
Event:  [SYNTAX 93.903] raised unexpectedly.
  Missing argument in method; argument 1 is required
  Program: /usr/bin/OOREXXUNIT.CLS
  Line:    282
```

Error statement from ooTest

In this case, the error message is very meaningful, it informs about why and where the error occurred (program line) and most important for the task of defining testcases, the error number `[SYNTAX 93.903]`.
Given that number, it is possible to expect this error by using `self~expectSyntax(93.903)` prior to the service call that produces this error.
Sometimes it is nonetheless not that easy to find the error and get a meaningful error description. In the demonstration above, only an single error occurred. If a severe error occurs, it might tear down the testscript and/or other involved ooRexx objects (e.g. the answering service implemented for the testcase). In that case it is not easy to assess what the error was. In most cases it is the very first error that is mentioned in the error message, but sometimes an error is followed by hundreds of subsequent failures if other test methods rely on an, in the meantime, corrupted service.
Sometimes it is not possible to reach the first error occurrence within the shell as the default limit of the scrollback buffer hinders this. If error messages should be studied in detail, it is recommended to redirect them in a text file (`rexx DBUS.testgroup > error.txt`). Another good approach is to use built-in commands like `less`.
It is also very important to consider how many assertions are to be placed within a single testmethod and in which order, because upon the first assertions fails, the following assertions are not invoked at all, therefore other possible errors within the same test method are not revealed.

29 Integer object types are supporting a given number of digits. (see Table 2, p 29) A number with more digits cannot be stored if not enough memory space was allocated.

This is also true if an assertion is intentionally coded to invoke an error and therefore caught in advance with `self~expectSyntax()`. Although no error is thrown, the script continues with the assertions of the next test method, skipping all remaining assertions within its executing method. Therefore assertions that are known to produce errors should always be implemented in their own method or placed at the very end of another test method.
If DBusooRexx was configured to use another setting than its default within a testmethod and the raise propagation of an error leaves the test method without reverting DBusooRexx to its previous state, upcoming assertions within other methods might fail.[30]

The absence of the remaining assertions after an error could lead to the frustrating situation that even only one error was reported and resolved, on the next execution the next error occurs because the erroneous assertion gets tested for its first time. Therefore it is useful to keep in mind how many assertions are actually implemented within the testcases. The after-error-skipping of assertions can be demonstrated with the following output of two test runs of nearly the same testscript. The first example shows the output of successfully executed tests. Note that there are 236 assertions incorporated into 69 tests.[31]

```
Tests ran:              69
Assertions:             236
Failures:               0
Errors:                 0
Skipped files:          0
```

regular testgroup

```
Tests ran:              69
Assertions:             221
Failures:               3
Errors:                 0
Skipped files:          0
```

erroneous testgroup

In the second example, three failures were detected and listed, but only 221 assertions effected. This reveals that some assertions were skipped (actually there were three (faulty) additional assertions added in the erroneous testgroup) due to the `Raise` propagation[32] what returns or exits from the current executed method.

30 For example method_a defines a bytearray to be marshalled as string and method b assert whether a bytearray is returned exactly as it was sent. If method_a is left without reverting the marhalling to its default setting, the assertino for equality will fail in method_b.
31 This represents a very early stage of the testgroup.
32 Raise Propagate see http://www.oorexx.org/docs/rexxref/x4080.html, accessed on 12, january 2014

3. Testing the ooRexx DBus Binding with ooTest

The following chapter introduces DBusooRexx in its assessment by ooTest. The examples include handling DBus signals, interaction with DBus objects on the system and on the session bus and interaction with properties.
In the first section the marshalling capability of DBusooRexx is investigated. A set of predefined, different objects type instances is sent to DBus services and asserted whether they are returned correctly. These testcases are not only valuable as functionality is tested, but might also provide templates how to implement an interplay with DBus for own applications.

The following chapter describes the tests that have been carried out with the basic DBus infrastructure. This incorporates interaction with the DBus class and testing general availability of type-codes.

3.1 Connect to DBus and Test the DBus Class

DBus provides two system-wide predefined buses, the session and the system bus. Any application can connect to one of these and invoke or provide services. In order to establish the connection, three different approaches are available.
The first test comprises an assertion whether the three different notations are resulting in the same connection, followed by a negative test where system and session bus are asserted to differ. As the following tests will stick to one of these notations for establishing a connection, a prior assertion is probably useful. The three connection approaches are:

```
long   = .dbus~connect('system')
medium = .dbus~new('system')
short  = .dbus~system
```

The testgroup compares unique busnames of the three connections with following assertions:

```
self~assertSame(short~busName('unique'),medium~busName('unique'))
self~assertSame(medium~busName('unique'), long~busName('unique'))
```

As no differences were noticed, the shortest version (`.DBus~system`) is chosen to be used as default within the testgroup. There are many different abbreviations available through DBusooRexx to further ease and speed up its application. A programmer can choose among different syntax what best matches the individual programming style.
The established connection to DBus can now be queried for openness, authentication and connection. As the following examples demonstrate, there are also different notations available to choose from, some programmers might prefer the short query `'A'`, whereas others might prefer the more meaningful query `'isAuthenticated'`.

```
 conn = .dbus~session
 self~assertTrue(conn~query('Authenticated'))
 self~assertTrue(conn~query('A'))
 self~assertTrue(conn~query('isAuthenticated'))
 self~assertTrue(conn~query('isA'))
```

Other valid queries are: (the four notations presented above are available for following queries as well)

```
 self~assertTrue(conn~query('Open'))
 self~assertTrue(conn~query('Connected'))
```

All of these preliminary tests were carried out without any problem.

3.1.2 Type Codes

The next step is to test the availability of type-codes. The query method of a DBus connection was implemented to query all different type-codes on the session bus, followed by the same procedure on the system bus. There are 18 different type-codes available and additional four that are reserved but not (yet) actively used. (Pennington et al., 2014) Assertions of type-codes are implemented like following:

```
self~assertTrue(.dbus~session~query('typeCode', 'a'))  -- array
```

Two type-codes are available that must not be used in signatures, but only used within bindings to represent the general concept of the referenced object containers, namely the containers entries for Struct and Dict with their type-codes `'r'` and `'e'`.
Looking up the specification of DBus identifies the type-code `'e'` as `DICT_ENTRY`, denoting an entry in a dict or a map.

```
  self~assertTrue(conn~query('typeCode', 'r')) -- Struct
  self~assertTrue(conn~query('typeCode', 'e')) -- Dict entry
```

These containers are represented with `()` and `{}` in their signature, (see Table 1 of page 28) respectively by .Arrays and .Directories as equivalent ooRexx objects.
All type-codes are available as expected, therefore no error was found.

3.1.3 Bus Names

DBus manages ownership of services by allocation of names. Names can either be represented as a unique connection name, (for example `:1.129`) or with a so called well-known bus name. (like `org.freedesktop.DBus`) A bus name is reserved for the application that demands it for its lifetime unless otherwise configured. Restrictions for valid bus-names are following: a Bus name has to be composed of at least one element separated by a period, only regular ASCII characters are allowed, a bus-name has to contain at least one period character and must not exceed the maximum name length allowed. (Pennington et al., 2014)

The ooRexx DBus class provides methods for requesting and releasing bus-names and two methods to acquire information about bus names. (Flatscher, 2011a, Methods 2)

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

The optional flag statement for the request method carries integer values that are specifying the call. Three flags are allowed:

The flag `DBUS_NAME_FLAG_ALLOW_REPLACEMENT` enables an application to define, whether another application is allowed to replace its ownership. The second application calling for the same name will only replace the owner if the owner got its name with this flag attached and the calling application uses the flag, `DBUS_NAME_FLAG_REPLACE_EXISTING`. If no flag is set the second application requesting the same name will not replace the owner, but will be placed in a queue. Upon the primary owner giving up its ownership, the queued application receives the name. If this behavior is not intended, the third flag, `DBUS_NAME_FLAG_DO_NOT_QUEUE`, has to be added. (Pennington et al., 2014)

These variables are available through the DBus class and can either be used with their full name: `.dbus.dir~DBUS_NAME_FLAG_ALLOW_REPLACEMENT` or with their integer representations which are `1`, `2` and `4` for the three flags. An additional possibility is to use shorter representations of the names. For example the first flag can also be referenced with

`.dbus.dir~AllowReplacement` and the others flags analogously.[33]

As DBus has to inform the application that request a name if the name is valid and available, return codes are used. DBus provides four different return codes. `DBUS_REQUEST_NAME_REPLY_PRIMARY_OWNER` informs that the name request was successful, because there was no primary owner or the request flags were set accordingly as described earlier in both applications name request. `DBUS_REQUEST_NAME_REPLY_IN_QUEUE` means that the name already has an owner, no flag was set to prohibit queuing and `AllowReplacement` or `ReplaceExisting` was not defined. The third flag `DBUS_REQUEST_NAME_REPLY_EXISTS` simply informs that the name already has an owner and nothing is changed. Finally `DBUS_REQUEST_NAME_REPLY_ALREADY_OWNER` informs that there is no need to request this name, as the demanding application already is the owner.

The name release method provides three different return messages, `DBUS_RELEASE_NAME_REPLY_RELEASED` means that the name is now unused, `DBUS_RELEASE_NAME_REPLY_NON_EXISTENT` informs that this name does not exist and finally `DBUS_RELEASE_NAME_REPLY_NOT_OWNER` informs that the calling program is not allowed to release this name as it had no ownership. (Pennington et al., 2014) The short forms of these return codes for DBusooRexx are `.dbus.dir~Released`, `.dbus.dir~NonExistent` and `.dbus.dir~NotOwner` (or even shorter 1,2,3).

In order to assess all bus-name methods with all different flags, following testsetup was created: A method called `test_busname` was defined where a subsequent set of requests and releases of names is processed. All tests were assessed with both long and short form of flags and return codes.

Bus-names that were defined contradictory to its rules were identified correctly by DBusooRexx. For example following assertion that uses a bus-name with non ASCII characters.

```
busName = "ooRexx.sörviß"  -- illegal characters in busname
self~assertFalse(conn~busName("request", busName))
```

For the tests, the system unknown bus-name `oorexx.dbus.Test` was defined. The first step is to check whether the bus-name already has an owner, which must not be the case. The busname gets requested again if it has an owner. Afterwards the name gets released again and requested two times, the first time with success, the second time with a message informing that we are already owner. To complete the bus-name tests, a name ownership is released, which we are not owner. This setup enables to assess all return codes.

It was described earlier that tests should not be created in a way that one instruction manipulates an object which is later used in another assertion, but in this case an exception was made. For testing the flags it is necessary to have a second application request the name, if a name is requested twice within the same script to simulate queuing, the result of the return code is `ALREADY_OWNER`.

These preliminary tests could be effected by using the DBus class, for the following tests some additional setup is necessary.

3.2 TestServer – TestClient Architecture

As marshalling capabilities are going to be tested, it is useful to create two scripts that are able to communicate with each other and thus are able to exchange object types. Therefore it was decided to establish a client – server architecture. On the client side, all assertions are

33 These codes can also be looked up in the File `DBUS.CLS`. Do not change anything in this file.

effected. The server offers services that are defined DBus compliant within an introspection file. Script 1 shows an example of valid introspection data, defined in XML. This information is used by a DBus applications to get knowledge about any available service, signals, properties and about all signatures that are used to define object-types. Therefore, the interface **`Introspectable`** is of utterly importance and has to be provided. Script example 1 additionally informs about the availability of a service called `ReplyString`. The introspection data defines a string value as input parameter and return value, realized through the type definition `type="s"` for both directions.

```
'<!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="oorexx.dbus.ooTestServer">'  -
     '<method name="ReplyString">'  -
       '<arg name="input" direction="in" type="s"/>'  -
       '<arg name="output" direction="out" type="s"/>'  -
     '</method>'  -
   '</interface>'  -
 '</node>'
```

Script 1: Example introspection data

The method `ReplyString`, referenced by the introspection file must be made available, that is easily done by defining a method with the same name like following:

```
::method ReplyString
  use arg input
  return input
```

After example representations of all object types are being assessed, another iteration with all four object containers is implemented in order to test, if an object type within a container was tampered with during the unmarshalling and marshalling processes.

3.2.1 Providing Introspection Data - DBusServiceObject

Providing introspection data was already mentioned to be of utterly importance for making services available over DBus. If the introspection data was defined incorrectly, the service will not work as expected or not start at all. But there are also minor errors that can render some services unreachable, but other still work. This section introduces all four different methods how introspection data can be provided.

The first method is implemented in a service called **`TestService`**, which is a subclass of **`DBusServiceObject`**. The introspection data is defined within its init method as a string, which gets passed to the superclass. Introspection data is defined as XML statements. (see Script 2)

```
::class TestService subclass DBusServiceObject
::method init
  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></node>'
  self~init:super(idata)
```

Script 2: Defining introspection data - TestService

The second possibility is to define the XML data in an external file and pass it over if demanded. (**`TestService2`**) This service also subclasses `DBusServiceObject`. The script looks cleaner as no string has to be defined that passes multiple lines. (see Script 3) This approach also offers some other advantages which will be discussed later.

```
::class TestService2 subclass DBusServiceObject
::method init
  idata='ooTestDBusServer.xml'
  self~init:super(idata)
```

Script 3: Defining introspection data - TestService2

The third method, later referred to **`TestService3`**, provides its introspection data via its own **`::method`** `Introspect`. (see Script 4) This method does not use the class `DBusServiceObject`. That has some implications like the need for manual “translation” of object types with the `DBus.Box` statement introduced in the next chapter and the need for a default object path defined in order to become visible for DBus debuggers. Generally these three versions do not differ much in the way the introspection data is provided.

```
::class TestService3
::method Introspect
  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></node>'
```

Script 4: Defining introspection data - TestService3

Finally, the last method provides the possibility to implement and register DBus services on the fly. (**`TestService4`**) The ooRexx class is a subclass of **`DBusServiceObject`** and uses services from **`IntrospectHelper`**. The introspection data gets created in the **`::init`** `method` of the class and **`IntrospectHelper`** publishes them to DBus. (see Script 5) This certainly is one of the more elegant versions to provide introspection data, it is not necessary to define any XML code.

```
::class TestService4 subclass DBusServiceObject
::method init
  node=.IntrospectHelper~new
  if=node~addInterface('org.freedesktop.DBus.Introspectable')
  if~addMethod('Introspect',,'s')
  if=node~addInterface('oorexx.dbus.ooTestServer')
  --.some methods, signals and properties..
  introspectData=node~makeString
  self~init:super(introspectData)
```

Script 5: Defining introspection data - TestService4

Apart from its aesthetics, using this method provides a big advantage in comparison with the

other possibilities explained. The class **IntrospectHelper** is able to check the syntax of the introspection data and throws errors if syntax rules are violated. That means if anything within the defined introspection data is defined incorrectly, **RAISE PROPAGATE** is called.
The following example demonstrates this behavior: Two methods were added to TestService4, one of them is defined according the syntax rules for introspection data, the other not. (`addMethod('correctsignature',,'as')` and `addMethod('invalidsignature',,'sa')`.)

```
  node=.IntrospectHelper~new
  if=node~addInterface('org.freedesktop.DBus.Introspectable')
  if~addMethod('Introspect',,'s')
  if=node~addInterface('oorexx.dbus.ooTestServer')
  if~addMethod('correctsignature',,'as')
  if~addMethod('invalidsignature',,'sa')
  introspectData=node~makeString
  self~init:super(introspectData)
 addInterface('oorexx.dbus.ooTestServer')
```

If the rexx scripts that instantiates the DBus class provides error treatment with **signal on syntax name** `halt` and the matching halt label, any syntax mistake that arises in the introspection data will be propagated there.

```
signal on syntax name halt  -- make sure message loop gets stopped
signal on halt                                 -- intercept ctl-c

halt:
 errormessage = (Condition('ADDITIONAL'))       -- error information
 if errormessage[1]==.nil then do               -- emit exit signal
  testservice4~service.sendSignal(objectPath, interface, 'Exit', -
                       'Goodbye, thanks for starting me')
 end
 else say errormessage[1]

  conn~close              -- close, terminating message loop thread
  say 'connection closed ...'
   exit -1
```
Script 6: Refinement for Error Treatment - TestService4

The execution of TestService4 yields in following error statement:
`'out'-signature: signature [sa] Missing array element type`
If a wrong typecode was defined, the error statement would look like:
`'out'-signature: signature [w] contains unknown typecode 'w' at position 1`
It is not allowed to specify a signature in this order as after the array declaration (a), it is mandatory to specify what kind of object type the array stores.

IntrospectHelper will not let the script execute until every introspection data syntax rule is met. The other methods to provide introspection data do not effect any syntax check. In the given example, all services that were defined before a faulty line of code will be available, the faulty service and all services that were defined afterwards in the script are omitted.
This also demonstrates that defining introspection data can be very tricky. If typing errors within the introspection data are made, the script might run nonetheless, but some services are not callable. (Although qdbusviewer lists every service, also those who could not be called due to an error in their definition).

It depends on personal programming preferences which of these methods is preferred for defining introspection data. People who are used to code XML, probably prefer this method.
As XML is a widely used markup language, most text editors ship excellent support for

creating, formatting and reviewing the code. With the help of syntax highlighting errors become much easier visible. Additionally, it is possible to collapse the code tree to better structure the data. If a bracket was omitted somewhere, the editor marks this mistake as well. For large introspection data it is probably more efficient to copy the string to a separate XML file for debugging than reviewing a 40 line long string within the script. The only drawback of this method is that the XML file has to be carried along with the ooRexx code. If changes in the services are to be made, both scripts have to be adapted as the introspection definition and the service definiton have to match. Handling very long strings that are distributed over multiple lines within a script also has certain drawbacks. If there is an error in any of the lines (an unmatched quote or an unsupported character), the error message only indicates the program line where the string starts, not the line where the mistake actually appears. Additionally it is very difficult to review the long string as providing a structure is only possible through indenting. The last method demonstrated is the authors personal favorite, `addMethod('ReplyBoolean','b','b')`.
The instruction is defined with one line, indicating the method-name, the object type of the input value(s) and the object type of the return value(s). Adding and removing instructions are much easier than in other methods demonstrated. Additionally identifying faulty introspection code is easier.
In the case of `TestService3`, which does not subclass DBusServiceObject, all object types are of type string, no matter if the signature demands it to be of another type. Any object has to be "translated" into another object type by using `DBus.Box`, for example: `DBus.Box('i', 15)` in order to convert the string "15" to an integer value. That involves unnecessary coding which would else be carried out through the DBusServiceObject.

Although `TestService` and `TestService2` do not differ much, all four testservices were implemented within the testgroup. The names of the services mentioned in this section will be used in the next section as well and refer to the method how introspection data is provided.

3.2.2 Establish a Private DBus Connection - DBusServer

A DBusServer does not differ much from a DBus Service except that it is used without a message bus daemon and controls its accessibility on its own. For this purpose the method `allowAnonymous` can be used. A server has an own address, this can either be defined with `localhost` or with a real IP address. The following examples are valid addresses for setting up a private DBus server.

```
address1="unix:path=/tmp/dbus-test"
address2="tcp:host=localhost,port=23000,family=ipv4;"
address3="tcp:host=localhost,port=23000,family=ipv4;unix:path=/tmp/dbus-test;"
address4="tcp:host=192.168.0.15,port=23000,family=ipv4;"    -- ip address
```
Script 7: Valid addresses for a private DBus Server

The latter enables to setup a server on a remote machine with only three lines of code. (See also script 26). The variable testservice refers to an instance of DBusServiceObject.

```
server =.DBUSServer~new(address3,testservice)
server~allowAnonymous=.true
server~startup
```

For the testgroup the address1 and address3 variation is used to setup two DBusooRexx private DBusServers. For version 4 of addresses, it is necessary to obtain the IP address and it might also be necessary to configure the operating system, respectively a firewall to allow the connection. As the intention of this testgroup is that it can be executed on a vanilla operating system with its default settings, this test was omitted, but script example 26 makes use of it and demonstrates how to enable a DBus connection between two different machines.

Connection to a private server can be established as easy as the setup was, with only a few lines of code:

```
address="tcp:host=localhost,port=23000,family=ipv4;unix:path=/tmp/dbus-test;"
privateconn=.dbus~new(address)
privateserver = privateconn~getObject(busname,objectpath)
```

The only difference for a user is that the connection does not use `.dbus~session` or `.dbus~system`, but provides an address. If the proxy object is created, the interaction does not differ from the DBus services described earlier.

For the testgroup the two servers are offering the same services as `TestService2` does, therefore the test is about effecting all assertions over the private connection as well.

3.2.3 Direct DBus Method Calls and the DBusProxyObject

The ooRexx DBus binding offers different methods to communicate with DBus services. If a certain service object has to be invoked, it is not necessary to define a proxy object for it, but call it directly. The following example demonstrates a direct call to a running instance of the vlc media-player.[34]

```
busname = 'org.mpris.MediaPlayer2.vlc'
interface = 'org.mpris.MediaPlayer2.Player'
objectpath = '/org/mpris/MediaPlayer2'
.dbus~session~message('call',busname, objectpath,interface,'PlayPause')
```
Script 8: Direct message call

In this example, using a direct call is easy as the service object does not require any arguments, nor returns any (only the method name is passed over). Therefore it is not necessary to indicate any signature for the object types at all.

The same call can be effected with a proxy object that, once defined, is more handy to use than a direct call.

```
busname = 'org.mpris.MediaPlayer2.vlc'
objectpath = '/org/mpris/MediaPlayer2'
vlc =.dbus~session~getObject(busname,objectpath)
vlc~PlayPause
vlc~Stop
```
Script 9: ProxyObject message call

It especially makes sense to use proxy objects if the DBus service is to be called more than once and different methods are invoked on it. If the service needs parameters and returns a result, the object types have to be defined. This is done via their signature. The following sample calls the method ReplyInt32, defines Int32 as input and as output parameter as indicated by the signature `'i'` and forwards the message, in this case the value 0.

```
conn~message('call','oorexx.dbus.ooTestServer','/oorexx/dbus/ooTestServer', -
'oorexx.dbus.ooTestServer','ReplyInt32','i','i',0)
```

Although DBus is strictly typed, DBusooRexx is capable to translate (nearly) everything on the fly, making its usage as easy as possible. This can perfectly be demonstrated with the

34 Chapter 4.1.3 The Media-Player-Interface Standard MPRIS will describe the interface used in the following examples in more detail.

previous example “translated” to use a proxy object instead of the direct call.

```
proxy = conn~getObject('oorexx.dbus.ooTestServer','/oorexx/dbus/ooTestServer')
proxy~ReplyInt32(0)
```

This is of course a very simple object type to handle, the testgroup will incorporate more complicated signatures as well, which will test the automated object type conversion of the DBusooRexx proxy object. But it is obvious that using a `DBusProxyObject` makes the code look much more familiar to ooRexx programmers.

In some cases it is nonetheless necessary to specifically define the object type manually. Therefore DBusooRexx offers a service to “force” objects into another objects with a signature: `DBus.Box('signature', value)`.

If a DBusooRexx Service does not subclass DBusServiceObject, all object types are of type string, no matter if the signature demands it to be of another type. The objects have to be marshalled with the DBUS.BOX instruction.

The other common usecase is the definition of a variant. If the instruction `DBus.Box('d', 1.9)` is used, the value 1.9 is actually converted into an array with two elements, the first defines the object type with the string `useThisSignature=d`, the second carries the value. The box functionality also allows to override the signature of the introspection data. For example even if the signature demands an object type to be of type string, it is possible to provide a byte array instead by using `DBus.Box(’ay’, 'examplestring')`.

Although in regular cases the DBusProxyObject targets the correct member automatically, if service objects provide different interfaces, but have members with the same name within these interfaces, the proxy object might target the wrong member (Flatscher, 2011a). In this case it is necessary to add the interface name to the method to be called. Script 10 demonstrates this method. (`vlc~org.mpris.MediaPlayer2.vlc.PlayPause`) It would be possible to define every call this way if it is preferred.

```
busname = 'org.mpris.MediaPlayer2.vlc'
objectpath = '/org/mpris/MediaPlayer2'
vlc =.dbus~session~getObject(busname,objectpath)
vlc~org.mpris.MediaPlayer2.vlc.PlayPause
vlc~org.mpris.MediaPlayer2.vlc.Stop
```

Script 10: ProxyObject message call full name

In order to assess both versions of method calls, the testgroup issues direct calls to the services and uses their proxy object within the test methods.

3.2.4 Assess the TestServices with DBus Debuggers

Before starting with ooTest, all testservices are inspected for proper functionality with the two DBus debuggers described in the introduction. The first test was restricted to look up if all services are available and to invoke each of them via the GUI of D-Feet and qdbusviewer.
These are also the first tests to list and call services from programs, that are not ooRexx scripts. As the interaction with debuggers and other DBus services that were already available work, the same is expected with services implemented in DBusooRexx.
In order to become visible for debuggers, **`TestService`** , **`TestService2`** and **`TestService4`** need to call the command `.IDBusPathMaker~publishAllServiceObjects(conn)`. (conn being the DBus connection where service objects are added to.) **`TestService3`** (no subclass of `DBusServiceObject`) needs to define a default path first:
`conn~serviceObject('add', 'default', .IDBusPathMaker~new(objectPath))`.

This leads the introspect finder to the available object paths. The first testrun revealed errors that never appeared if accessed with another ooRexx DBus service class. Although the bus name is listed, D-Feet was not able to connect to `TestService` correctly. Trying to invoke the introspection tears down the script.
Investigation of the error message[35] showed that the debugger calls the method `GetAll`[36] during the introspection. (which was not provided at that time) This is not an error of DBusooRexx, but a programming error. It is however strange that D-Feet struggled, whereas qdbusviewer had no problems with displaying available services.
Adding `::method unknown` to the script partly solved the introspection problem. It was not necessary to define any action within the unknown method, it just has to be available in the case of a (misleaded) method call, in this case, the `GetAll` call.
This setup allows both debuggers to list available services. But there are nonetheless some remarkable differences.[37] qdbusviewer lists everything which was defined according to the introspection data, whereas D-Feet only lists a subset of the services, but none of signals and properties and no cascaded containers. The listed subset of methods could be invoked successfully through D-Feet. It showed later that four faulty lines of code were stopping the discovery of the remaining services within the introspection data. It seems that D-Feet reads introspection data until it discovers something that does not conform with the specification. qbusviewer seems to follow another approach and lists everything it parses from the XML data.
qdbusviewer could not invoke all services of the test service objects successfully. Whereas D-Feet could process all integer objects, qdbusviewer struggled with “less prominent” integers, namely Integer16 and UInt16. Their invocation gets answered with an error message.[38]
These debuggers do additionally show different output in their object type representation. qdbusviewer could not invoke the service `replyByte`, whereas D-Feet demonstrated other behavior. If this service is called and a character is passed as argument, D-Feet returns the Byte in its decimal representation. (e.g. the byte `'A'` is returned as `65`, `'B'` as `66` and so forth.)[39] If a German Umlaut is passed over, (like `'Ö'` for example) the debugger answers with `Must be a single character`. The service `ReplyArrayofByte` worked for both debuggers. If the same character from the previous example (`'Ö'`) is passed to this service, D-Feet answers with `[195,150]`, whereas qdbusviewer answers with `{-42}`. The byte representation differs between these two debuggers.
`TestService2` was designed the same way as the already investigated `TestService`, but reads the introspection data from an attached xml File and therefore has a clearer arrangement as XML data is separated from ooRexx code.
Both of these approaches were realized by creating a subclass of the `DBusServiceObject`. The third approach `TestService3` does not use the class `DBusServiceObject`, but defines all methods by itself. That means that the introspection data is returned in a method called Introspect. Without making introspection available trough adding `conn~serviceObject('add', "default", .IDBusPathMaker~new(objectpath))`, qdbusviewer quits the introspection attempt with an error message[40], the same applies to D-Feet. It was not even possible to list any

35 DBusooRexx Error message: `REX0646E: Error 97.900: Object "a TESTSERVER" does not understand message "GETALL" (DBus message type "method_call": objectPath="/oorexx/dbus/ooTestServer", interface="org.freedesktop.DBus.Properties", member="GetAll"`.
36 The sub-chapter Properties explains `GetAll` more detailed.
37 The DBus-debuggers seem to pursue different approaches to establish an introspection tree.
38 Error: `Unable to find method ReplyInt16 on path /oorexx/dbus/ooTestServer in interface oorexx.dbus.ooTestServer`.
39 Byte codes can be looked up at http://www.ascii-code.com/, accessed on 2 September 2014.
40 Erroe message if path for introspection is not made available: `Call to object / at oorexx.dbus.ooTestServer: org.freedesktop.DBus.Error.ServiceUnknown (Rexx service object`

service for both debuggers. But although the introspection data could not be retrieved correctly, invoking services work.
Also note that **TestService3** needs all its objects types getting boxed. In a first attempt **TestService3** was not instructed to marshall its object types with the **DBUS.Box** instruction.
The command line tool **dbus-send** was used to invoke a simple service that returns the argument, that was passed over in the invocation.

```
dbus-send --print-reply –dest=oorexx.dbus.ooTestService
 /oorexx/dbus/ooTestService3 oorexx.dbus.ooTestService.ReplyInt32 int32:5
```

The arguments passed within this command are busname, objectpath and the interface name combined with the service name to call. The syntax of an argument is **<object type : value>** . In this example a service named **ReplyInt32** is called and the argument **int32:5** is passed over. The return message of this program reveals that the return value is a string:

```
method return sender=:1.339 -> dest=:1.369 reply_serial=2
string "5"
```

Script 11: dbus-send – Testservice3 without boxing

Although **TestService3** defines the return value to be of object type Integer32, `dbus-send` answers with a String as demonstrated in script 11. Of course this is no problem for Rexx as everything is a string, but it is not the expected object type and the signature should force the value to be of the expected object-type, namely Integer32.
Repeating the same test with the tool `dbus-send` and `TestService3` instructed to box object types accordingly, the expected integer32 arrives.

```
method return sender=:1.434 -> dest=:1.436 reply_serial=2
int32 5
```

Script 12: dbus-send – Testservice3 with boxing

To complete these preliminary tests with the debuggers, the last described method, defining the introspection data on the fly was tested (**TestService4**).
This service showed the same behavior as the other already successfully tested **TestService** and **TestService2**.

3.3 Final Test Services

After these scripts have been invoked one by one, the author decided to incorporate them all into one serverscipt that allows to test all **TestServices** simultaneously and that also allows to test different methods to declare properties, (see 3.7 DBus Properties) within the same testgroup.
For this setup another class was created that subclasses **DBusServiceObject** and provides all test methods that all **TestServices** have in common, like:

```
::class DBusServiceObjectProxy subclass DBusServiceObject
::method ReplyBoolean
  use arg input
  return input
```

This class is further subclassed by the different **TestServices**. As **TestServices3** is intended not to subclass **DBusServiceObject**, it does consequently does not subclass **DBusServiceObjectProxy**.

[/] does not exist (you intended to invoke member=[Introspect] in the interfaceName = [org.freedesktop.DBus.Introspectable]) failed.

The final serverscript is further enhanced by **`TestService5`**[41] that offers only two methods, unknown and shutdown. If a method call arrives at **`TestService5`** that is not mapped to a method within the introspection, `::method unknown` is invoked. By implementing this method and return the object type of the message call, **`TestService5`** is able to catch any call, unless the name of the method call exists.[42] As the signature is extracted out of the call, it is only possible to use it with a direct message call that defines the return object type like integer in the following example, else a string is returned.
`conn~message('call', busname, objectpath, interface, unknownname, 'i','i', 1)`. This example sends an integer value to an unknown name and provides the signature `'i'`. If using the `DBusProxyObject` it would not be possible to make a call to a **`TestService5`** service as the proxy object relies on the signature provided by the introspection data.

This approach to use the DBus infrastructure is obviously interesting as no introspection data has to be provided in advance and is useful for testing purpose as signatures can be combined randomly and tests are easily extensible without much effort.
`ooTestServer` and **`ooTestServer2`** were also added to the testserver script. Thus resulting in seven slightly different services sharing all methods. All different variations to provide introspection data as well as all variations to define properties are used. Additionally establishing three different methods for connections were inspected, the common **`dbus~session`**, a private server reachable over the localhost and a private server reachable over an unix path.

3.4 Assessing Different Object Types

The following tests demonstrate how DBus object types are created in ooRexx and sent to existing DBus services. As DBus is strictly typed, it is necessary for DBusooRexx to marshal the objects during their transport over DBus and demarshal them again for the receiving program, no matter what language it is written in. It would not be very handy to exactly define every object type within an ooRexx script, therefore the binding does this complicated part by itself. The programmer only needs to create regular ooRexx objects and let the binding translate and transmit these.

41 A testset provided by the author of DBusooRexx was adapted as it contains a possibility to test marshalling and unmarshalling in an easy way.
42 Using `::method unknown` might need all methods of the interface `org.freedesktop.properties` implemented if properties are provided. Else they are not reachable anymore via `Get` and `Set`.

There are 17 different object types which are used by DBus (see Table 1). (Flatscher, 2011a)

Data Type	Type Indicator (Signature)	ooRexx representation
array	a	.Array
boolean	b	Rexx String
byte	y	Rexx String
double	d	Rexx String
int16	n	Rexx String
int32	i	Rexx String
int64	x	Rexx String
objpath	o	Rexx String
signature	g	Rexx String
string	s	Rexx String
unit16	q	Rexx String
unit32	u	Rexx String
uint64	t	Rexx String
unix_fd	h	Rexx String
variant	v	Signature dependent
structure	(...)	.Array
map/dict	a{.s}	.Directory

Table 1: DBus type indicators

Four of them are container types. Probably the most common container type is the array, which denotes an ordered list of objects. A Variant is a special container that also carries the signature of its transported values. A Struct can contain any type according to its signature and finally the Map or Dict is a container type, where the index always consists of a string and the associated value can be of any type. Most abbreviations are easily assignable. The abbreviations `“x”` for int64 and `“t”` for uInt64 were chosen because `x` and `t` are the first characters in the word “sixty-four” that were still available (s was already used for strings and i for int32) (Pennington et al., 2014)

Some DBus bindings might possibly not incorporate all object types properly as some of the object types can easily be replaced with others. (eg small integers.) Of course DBusooRexx supports all object-types.

The serverscript that is used by the testgroup incorporates methods, whose only functionality is to return the object type which was sent to it. Each object type listed in Table 1 is incorporated into dedicated reply methods. The sending and receiving of these objects and the assertion of transported values are main tasks of the testgroup, which will act as clients in this testing setup.

The following section will describe each object type in more detail, especially if there is any remarkable specialty that delivers an unexpected result.

3.4.1 Integer Objects

As the various Integer objects support different value-ranges, each object has to be tested individually. For example an integer with 16 Bits supports values from -32768 to 32767, whereas an unsigned Integer with 16 Bits does not support negative values and therefore, (as covering the same value-range) supports values from zero to 65535.
This can easily be calculated as the range of a signed integer covers `-(2^{n-1})` to `(2^{n-1}-1)` whereas the unsigned representation covers the range from `0 to 2^n-1` (see Table 2).
These value ranges are being tested as well as some intentionally incorrect defined values,

```
::method test_dbusobjects_int32_invalid_93.900_digitserror
  numeric digits 5        -- setting digits intentionally too low
  self~expectSyntax(93.900)                 -- value is -2.1475E+9
  self~assertEquals(-2147483648, dbustest~ReplyInt32(-2147483648))
```
Script 13: Assert a digit error

including .nil, a string, an empty string, no value and values above and below the covered value range.

Object	min value	max value
int16	−32.768	32.767
uInt16	0	65.535
int32	−2.147.483.648	2.147.483.647
uInt32	0	4.294.967.295
int64	−9.223.372.036.854.775.808	9.223.372.036.854.775.807
uInt64	0	18.446.744.073.709.551.615

Table 2: Supported integer values

It comes naturally to a Rexx programmer that the String “1” is understood as 1, as everything is a String. (see Table 1) Therefore no error occurred with the following assertion: `self~assertEquals(1, dbustest~ReplyuInt16("1"), "string 1")` Tests with the String “one” and an empty String failed as expected and all values below or above the value coverage range also resulted in an expected failure. When no argument is sent on its round-trip, `Syntax 93.903` is raised, denoting `“Missing argument in method”`. This is common to all different object types and also marks expected behavior.

The test revealed that integers need to be handled with care. For example processing .nil values can be tricky. It has to be taken into consideration that DBusooRexx always return the value 0 instead of .nil if the signature forces the return value to be of type integer and the argument passed was the .nil value. (see chapter 3.5 DBus and NULL Values) The following assertion demonstrates this: `self~assertEquals(0, dbustest~ReplyuInt32(.nil), ".nil")`. This feature was implemented to ease the use of DBus and allows DBus to process .nil values without errors. (Flatscher, 2011a) Hence 0 can be regarded as safe default value for any integer.
Another possible source of misbehavior is the amount of digits that are processed correctly, especially in the case of high integer values. ooRexx has its default number of digits, controlling the precision to what arithmetic operations are evaluated, set to 9 (Ashley, Flatscher, Hessling, McGuire, et al., 2009). The current digit value can be retrieved by issuing `say digits()` and can be changed to another value with `numeric digits`, followed by the

desired number of digits (eg: `numeric digits 20`).

For handling big integers like int64, it is necessary to define the number of digits properly in advance. Processing the lowest allowed value for an int64 produces errors unless the number of digits is set to 17 at minimum, using any digit value below make assertions fail.
The tests therefore also include to intentionally restrain the amount of digits to test whether the error can be detected correctly. (see Script 12) Restraining the amount of digits to five, invokes a conversion as the value is rounded (for example the lowest allowed value for int32 is -2147483648, which is converted to -2.1475E+9). As expected, an error is thrown with the `code 93.900`, stating:

```
conversion (value=[-2.1475E+9]) to an INT32 value failed / error position (1-
based)     [1] in full signature: [i] (maybe typeCode not supported on platform?
```

The next step was to expect the stated syntax, but it was not thrown anymore therefore invoking the syntax with the `code 91.999` by itself.[43]
In order to test many integer values in an efficient way, a random number statement is implemented. Defining a random number for an integer lower than 32 bit works, but given the random statement's maximum difference of 999999999[44] between two values, it is not possible to cover the whole range within a single statement. A simple solution is provided. The random int64 value is defined as a sequence of three random values that must not exceed a certain limit.
As the lowest value for Int64 is -9223372036854775808 and has 19 figures, the first part was bounded to be between -922337203 and 0, the next part between 0 and 685477 and the last part between 0 and 5808. (see Script 13)

```
int64 = .list~new
do i= 0 to 100
  int64~append(random(-922337203,0)||random(0,685477 )||random(0,5808))
  int64~append(random(0,922337203)||random(0,685477 )||random(0,5807))
end
```
Script 14: Create int64 values randomly

A negative value value must never surpass the lowest boundary and not exceed 0. The second statement defines positives values for int64 in the same way. All of these values were processed as expected.

3.4.2 Double IEEE 754

A double is a 64-Bit floating point number. Analogously to the tests before, regular as well as irregular values are tested. The binding has no problems with passing over strings, they get “translated“ automatically. As seen in the following assertion:.
(`self~assertEquals(1, dbustest~ReplyDouble("1.0")`). It is also possible to make an exact assertion with `self~assertSame(1, dbustest~ReplyDouble("1.0"))` without invoking any error. But in this example the expected value has to be defined without comma, as whole number as seen in the following assertion and the corresponding error code:
`self~assertSame(1.0, dbustest~ReplyDouble(1.0), "value 1.0")`.

43 ooRexx error-codes often provide helpful information: See Error Numbers and Messages: http://www.oorexx.org/docs/rexxref/a34980.htm, accessed on 2 September 2014.
44 Error message if the value within the random statement is exceeded:: `[SYNTAX 40.32] raised unexpectedly. RANDOM difference between argument 1 ("-2147483648") and argument 2 ("147483647") must not exceed 999,999,999`.

```
Failed: assertSame
  Expected: [[1.0], identityHash="17466049620176"]
  Actual:   [[1], identityHash="17466050757742"]
```

ooRexx “converts” double values like 1.0, 2.0, 3.0 and so on to a shorter form, cropping away the (unnecessary) zero. This actually makes an assertion with `assertSame` fail. In this case `assertEqual` is very useful as it correctly assesses 1.0 and 1 to be equal..
Analogously to integer values, the digit limit has to be kept in mind when handling big doubles. A first version of DBusooRexx always cropped the digits to 9 characters, no matter if instructed otherwise with `numeric digits`. If this amount of digits was exceeded, the double value got cropped and only the remaining digits were asserted.
Following example demonstrates the cropping behavior which makes the assertion true, although it must be false.

```
self~assertSame(123456789.0,dbustest~ReplyDouble(123456789.12345678901234567890))
```

Fortunately the author of DBusooRexx was able to correct this error upon he got knowledge about it. In the current version, DBusooRexx 2, the error was already fixed, but nonetheless a programer has to keep in mind the default amount of digits (9). This can be demonstrated with the following example:
Following assertion with `assertSame` will not fail upon the default numeric digit is raised:

```
self~assertSame(-123456789.19,dbustest~ReplyDouble(-123456789.11),-
                               "different values after 9th digit")
```

Although there are different values after the 9th digit, this assertion does not fail. Using a say statement reveals that both values were processed as `-123456789` , therefore equality is de-facto given. If the digit limit is raised to 20, this assertion fails as expected, because (0.19 is not the same as 0.10999999940). See following ooTest output:

```
Failed: assertSame
  Expected: [[-123456789.19], identityHash="383569021"]
  Actual:   [[-123456789.10999999940], identityHash="383565577"]
  Message:  different values after 9th digit, limit 20
```

That makes doubles excellent demonstration objects for distinguishing between the two different assertion mechanisms `assertEquals` and `assertSame`. and its usage forces to always keep the digit limit in mind. But there is an additional specialty that is only valid for doubles. Sending a double value to the testserver and expecting exact the same value reveals that doubles are following “special rules”.
Double IEEE 754 is the standard for a binary floating-point arithmetic that stores its value in 8 Bytes (64Bits). One Bit is reserved for the sign, eleven Bits for the exponent and the remaining 52 Bits for the mantissa. (IEEE, 1985)

The agreement to use the IEEE 754 standard for floating-point arithmetic ended a chaos which was introduced by different arithmetics for precision and rounding procedures in the 1970s. Nowadays nearly every modern microprocessor complies to this standard. (Kahan, 1996). MikeCowlishaw who can be regarded as the father of ooRexx, defined the general decimal arithmetic. (Cowlishaw, 2003). Using 53 Bit for representing a double results in a precision of $53 \log_{10}(2)$ = 15.955 meaning that 16 digits can be processed with precision.
The (internal) value of a double is defined through $x=s.m.b^e$, whereas `s` stands for the sign (0 for positive numbers and 1 for negative values), `m` denotes the mantissa, `b` the basis which in this case is always 2 (binary) and e has to be calculated upon following rules: `e=E-B` (B stands for biasvalue and is calculated by $2^{r-1}-1$), m is calculated by $1+M/2^p$ or more easier `1,M`.

As the basis for double is binary, it is unavoidable that the value has to be rounded (e.g periodic values). This necessitate a programmer to understand rounding procedures as well as keeping digit limits in mind for assertions. Table 3 lists the boundaries of double IEEE754

Double	decimal	hexadecimal
min value	`2.2251 x` 10^{-308}	`0010 0000 0000 0000`$_{16}$
max value	`1.7976931348623157 × ` 10^{308}	`7fef ffff ffff ffff`$_{16}$
smallest number >1	`1.0000000000000002`	`3ff0 0000 0000 0001`$_{16}$

Table 3: Supported double values

It is not necessary to calculate every value with this formula if assertion with `assertSame` are envisaged. There are many converters online for calculating double values.[45] Some of them are able to separate the segments of the double visually. Figure 5 demonstrates the hexadecimal and binary representation of the value -1.9.

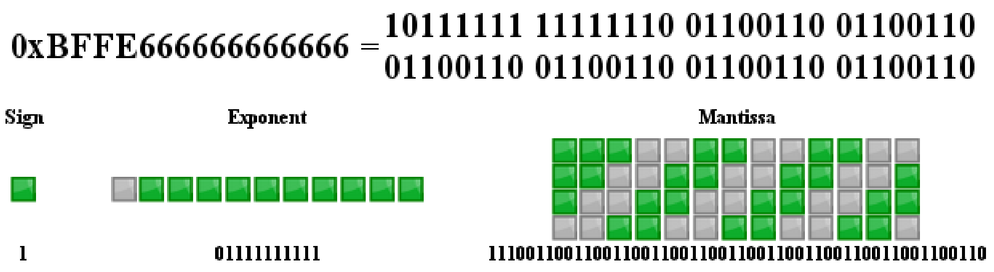

0xBFFE666666666666 = 10111111 11111110 01100110 01100110 01100110 01100110 01100110 01100110
Sign
Exponent
Mantissa
1
01111111111
1110011001100110011001100110011001100110011001100110

Figure 5: Example representation of double value -1.9

Source: www.binaryconvert.com

Table 4 lists some example values for doubles that were calculated using an online tool:

Double	decimal value	binary representation
Example 1	`-1.9`	`-1.8999999999999999112`
Example 2	`1.0`	`1.0`
Example 3	`1.1`	`1.1000000000000000888`
Example 4	`1.2`	`1.1999999999999999556`
Example 5	`1.3`	`1.3000000000000000444`
Example 6	`1.4`	`1.3999999999999999112`
Example 7	`1.5`	`1.5`

Table 4: Some example double values

Example 1 demonstrates, that the value -1.9 is represented by a value being very close to the -1.9, but nonetheless slightly different. (the default setting of dbusoorexx's underlying C library is 20 digits for double values, therefore -1.9 is the same as -1.8999999999999999112) If values are asserted with `assertSame`, expected values have to be adapted as indicated in Table 4 and rounded accordingly to 20 digits. `assertEquals` works anyway.

45 For example: http://www.binaryconvert.com/, http://babbage.cs.qc.cuny.edu/IEEE-754/, and many others, accessed on 2 September 2014.

If digits are changed with `numeric digits`, this instruction is only valid for the expected value within the assertion and does not influence the actual double value that get returned from the testservice. Therefore, if tests with numeric digits restrictions are effected, it is necessary to manipulate the returned double value as well in order to render them comparable is assertions with `assertSame` are effected..

For example if numeric digits are limited to 5, the assertion `self~assertSame(-1.8999999999999999112, dbustest~ReplyDouble(-1.9)` fails, although the expected value actually is the internal representation of the double -1.9. But in this test case, the maximum available number of digits is five (resulting in -1.9000 as return value).

The function `Format([value],[digits before decimal marker],[digits after decimal marker])`, rounds a given value according to the rules defined. This enables the assertion to become true, as both, the actual value and the expected value are rounded with the same format instruction. For example the negative value -1.9 is formatted with `Format(dbustest~ReplyDouble(-1.9),2,4)`. The first parameters reserves place for the sign and the whole number, the second parameter defines the digits after the decimal marker.
In order to assert many different values, two list were created that were filled with concatenated positive and negative random doubles values. See the following script:

```
positives = .list~new
negatives = .list~new
positives~append(random(0,9999999)||'.'||random(0,9999999))
negatives~append('-'||random(0,9999999)||'.'||random(0,9999999))
```
Script 15: Create random double values

It revealed that assessing big double values with the introduced debuggers is poorly possibly. The default settings of D-Feet restricts doubles to twelve digits and qdbusviewer, although allowing more digits, only processes two comma values.
Another specialty of doubles is, that some languages demand a point to separate the whole number, whereas others use a comma (e.g. English 1.0 vs. German 1,0). ooRexx always interprets a comma as separator between multiple objects and never as delimiter for doubles. Therefore following assertion results in the stated error as ooRexx expects additional arguments which were not defined in the signature:
`self~assertEquals(1,1, dbustest~ReplyDouble(1,1),"1,1 used in German for example")`.

```
[SYNTAX 93.900] raised unexpectedly.
    DBusooRexx/method/DbusBusCallMessage(), error 5: argSignature [d]
mandates at most [1] Rexx arguments, however Rexx supplies too many
arguments [2]
```

3.4.3 Strings

Strings must be encoded in UTF-8 (Unicode Transformation Format 8-bit) for DBus. This format can represent every Unicode character (U+0000 is not allowed.[46]). Since DBus version 0.21 also some “noncharacters” are allowed, namely “U+FDD0..U+FDEF, U+nFFFE and U+nFFFF”. (Pennington et al., 2014) Some of them originate from Arabic presentation form blocks.[47]
As ooRexx does not support UTF-8 encoding per default, the help from Java through

46 The code **`U+0000`** represents **`NULL`**.
47 Information about Unicode characters and non-characters can be retrieved at www.fileformat.info. For example U+FDD0: www.fileformat.info/info/unicode/char/fdd0/index.htm.accessed accessed on 2 September 2014

BSF4ooRexx can be used. As already described in the introduction, DBusooRexx first tries to use Javas translation function and if it is not available use a fallback mode. A conversion can be effected with the command `stringToUtf8('äöü')` for example.

Tests with different String values were implemented the same way like the previous test before, including invalid values in order to assess whether they are identified correctly. The assertion with .nil values showed a difference in contrast to the previous inspected integer and double object types. As .nil is not supported by DBus, the safe default value for a string is an empty string and not a 0. It is nonetheless strange that DBus transports some non-standard characters flawlessly. The following example assesses two strings that use special signs from their language.

```
germanstring = "gemäß Übereinkommen, Behörde"
frenchstring = "à l’école, être évidente, la façon"
self~assertSame(germanstring,dbustest~ReplyString(germanstring))
self~assertSame(frenchstring,dbustest~ReplyString(frenchstring))
self~assertSame(frenchstring,dbustest~ReplyString(stringToUtf8(frenchstring)))
```
Script 16: Processing of non-UFT-8 characters

There was no error remarkable. The assertion worked as if the strings would only contain regular characters. It was possible to use the function stringtoUTF8, or make an assertion without it.
For the next test a string was defined that uses characters that are not allowed for DBus. (see Script 16)

```
string = xrange('F0'x,'FF'x)
self~assertEquals(string ,dbustest~ReplyString(string))
string2 = stringToUtf8(string)
self~assertEquals(string2,dbustest~ReplyString(string2)
```
Script 17: Using stringToUTF8 for non-UFT-8 characters

Execution of the first assertion results in an error:

```
Event:  [SYNTAX 93.900] raised unexpectedly.
   DBusooRexx/method/DbusBusCallMessage(), error 7: Rexx message argument
# [1], typeCode=[s]: DBus-API returned 'no memory' while appending message
argument / error position (1-based): [1] in full signature: [s] (maybe
typeCode not supported on platform?)
```

If the string is converted prior with `stringToUtf8`, (as indicated in the third line of script 16) the assertion works. That means that DBusooRexx is actually able to use the conversion function correctly, but it is not necessary to make an conversion to UTF-8 for all special characters. All printable characters listed on the UTF-8 table at the referenced link[48] could be processed correctly, even unpopular ones like:

```
self~assertEquals("®",dbustest~ReplyString("®"),)
self~assertEquals("½",dbustest~ReplyString("½"),)
self~assertEquals("Ø",dbustest~ReplyString("Ø"),)
self~assertEquals("ñ",dbustest~ReplyString("ñ"),)
```

48 A complete UTF-8 table can be found at: http://www.utf8-chartable.de/ accessed on 2 September 2014.

3.4.4 Byte

There are different ways to define a byte in ooRexx. Table 5 lists different representations and three sample bytes, a letter, a special sign and one of the most important characters for ooRexx programs. Table 5 also list some useful methods for conversion between the different representations of bytes, some of them will be needed later in an usecase example.

Character	Decimal	Hexadecimal	Binary
'A'	65	'41'x	'01000001'b
'$'	36	'24'x	'00100100'b
'~'	126	'7E'x	'01111110'b
Useful Functions	D2C (decimal to character) D2X (decimal to hex)	X2B (hex to binary) X2C (hex to character) X2D (hex to decimal)	B2X – (binary to hex)

Table 5: Different byte representations

The characters below decimal code 32 are so called non-printable characters or control characters. For example 27 represents escape, 11 represents a vertical tab. Bytes with the decimal character code 32 to 127 are representing all characters that can be found on a standard English language keyboard.

The testgroup comprises all different possibilities to define a byte within their assertions. That means all four different representations are assessed (For example `self~assertEquals("$",dbustest~ReplyByte("24"x), "24x == $"`). For the assertions it is not necessary to “convert” the Byte representations. For example it is always possible to expect the symbol `'A'`, although its binary representation was sent to be returned (`'01000001'b`). It makes no difference whether the assertion is done via `assertSame` or `assertEquals`, both return true. In contrast to the strings presented earlier, a byte must not contain a non-UTF-8 character. Following assertion `self~assertEquals("ü",dbustest~ReplyByte("ü"))` fails and DBusooRexx informs that the value `ü` cannot be converted to a byte value. If its decimal representation is used instead of the character representation:
`self~assertEquals("ü",dbustest~ReplyByte(d2c(220)))`, ooTest emits another error message:

```
  Failed: assertEquals
    Expected: [[Ü], identityHash="17564778517334"]
    Actual:   [[�], identityHash="17564778351770"]
```

Null values (.nil) for bytes can be asserted with `self~assertEquals("00"x, dbustest~ReplyByte(.nil))` for example.

3.4.5 Signature

Signatures are restrained to a maximum length of 255. If the limit is surpassed, (the signature in the following example was defined with a length of 256 characters) following error message is thrown:

```
Event:  [SYNTAX 93.900] raised unexpectedly.
    DBusooRexx/method/DbusBusCallMessage(), error 7: Rexx message argument # [1],
typeCode=[g]: DBus-API returned 'no memory' while appending message argument /
error position (1-based): [1] in full signature: [g] (maybe typeCode not
supported on platform?)
```

Signatures are following strict rules for being valid. It is allowed to use any of the type-codes presented in Table 1 on page 28, but not in any given combination.

If an array signature is defined with a sole `'a'`, any method call to it will result in an error as an array always need defined what object types it carries[49]. That means asserting `self~assertEquals("a",dbustest~ReplySignature("a"), "array signature")` results in the same error code listed above (Syntax 93.900). It is possible to define a signature like 'as', denoting an array of strings. But if cascaded signatures are used, it is not allowed to use an 'a' at the end. For example `self~assertEquals("anton",dbustest~ReplySignature("anton"))`, is valid whereas `self~assertEquals("tina",dbustest~ReplySignature("tina"))`, invokes given error.

An additional specialty for ooRexx programmers is that capitalization matters within a signature. An assertion with `'A'` as signature results in an error.
It was already stated that defining introspection data and handling signatures is a rather difficult task. Signatures do not show any additional specialty, except that using irregular signatures always result in following debug message:

```
process 3259: arguments to dbus_message_iter_append_basic() were
incorrect, assertion "_dbus_check_is_valid_signature (*string_p)"
failed in file ../../dbus/dbus-message.c line 2608.
This is normally a bug in some application using the D-Bus library.
```

Of course all assertions that are known to produce errors are caught with `self~expectSyntax(93.900)`. But this debug information gets displayed by DBus and does not stop the script from execution.

3.4.6 ObjectPath

The object Path is used for referencing a DBus object instance. According to its definition, an object path may be of any length, therefore the first test defined an object path according its syntax requirements with `longpath = '/oorexx/dbustest'~copies(1000)`. As expected no error occurred. The syntax rules for an object path are simple, it must start with a `“/”`, followed by letters (A-Z or a-z) and/or digits (0-9) and there must not be a `“/”` at the end of the path. If a .nil value is sent, DBusooRexx automatically translates it to a root object path (`/`). All tests assessing intentionally wrong object paths were identified and caught correctly.
Therefore nothing special revealed during the assessment of objectPaths.

3.4.7 Arrays

Arrays are probably the most common type of container among different programming language as objects are simply arranged by their index, one after one.
For the first test, the method ReplyArray was defined to return arrays with the signature `'a'`, as it was just mentioned that it is necessary to define an additional type-code for an array, the assertion: `self~assertEquals(.array~new, dbustest~ReplyArray(.array~new))`, did not

49 Although **`'a'`** is a valid typecode, it is not a valid signature on its own.

invoke any error, whereas calling the same method with `self~assertEquals(.array~of(1), dbustest~ReplyArray(.array~of(1)))` results in errors that are able to tear down the testserver and therefore make all following assertions resulting in failures because the server is not available anymore. (See following two errors, first from ooRexx and second from ooTest)

```
REX0645E: Error 93.900:  DBusooRexx/DbusMessageLoop, panic! Unexpected
marshalling error for return value [an Array] with signature [a] by
service object [/oorexx/dbus/ooTestServer] after returning from Rexx
method [ReplyArray] (interfaceName=[oorexx.dbus.ooTestServer])
```

```
[SYNTAX 93.900] raised unexpectedly.
    DBusooRexx/method/DbusBusCallMessage(), error 12:
error.name=[org.freedesktop.DBus.Error.MarshallingReturnValue],
error.message=[Rexx service object [/oorexx/dbus/ooTestServer] supplied
a return value that could not be marshalled with signature [a]]
```

As these errors teared down the testserver, they were followed by hundreds of lines denoting:

```
error 12: error.name=[org.freedesktop.DBus.Error.ServiceUnknown], error.message=[The
name oorexx.dbus.ooTestServer was not provided by any .service files]
```

This test was only useful to prove that using `'a'` as signature return value is not allowed. This method was then changed to return an array of strings with the signature `'as'` instead of `'a'`.
On examining arrays, the example from the introduction, demonstrating that `assertEquals` and `assertSame` cannot be used interchangeably was tested as well. Therefore an array was defined, assessed with `assertEquals` which was found to be true and then assessed with `assertSame`. Although carrying the same values, the sent and returned array were not identical, therefore `assertSame` failed as expected, which was then changed to `assertNotSame` to incorporate this kind of assertion in the test cases as well.

Whereas integer values could be marshalled and demarshalled without problems within an array, processing arrays of doubles demonstrated unexpected behavior.
During the assertion of doubles it was already demonstrated that the value 1.0 got translated to 1. Nonetheless `assertEquals` worked to assess both, 1 and 1.0. Incorporating the value 1.0 into an array makes the assertion fail.

```
array = .array~of(0.1, 1.0)                         -- 1.0 is converted to 1
self~assertEquals(array, dbustest~ReplyArrayofDouble(array), "0.1 , 1.0")
```

This assertion results in the following message:

```
Failed: assertEquals
    Expected: [[an Array], identityHash="17458577806574"]
    Actual:   [[an Array], identityHash="17458577768412"]
```

Testing the values of the array with a loop and say instructions revealed that it is the different representation of the same value, (1 instead of 1.0) that disables the array from assessing its equality.[50]
Therefore the assertion for arrays of doubles was adapted to unwrap and assert each value of the sent and received array individually. (see Script 18)

50 I submitted a bug report and was told that this is expected behavior and will not be fixed. (https://sourceforge.net/p/oorexx/bugs/1272/)

```
retarray = dbustest~ReplyArrayofDouble(array)
do i=1 to array~length
  self~assertEquals(array[i], retarray[i])
end
```
Script 18: Workaround for assessing arrays of doubles

Same is valid for large double values, if for example the double with the value `999999999.9` is returned through DBus, it is converted to `1.00000000E+9` what produces errors unless unwrapped and individually asserted as demonstrated in the following example.

```
double1 = 999999999.9
double2 = 1.00000000E+9
self~assertEquals(double1, double2)
```

3.4.8 ByteArrays

Array of bytes are commonly used by different services on the DBus. Binary data is transported that way and in an example presented later, an USB device returns its mountpath as byte array. A byte-array can be defined with `.array~of('T','E','S','T')`. Some DBus debuggers[51] represent byte-arrays like `(84,69,83,84)`. DBusooRexx provided two different settings for handling bytearrays with the property **`unmarshalByteArrayAsString`**. If set to `.true`, the following assertion works.

```
.dbus~session~unmarshalByteArrayAsString = .true
self~assertEquals('TEST',dbustest~ReplyArrayofByte(.array~of('T','E','S','T')))
```
Script 19: Unmarshall byte array as string

The default setting is `.false`, which would return the byte array as it was received. Within the testgroup both switches were tested and the result was converted successfully.

3.4.9 Struct

A struct is a universal container that allows to carry any valid object type. The testserver used for the assessments, provides the method `ReplyStruct` with the signature `(si)`.
A valid struct for the given signature can be defined with `struct= .array~of('test',1)`
A Test with a value, set intentionally too low for an integer32, (described earlier) resulted in an expected failure. The next test with two .nil arguments showed expected conversion as listed in Table 6.

```
struct = .array~of(.nil,.nil)
struct2 = dbustest~ReplyStruct(struct)
self~assertEquals("", struct2[1])
self~assertEquals(0, struct2[2])
```

If a .nil value is sent instead of the expected array, the values within the struct (as gathered through the signature) are converted to their safe defaults. Therefore the same test described above works to assess a .nil struct.

3.5.10 Dict

A DBus dict is similar to a struct, but has a few more restrictions. A dict Entry is an array consisting of two elements, therefore a dict entry is always a key-value pair.
The first element, being the key, must not be a container, but a string. (Pennington et al., 2014) The testservice provides a method called ReplyDict with the signature `a{si}`. Test with

51 Both tested DBus debuggers use decimal instead of character representation.

regular string and integer values showed nothing special. A valid dict for `a{si}` is for example:

```
dict = .directory~new~~put(-2147483648, "lowest value -2147483648")
```

As the string is the key for the DBus dict and the put method of ooRexx is defined as `put(item, index)` this is valid. A counter-test with the order “string, integer”, as indicated in the introspection signature failed as expected.

It was already demonstrated that .nil is converted to 0 for any integer value. (see Table 6)
Therefore a test was created that passes a .nil value and expects the dicts to be not equal:

```
dict = .directory~new~~put(.nil,".nil is converted to 0")
self~assertNotEquals(dict, dbustest~ReplyDict(dict), "test array")
```

As there was no error remarkable, the author unwrapped the integer value and expected it to be 0 was is the case:

```
dict2 = dbustest~ReplyDict(dict)
self~assertEquals(0, dict2[".nil is converted to 0"])
```

Another test with an integer value that passes its value range demonstrates expected behavior, namely throwing an error.

```
self~expectSyntax(93.900)
dict = .directory~new~~put(2147483648, "highest value passed")
self~assertEquals(dict, dbustest~ReplyDict(dict), "test array")
```

If the dict itself is passed over as .nil, the return value consists of an empty directory. Therefore following assertion works:

```
dict=dbustest~ReplyDict(.nil)
self~assertEquals(.directory~new,dict)
```

3.4.11 Variant

A variant is a special type of container that carries the signature of a value and the value itself within an array. The function `dbus.box()` “converts” a value into a variant. For example `dbus.box('i',2147483647)` allows to define that the value carried is of type integer32. This information is only used by DBus to get knowledge about the object type. The return value of a variant defined that way is the integer value only.
For the testgroup a boxed and an unboxed variant were created. For example:

```
boxedvariant =.array~of(dbus.box('b', .true), dbus.box('n', 32767))
unboxedvariant=.array~of(.true, 32767)
```

The return value of the the service `ReplyArrayofVariant(boxedvariant)` was assessed with the `unboxedvariant` and asserted as equal. If a value surpasses its limits, for example an integer value that is too large, DBus returns an error message what proves that the value was unwrapped correctly.

In order to test how .nil values are converted, an array with boxed .nil values for all simple object-types (dbus.box('b', .nil), dbus.box('n', .nil), …) was created and the return value of this array was asserted with the expected save default values for .nil for the given object-type.
All object-types were converted as expected, except the byte. It was demonstrated earlier that a .nil byte can be assessed with `self~assertEquals("00"x, dbustest~ReplyByte(.nil))`, but if transported within the array and boxed with `dbus.box('y',.nil)`, the expected value has to be an empty string instead of `"00"x`.
The next chapter will cope the .nil translation for all object types.

3.5 DBus and NULL Values

DBus does not support Null values in its messages. Many programming languages nonetheless need this kind of value to represent that there is no value.
In order to process Null vales correctly, DBusooRexx automatically translates them to values that can be processed via DBus. Table 6 illustrates the DBusooRexx equivalent of .nil for the given object type. For the testgroup, it was assessed how the return value looks for no argument and for a .nil argument.

In order to assert them within a loop, following select statement was used to define the expected value first, otherwise the .nil values would not be assessed according to their object type correctly.

```
select
when type='g' then null = ""
when type='y' then null = "00"x
when type='s' then null = ""
when type='o' then null= "/"
otherwise
  null=0
end
```
Script 20: Null value conversion for assertions

For example the following assertion works for all `TestServices`:
`self~assertEquals('00'x,dbustest~ReplyByte('00000000'b), "'00000000'b == .nil")`
Table 6 shows what has to be expected if a .nil value is sent and returned with the same signature from any of the `TestServices`.

DBus Object Type	DBusooRexx Null (.nil) value representation
array	empty array
boolean	0
byte	"00"x
double	0
int16	0
int32	0
int64	0
objpath	/
signature	empty string (“”)
string	empty string (“”)
unit16	0
unit32	0
uint64	0
variant	empty string (“”)
structure	carried object types are converted to their safe default values
map/dict	empty .Directory

Table 6: DBusooRexx null value representation

3.6 DBusooRexx's ReplySlotDir

DBusooRexx provides a so called slotDir argument that comprises some information about the caller and the sender of a DBus message. It can be activated and deactivated with the booleans `.true` and `.false`. If activated with `.dbus~session~makeReplySlotDir=.true`, any message call will get added an additional directory which contains information like signature of the call, sender and receiver id, a timestamp and some more information.[52]
In order to assess both sides of this “information package”, a testmethod `TestReplySlotDir` with a string as input value and a dict `'a{sv}'` as return value was implemented. This enables to capture the SlotDir from the message call to this method (named cSlotDir) and return it to the caller along with the new SlotDir that got created for the message response. The following output demonstrates this two different views on the SlotDir, rslotDir is the new slotDir created due to the method call from the server side and cslotDir represents the slotDir, the caller issued.

52 True is the default setting for slotDir.

SlotDir agruments	ReturnSlotDir (rslotDir)	CallerSlotDir (cslotDir)
MESSAGETYPENAME	method_return	method_call
SIGNATURE	a{sv}	s
MESSAGETYPE	2	1
OBJECTPATH		/oorexx/dbus/ooTestServer
INTERFACE		oorexx.dbus.ooTestServer
NOREPLY	1	0
AUTOSTART	1	1
DATETIME	2015-05-14T18:11:59.228936	2015-05-14T18:11:59.228611
SENDER	1.154	1.153
SERIAL	11	10
CONNECTION	a Dbus	a Dbus
MEMBER		TestReplySlotDir
DESTINATION	1.153	oorexx.dbus.ooTestServer

Figure 6: Comparison of two different views on DBusooRexx's ReplySlotDir

As the values of both slotDirs indicate, there is enough information for implementing some assertions. For example:

```
self~assertSame(  's'  , cslotDir['SIGNATURE'], 'assert the SIGNATURE')
self~assertSame('a{sv}', rslotDir['SIGNATURE'], 'assert the SIGNATURE')
```

The bus id looks like :1.480 and serves to uniquely identify a DBus connection, therefore is is ideally suited for assertions like:

```
busid = conn~busName('unique')
self~assertSame(busid, cslotDir['SENDER'], 'assert id of SENDER')
self~assertSame(cslotDir['SENDER'], rslotDir['DESTINATION'], 'assert SENDER')
```

If the SlotDir is deactivated on the DBus connection, it must not appear in a method call:

```
.dbus~session~makeReplySlotDir =.false
 rv = dbustest~TestReplySlotDir('test')
   rslotDir = rv[2]
   self~assertSame(.nil, rslotDir, 'ReplySlotDir deactivated')
```

The slotDir proves to work as expected and is definitely a nice feature to obtain further information about a message caller. With this information it is possible to react differently to method calls, depending on who the caller was.

3.7 DBus Properties

Implementing and handling properties is easy with DBusooRexx, there are two possibilities to implement properties. Either provide them via a common `::attribute` directive or return them via a `::method` directive.

If there is only one property defined, giving access to it is easy. The introspection data has to provide a line such as `<property name="ServiceName" access="read" type="s"/>`. Then the “bridge” between DBus and the ooRexx method can for example provided with:[53]

```
::method Get
  return self~serviceName
```

If the introspection data is provided by subclassing DBusServiceObject, the method `Get` is already provided. Properties can be defined as regular ooRexx `::attribute` and are referenced by:

```
::method info
  expose info
  return info
```

The parameter access within the introspection line denotes how and if the property can be manipulated. Valid parameters are 'read', 'write' and 'readwrite'. A possibility to implement a readwrite property in ooRexx is provided by the author of DBusooRexx in his examples.[54]

```
::method info         /* the Rexx attribute is a 'readwrite' property!     */
  expose info
  if arg()=2 then     /* argument (+ slotDir), then behave as setter method*/
     use arg info     /* fetch argument and save with attribute            */
  else                /* no argument, then behave as a getter method       */
     return info      /* return current attribute                          */
```

It was mentioned earlier that the `::method unknown` can be added to any script as backup if an unknown message call arrives. In this example, adding `::method unknown` intercepts all calls to the properties. If the `::method unknown` is used anyway, both `Set` and `Get` methods must be implemented in order to interact with properties.[55] That is also true if the ooRexx class does not subclass DBusServiceObject, but returns the introspection data in its `::method Introspect`. A possible implementation for `Get` with two properties could look like:

```
::method Get
  expose MethodName info
  use arg caller, propertyname
  if (propertyname='Info') then return info
  if (propertyname='MethodName') then return MethodName
```

An implementation with direct access to properties as well over the DBus default method `org.freedesktop.DBus.Properties` is demonstrated in Script 21 .

```
node=.IntrospectHelper~new
if=node~addInterface('org.freedesktop.DBus.Properties')
if~addMethod('Get','ss','v')
if~addMethod('Set','ssv','')
if=node~addInterface('org.rexx.demo')
if~addProperty('Info', 's', 'readwrite')
if~addProperty('Version', 'i', 'write')
introspectData=node~makeString
self~init:super(introspectData)
```
Script 21: Providing access to DBus Properties

The most convenient method to provide properties is to define them via the `::attribute` directive in a subclass of DBusServiceObject. If using the `::attribute` directive, ooRexx automatically creates a get and set method for the object instance variable.

53 Is is only necessary to implement the `Get` and `Set` method with ooRexx program code if DBusServiceObject is not subclassed like `TestService3`.
54 See DBusooRexx, Script in folder /examples/demoHelloService3.rex, line 89ff.
55 More details about properties at: http://dbus.freedesktop.org/doc/dbus-specification.html#standard-interfaces-properties. accessed on 2 September 2014.

There are also different ways to get access to properties. The easiest way probably is to retrieve or set a value by using the object reference followed by the name of the property. For example: `own~variablename` for querying and `own~variablename('newvalue')` to set a new value. It is also possible to use the `Get` and `Set` methods of `org.freedesktop.Introspectable`.

As mentioned earlier, properties can be defined with read, write or readwrite access. That means that a write variable must not allow to be queried. If the property “Version” of Script 21 is called with `own~Version`, DBusooRexx gently informs: `argument missing for a "write" property`. `own~Version(2)` is a valid command. Properties can also be changed with `own~variablename=variable`. This is the default set method automatically created from ooRexx for handling attributes. The testgroup revealed that handling properties that way works flawlessly as well.

If the `Get` method of `org.freedesktop.Introspectable` is used, the name of the interface is necessary as additional parameter (for example `own~Get(interfacename,'Info')`). According the signature, the `Set` method needs a string and a variant as object types, the first two parameters are (analogously to the `Get` method) the interfacename and the name of the property, the third parameter is a variant. Dependent on the complexity of the object type, it might be necessary to use the dbus.box instruction like: `own~Set(interfacename,'SetInfo', dbus.box('s','newvalue'))` . In this example it would be enough to just use the string without boxing it as DBusooRexx translates the object on the fly..

Within the testgroup three properties are used with different access and different definition methods. (Script 22)

```
<property name='Info' access='read' type='s'/>
<property name='SetInfo' access='write' type='s'/>
<property name='MethodName' access='readwrite' type='s'/>
```
Script 22: Attributes, the testgroup interacts with

The property `MethodName` has read and write access. When a test is carried out in the testgroup, this property of the server is changed to the name of the calling method. For example:

```
dbustest~MethodName('test_dbusname')
```

When all assertions in this testmethod are carried out, the prior changed property is asserted with: `self~assertSame("test_dbusname",dbustest~MethodName)` to test read and write access. When trying to set a property that was defined as being read-only with `Info(newvalue)`, DBusooRexx raises an error condition and informs about wrong interaction with this property by issuing:

```
Incorrect interaction with property "oorexx.dbus.ooTestServer.Info": attempt to
assign the value "oorexx.dbus.ooTestServer" to a "read"-only property
```

All properties are tested if they behave according to their signature. A readonly property must not allow its value to be set, whereas a writeonly property must not allow to read its value. This was effected by using them incorrectly and expect an error.[56] (see Script 23)

56 These assertions must not be tested in this sequence, as after the first syntax error, the testmethod quits and the second assertion would not be executed anymore.

```
self~expectSyntax(93.900)
say dbustest~SetInfo

self~expectSyntax(93.900)
dbustest~Info('set readonly value')
```
Script 23: Wrong interaction with properties

During the assertions with ooTest the property set and assertions with get did sometimes not work correctly. (meaning that every time another assertion failed that was already assessed to work multiple times before.)

It showed that it is necessary to let DBus time to set the value before inquiring it again. A short syssleep is enough to make all assertions succeed every time. This approach was used for the test method that tests the setting and getting of properties directly.

3.8 DBus Signals

Signals are very useful to broadcast any information around the DBus, if any other service is interested in receiving this signal it can be configured to listen to it. This also means that a signal never has a direct receiver, but is broadcasted around the DBus. Emitting a signal can be effected with:

`.dbus~session~message("signal",objectPath,interface,methname,objecttype,message)`.

The object type is defined with its signature, if multiple object types are defined, each has to be provided after the signature definition, separated by commas. It does not matter what objectpath or what interface is defined as long as they are defined according to their syntax requirements. It is even possible to emit signals to known members.

On the first sight this might be perceived as a security flaw. Further investigation reveals that signals are nonetheless distinguishable. See following output of dbus-monitor:

```
signal sender=org.freedesktop.DBus -> dest=(null destination)
path=/org/freedesktop/DBus;interface=org.freedesktop.DBus;member=NameOwnerChanged
   string "oorexx.dbus.ooTestServer"
   string ""
   string ":1.279"

signal sender=:1.279 -> dest=(null destination)
path=/org/freedesktop/DBus;interface=org.freedesktop.DBus;member=NameOwnerChanged
   string "oorexx.dbus.ooTestServer"
   string ""
   string ":1.279"
```
Script 24: Signal example captured by dbus-monitor

Although the signal was faked to look exactly like the one from DBus, the name of the sender differs (marked red). If trying to acquire the name `org.freedesktop.DBus`, a message appears that informs about owning the name `org.freedesktop.DBus` is not possible.

If signals are not defined correctly within their introspection data, a DBus debugger like D-Feet is not able to list it, but does not issue any failure notification.

`TestService4` was instructed to provide a Signal with `~addSignal('Exit','s')`. With this setup the signal was not visible for D-Feet. It showed that by changing the definition from `~addSignal('Exit','s')` to `~addSignal('Exit',)`, D-Feet was able to list all services, signals and properties correctly.

In order to test signal emission as well as signal immission, the testserver provides a signal called Ready. Any signature that is defined according to its rules is valid for a signal.[57] In this example it was decided to use two arguments for demonstration purpose. The signature `'sb'` denotes String and Boolean. See following example output of how the emitted signal is monitored by dbus-monitor.

```
signal sender=:1.156 -> dest=(null destination) serial=3
path=/oorexx/dbus/ooTestServer; interface=oorexx.dbus.ooTestServer; member=Ready
   string "started"
   boolean true
```

A `DBusSignalListener` that defines the methods “Ready” and “Exit” was added to the testclients `::method setUp` that was described earlier. The intention of this approach is, as the testserver (and all its services) is started within this method, to wait until the services are accessible and the setup process is therefore finished. The method Ready of the class `DBusSignalListener` can be implemented as following:

```
::method Ready                          -- changes .local~ready
  use arg text, boolean
  say 'server:' text '=' boolean
  .local~server.ready = boolean  -- set ready to .true
```

Within the `::method setUp`, the testsuite is instructed to wait until the attribute Ready was changed through the `DBusSignalListener` in its Ready method. (see following script)

```
/* wait until program fully initializes */
 do while \.server.ready
  call syssleep 0.5
 end
```

The server script on the other side effects its signal emission soon after all `TestServices` were added to the DBus connection and all private servers were started:

```
conn~serviceObject('add', objectPath, testserver)
conn~serviceObject('add', "/oorexx/dbus/TestService2", testserver2)
conn~serviceObject('add', "/oorexx/dbus/TestService3", testserver3)
conn~serviceObject('add', "/oorexx/dbus/TestService4", testserver4)
conn~serviceObject('add', "/oorexx/dbus/TestService5", testserver5)
privateserver~startup
privateserver2~startup

conn~message('signal',objectPath,interface,'Ready','sb','started',.true)
```
Script 25: Signal emission after startup of the serverscript

As the testgroup starts its assertions and all services are available, signal handling was tested to work in this setup.

3.9 DBus Errors

If a service is called the wrong way, DBusooRexx raises an error and informs about what went wrong. An example already described is a service call with a wrong number of arguments passed. Errors can be raised intentionally with additional information, allowing the calling program to react appropriately.
For example a method is called that makes an illegal mathematic operation. If the error is not passed over to the caller, it would not know what went wrong as the service would not

57 The first concept was to name the signal start, but as already mentioned by the author of DBusooRexx in the example script “signalListener.rex”, using signal names that have the same name as methods of the .Object Rexx class forces to implement this method within the SignalListener as well.

answer.

```
::method test_errorpropagation
  signal on syntax
  say 1/0    -- provoke a runtime error
  return
syntax:raise propagate
```

This method allows to intercept the ooRexx error message and direct it to the caller. Error messages do not have strict rules for their setup, they can use any name and carry any message. For example:

```
call raiseDBusError "org.freedesktop.DBus.Error.ExtremeError", -
'Sorry I give up, the error is too heavy!'
```

It is also possible to omit the error name or the error message at all. An example output (taken from D-Feet) for an unnamed error without any additional information yields in:

```
'GDBus.Error:org.freedesktop.DBus.Error.RexxServiceRaised: Error return with empty
body: '
```

The testgroup contains five different tests for error messages,[58] the first test method does actually provoke a real error that gets propagated within the DBus error message. The second test method raises an error and passes over a name and a body, the third omits one of them, the fourth omits the other and finally, the fifth omits all information.
The assertions showed that all errors were thrown correctly by all services.

3.10 Start Service by Name

As services are exchanging messages over DBus they need to be available in time to guarantee that a method call receives its destination. One possibility is to always start all services that might be needed by default, but in order to reduce overhead, a minimal system can be configured that only starts services that are really needed. Internally, a dbus-daemon-launch-helper is called and an argument, the name of the service to start, is passed. This name then gets searched for available configuration information within the directories **`/usr/share/dbus-1/system-services`** or **`/usr/share/dbus-1/services`** for system or session bus.[59]
A service file contains the name of the service, the location of the service (path) and optionally an user statement. (for example `User=ftp` for invoking the program with the permissions for the given user)[60] [61]

```
[D-BUS Service]
Name=oorexx.dbus.ooTestServer
Exec=/home/zerkop/MasterThesis/DBusooRexx/ooTestDBusServer.rexx
```

Script 26: Example ooTest.service configuration

Saving the file in the script 25 with the name `ooTestserver.service` with administrative privileges in the directory `/usr/share/dbus-1/services`, enables the program to be started on demand by DBus.
Additionally to the service file configuration, it is necessary to make the script executable if it is not already. This can be effected by adding a shebang to the first line of the script

58 These tests were adapted from testErroneousServer.rex by Rony Flatscher.
59 The name of the configuration file is irrelevant as long as it has the ending .service. Only the name defined within the configuration files is important for the DBus infrastructure in order to start it..
60 It is also possible to elevate user rights by requiring administrative privileges with user=root. This is probably an interesting method to gain deeper access to the operating system through DBus.
61 http://dbus.freedesktop.org/doc/system-activation.txt, accessed on 2 September 2014.

(#!/usr/bin/rexx) in order to let the operating system know how the script has to be invoked and the script has to be defined as executable for the operating system as well.[62]
Whether the configuration file was created successfully can be queried with the method ListActivatableNames of org.freedesktop.DBus. If the name oorexx.dbus.ooTestServer is listed as available service, the method of org.freedesktop.DBus, named StartServiceByName can be used to start the program. The parameters to be passed are the name and the integer 0 if the program should just be started and there is no interest in calling a specific method on it.
By adding configuration files for services it is possible to call a method of a program that is known not to be available at that time. The program will start and the service is available.

This is a very useful feature to start any program by demand in an elegant manner. It is not even necessary that the program has anything to do with DBus at all. DBus can be used as shortcut to start all services that are necessary. Although it is not possible to talk to the program over DBus if the program does not support that, it is possible to preconfigure any switch or enable a dynamic choice by creating multiple service files calling the same program with different parameters and choose amongst them which to start at runtime.

3.11 Penetration Tests

Penetration tests are envisaging to call every service subsequently or parallel very often within a short time range. The goal behind that is to assess when a service reaches its service-limit and is able to handle the demanded requests on time. For this purpose a few limits of DBus have to be kept in mind, that is for example the response time and of course the overall connection limit a service is able to handle.
The default configuration file of DBus (for the session bus) can be found under the path **/etc/dbus-1/session.conf**. Do not change anything in the file unless you know exactly what you are doing. Any misconfiguration results in an unbootable state for the system, making it necessary to log in to a virtual console and revert the file to its original state.
The file states some of the limits imposed for the connections. (See Script 25)
For this testgroup and this paper, there are no tests of this kind implemented so far.[63]

```
<!-- the memory limits are 1G instead of say 4G because they
can't exceed 32-bit signed int max →
  <limit name="max_incoming_bytes">1000000000</limit>
  <limit name="max_incoming_unix_fds">250000000</limit>
  <limit name="max_outgoing_bytes">1000000000</limit>
  <limit name="max_outgoing_unix_fds">250000000</limit>
  <limit name="max_message_size">1000000000</limit>
  <limit name="max_message_unix_fds">4096</limit>
  <limit name="service_start_timeout">120000</limit>
  <limit name="auth_timeout">240000</limit>
  <limit name="max_completed_connections">100000</limit>
  <limit name="max_incomplete_connections">10000</limit>
  <limit name="max_connections_per_user">100000</limit>
  <limit name="max_pending_service_starts">10000</limit>
  <limit name="max_names_per_connection">50000</limit>
  <limit name="max_match_rules_per_connection">50000</limit>
  <limit name="max_replies_per_connection">50000</limit>
```

Script 27: Excerpt of /etc/dbus-1/session.conf

62 This can be effected with chmod +x filename.
63 Although that is no penetration test, the author can guarantee that some of the assertions have been executed a few thousand times so far.

3.12 Limitations

The testgroup was only tested on Debian-based Linux-systems, on Ubuntu and some of its derivates with 32 and 64-bitness. As DBus is originated from and is dedicated to Linux-systems, this application comes naturally. But there are ports for other operating systems as well, that were not tested extensively. For example there are ports for Windows or MacOSX. On Windows system, it was only tested successfully to start DBus scripts but no testgroup run was carried out so far.
As programing complete tests requires profound understanding of the DBus language binding and DBus itself, it is probably possible that some features were unintentionally omitted from the tests.

The testgroup is of course a good candidate to be further enhanced by adding additional tests. Especially complicated containers have not been tested thoroughly so far. The testgroup assesses only simple forms, not cascaded ones. It would theoretically be possible to assess any combination of any signature in any container, but that is of course not feasible. Therefore basic tests were effected that are of course to be enhanced over time.
During the tests a few errors have been revealed and got corrected by the author of DBusooRexx, but of course it can never be guaranteed that there are still errors that have not been found so far. DBus itself is also not immune to bugs and might run through changes over time. For the testing, Ubuntu's DBus version from the standard software repository was used. The ooRexx DBus binding, DBusooRexx, was assessed in its versions 099.20110912 to 100.20140807, which were compiled with DBus version 1.4.6.[64] A closer look at the version used within Ubuntu14.04 and the official releases of DBus reveals that the shipped version best matches the dbus release of 1 November, 2013. The most current version at the time of writing was dbus-1.8.6, released on 2 July 2014.[65]

64 Querying of version is possible via `say "DBusVersion:" DbusVersion()`.
65 DBus releases: http://dbus.freedesktop.org/releases/dbus/, accessed on 2 September 2014.

4. Exploiting DBusooRexx In a Practical Setting

The following section provides script examples that are using the infrastructure of DBus provided through DBusooRexx. It covers connections to the system and the session bus, listening for signals, process “complicated” signals and also gives insights how to code a solution for a problem. These scripts should not be seen as fully featured programs, but are intended to demonstrate the ease of use of DBusooRexx and a few very interesting features of DBus.

These script examples can be extended easily. Some features introduced in one of the examples might be useful for the other as well. For example the possibility to only react on a certain device, which will be demonstrated later.

As these examples interact with other DBus objects, it can be seen as a test in the wild. It has already showed to be useful to test DBusooRexx “in the wild”, as a severe error concerning signal handling was uncovered (which was corrected in the actual version).

4.1 Script Example: Automated Media-Player Control with Bluetooth and DBus

A so called “Hello World” program in the context of interprocess communication probably is controlling a Media-player. Displaying a text message on a screen does not offer the same understanding of controlling a program, as the message windows, appearing on the screen could have been built purposely for this very appearance, whereas the media player rather feels like a standalone program. Therefore controlling a “foreign” program impresses much more.

Concerning demonstration of DBus concepts, the following script will make use of a private DBus server connection over Tcp/Ip, and control and coordinate a media-player over its DBus services on the session bus. Additionally, two interesting interfaces are introduced. One for controlling various media-players, the other for Bluetooth communication, namely MPRIS and bluez. The latter also demonstrate how services on the system bus can be deployed.

4.1.2 Listening to Signals and Invoke Methods over Network

The use-case is the following: there a two machines which are distributed over rooms (e.g one is in the living room, the other in the working room), both have their own sound playback capabilities (connection to speakers). A music player playbacks sound on the main computer.

Now it would be useful if the computer tracks your position and decides to stop the playback on the machine your are departing from and resumes the playback on the other machine you are heading to, the music just follows you. Figure 7 illustrates this usecase.

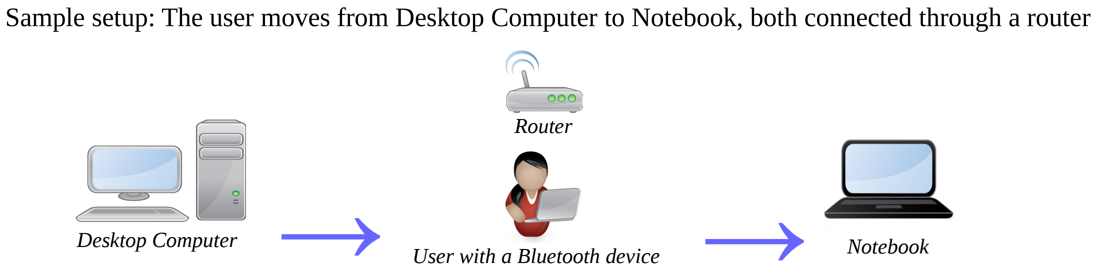

Sample setup: The user moves from Desktop Computer to Notebook, both connected through a router
Router
Desktop Computer
User with a Bluetooth device
Notebook

Figure 7: Two computers and an user with a Bluetooth device moving between

4.1.2 Prerequisites

For this setup two computers, both disposing of own speakers and an active Bluetooth adapter[66] on one machine is needed. A third device with Bluetooth capabilities, carried by the user is necessary as well, serving as the device to be tracked.

The Bluetooth device needs to be instructed to keep its discovery active; most devices will accept this settings. The default setting is to keep it alive only for a given period of time, usually a few minutes. As no connection to the device is established, but only its signal strength is observed, the device would suddenly disappear when its discovery period is over. Any mobile phone is perfectly useful for this purpose. Regretfully, there are devices which will not keep discovery alive[67].

For better understanding, the term main computer is used for the Computer the music plays and the user is sitting before, the term target computer refers to the remote computer (Notebook in this setup). The script running on the remote computer is called `MPRISremoteServer` and the script running on the main computer is called `MPRISremoteClient`.

4.1.3 The Media-Player-Interface Standard MPRIS

The abbreviation MPRIS stands for **M**edia **P**laye**r** **I**nterface **S**tandard which aims to act as a common interface to all compliant audio-players. Most desktop environments already support this standard and integrate MPRIS within their GUI. Therefore it does not matter what media player is installed on the system, all of its basic functionalities are controllable over DBus, as long as MPRIS compliance is provided. The standard has evolved over time, now being available in version 2.2, which offers extended control functionality and has a growing number of compliant players.[68] In addition to its own interface, MPRIS emits status messages over the default `org.freedesktop.Dbus.Properties.PropertiesChanged` signal, enabling desktop environments to display basic information about playback status, title, volume and so on in their notification area.
For demonstration purpose, the vlc player was chosen as it is available for any common (desktop) operating system and offers great compatibility with many different audio and video-codecs. Since vlc player dropped its MPRIS standard in favor of MPRIS2 (Ennaime et al., 2012), the bus name changed to `org.mpris.MediaPlayer2`, followed by the player name.

66 Most current Notebooks and some desktop PCs already have a Bluetooth adapter build-in.
67 A smart-watch from the smartdevice market leader is discoverable to two minutes only, what makes it useless for this setup, although this device would be perfectly useful for location tracking purpose.
68 A list of all MPRIS interfaces: http://specifications.freedesktop.org/mpris-spec/latest/fullindex.html, accessed on 2 September 2014.

MRPIS scripts will not be able to establish a connection to MPRIS2 anymore. During the changes from MPRIS to MPRIS2, DBus support was temporarily removed from the configuration options within the graphical user interface of vlc and could only be invoked over command line parameters. (`vlc --control dbus`) Since vlc version 2.06 this functionality can again be enabled within the GUI in the settings menu under the control interfaces tab.

Since MPRIS2 compliant players need not necessarily implement all methods or provide additional services, the introspection file of vlc has to be queried first.
DBusooRexx already ships an excellent tool for obtaining this kind of information, graphically refined and colorized to be easily readable and clearly represented. Executing `rexx dbusdoc.rex org.mpris.MediaPlayer2.vlc`, creates a HTML file, listing all services provided through the given address. This is probably necessary from time to time, as MPRIS2, although being a standard, very often changes and therefore renders (older) scripts useless.

The interesting part of the introspection starts with section two, where the bus name of vlc gets revealed. A few versions ago, a regular vlc bus address looked like `org.mpris.MediaPlayer2.vlc-7592`, 7592 being the process number of the instance. As this process number is only valid for a session, it has to be obtained dynamically every time. In the most current version of MPRIS2, the bus name is added the word “instance”. (`org.mpris.MediaPlayer2.vlc.instance7592`) Therefore being incompatible with the prior notation. But a positive development was remarkable as well, whilst vlc was only reachable if there is knowledge about its process number, the most current version allows to connect to `org.mpris.MediaPlayer2.vlc`, acting as proxy for the active instance. If the setup does not intend to create multiple instances, handling process numbers is not necessary anymore.

An interesting observation acquired through dbusdoc.rexx is, that one signature of vlc does not match the specifications of MPRIS2.[69] The reference implementation demands x as object type for the property position, referring to Int64, but dbusdoc.rex identifies the return value as Int32. This is no failure, but nonetheless demonstrates that some services have their troubles to be 100% compliant to the reference implementation.

4.1.4 The Linux Bluetooth Standard BLUEZ

The official Linux Bluetooth protocol stack is called bluez. It is part of the official Linux kernel since version 2.4.6.[70] and enables easy configuration of any Bluetooth device. Bluez is available for nearly any operating system. It became an unified standard. The most current version of bluez is 5.22, on Ubuntu 14.04 bluez is installed in version 4.101.
As bluez is integrated into the kernel, the DBus interface registers itself on the system bus. (Holtmann, 2006) investigated the bluez DBus API and also provided an example how to interact with bluez and the python DBus language binding.
bluez serves as an excellent example for using signals over DBusooRexx. There are two signals that are especially interesting to be intercepted by the ooRexx class for the given usecase which will be described later.

There are various ways to obtain the position of a person, respectively a device the person is carrying. GPS is the most popular positioning system. Bluetooth is especially useful if

69 http://specifications.freedesktop.org/mpris-spec/latest/Player_Interface.html#types, accessed on 2 September 2014

70 More information at: http://www.bluez.org/ the package **`bluez`** is probably available through the standard software repository. accessed on 2 September 2014.

connection to GPS satellites cannot be established, for example indoors.
Generally, a positioning system needs three anchor points in order to enable triangulation. GPS uses time information to calculate the position, the longer the signal needs to travel, the longer is the distance to the receiver. Given the distance to three known positions, the location can be calculated. Indoor positioning can be done via Bluetooth as well, but in that case the data for the position calculation is not the signal roundtrip time, but the signal strength. Since Bluetooth version 1.1, the signal strength is indicated and obtainable through the variable RSSI (Received Signal Strength Indication).[71]
Bluetooth offers four different signal related indicators, that can be used for localization, the RSSI value is among them. As this is the only value obtainable over DBus, it was decided to use it for this usecase, although using a RSSI value for localization purpose was investigated thoroughly and already found to be a poor candidate. (Hossain & Soh, 2007)
Usage of RSSI values have also been investigated by (Kikawa, Yoshikawa, Okubo, & Takeshita, 2010) who propose following exactly defined preconditions in order to obtain valuable results. They investigated the position of the bluetooth sender and receiver with different parameters (eg one time the bluetooth device is held in hand or put in pocket, position of the bluetooth sender changed, and many others) That implies that an exact setup has to be defined for each device individually every time.
If the signal is triangulated, meaning that the signal strength is measured from three fixed points, it would be possible to more precisely define the position, this comes close to an indoor positioning system. (Feldmann, 2003)
It would be possible to place three Bluetooth adapters at distant, known and fixed positions within a flat, eventually with USB extension cables, but for the usecase demonstrated, the exact position is of little interest.

4.1.5 Establish Connection to Two Media-Players Via DBusServer

Of course there are many possibilities to solve the problem presented in the usecase, not only concerning the positioning method. It would be possible to use the streaming capabilities of vlc and instruct the main computer to stream to its target, but there would be no need for implementing an ooRexx DBus backend on the target computer. As the intention of this section is to provide scripts and demonstrate capabilities of DBus, the following approach will be envisaged:
A server program runs on the target computer and provides two services, reachable via TCP/IP.[72] The first service called `Play` needs the URL of a file and its desired playback position as input parameters, the other one, `Stop`, acts according to its name and returns the url and current playback position of a song. The script also demonstrates how easy it is to establish a remote private connection via a DBusServer object. A few lines of code are enough to setup the server and make DBus services available for remote machines. (see Script 27) Note that the IP address of the machine is hardcoded in this example, it has to be replaced with the current IP of the machine. (`ifconfig` can be used on Linux systems to obtain network information) Another possibility is to connect to `org.freedekstop.NetworkManager` over DBus and acquire the current IP address dynamically, this procedure will be explained later (script 36).

71 The first external Bluetooth adapter used during the tests was a 12 years old version 1.0 device, that kept its RSSI value constant at zero.

72 It might be possible that security settings permit connection to DBus. Within a local network everything was tested to work without additional setup. Configuration for DBus can be looked up under /etc/dbus-1/

```
/* this script runs on target computer and offers
   music playback services for the main computer*/

ownaddress='tcp:host=192.168.0.15,port=23000,family=ipv4;'
vlcservice =.VlcService~new
server =.DBUSServer~new(ownaddress,vlcservice)
server~allowAnonymous=.true
server~startup
signal on halt       -- intercept ctl-c (jump to label "halt:")

say 'This script starts the vlc mediaplayer without graphical userinterface'
say 'Connection to this service is possible via:'  server~serverAddress
say 'Press return to quit'
parse pull enter

halt:                 -- a ctl-c causes a jump to this label
say 'shutdown server and mediaplayer and ' -
    'closing all connections to clients ..'
server~shutdown
vlcservice~Quitvlc
.dbus~session~close

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

::class VlcService subclass DBusServiceObject
::attribute vlc
::method init
  expose vlc
  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="oorexx.vlc">'  -
            '<method name="Play">'  -
              '<arg name=   "file"   direction= "in" type="s"/>'  -
              '<arg name= "position" direction= "in" type="x"/>'  -
              '<arg name=  "result"  direction="out" type="b"/>'  -
             '</method>'  -
            '<method name="Stop">' -
           '<arg name="data" direction="out" type="as"/>'  -
               '</method>'  -
          '</interface>' -
         '</node>'
 self~init:super(idata)      -- let DBusServiceObject initialize
```
Script 28: MPRISremoteServer (part 1/2) setting up a private DBusServer

The second part of the serverscript serves the methods that are provided through the DBus introspection information listed above. On the remote machine, the mediaplayer is started without GUI and the connection to it is established over a DBusProxyObject, where the stop and play commands are forwarded to. (see MPRISremoteServer (part 2/2) offering services via DBusServer)

```
  procId=getVlcId()  -- get procid of newest vlc or create an instance
  if procId="" then  -- no vlc instance found (and could not be created)
  do
    say 'no vlc instance available, could not create one either, aborting ...'
    exit -1
  end

-- MPRIS2 does not necessarily need a procID unless multiple instances are running
  busname="org.mpris.MediaPlayer2.vlc.instance"procId -- create MPRIS2 object
  objPath="/org/mpris/MediaPlayer2"              -- define object path
  vlc = .dbus~session~getObject(busname,objPath) -- get the vlc proxy object

::method Stop
  expose vlc
  url = vlc~Metadata['xesam:url']
  say 'Url to return =' url
  position = vlc~Position
  vlc~Stop
  return .array~of(url, position)

::method Play
  expose vlc
  use arg file, position
  say 'file to play:' file 'at position:' position
  vlc~OpenUri(file)
  call syssleep 1                 -- let vlc time to load the file
  url = vlc~Metadata['xesam:url']
  vlc~Seek(position)
  if file=url then return .true  -- if file loaded successfully
  else return .false

::method Quitvlc
  expose vlc
  vlc~Quit

-- return process id of newest vlc instance of current user; or start an instance
::routine getVlcId
  cmd='pgrep -n -x -u "$USER" vlc | rxqueue'   -- get procid via Rexx queue
  proc=getProc(cmd)            -- get vlc's proc id
  if proc="" then              -- found no vlc instance for this user
  do
     'cvlc --control dbus &'   -- let the shell create a new instance of vlc
     call syssleep 1.5         -- wait to let the new instance request a busname
     proc=getProc(cmd)         -- get vlc's proc id
  end
  return proc                  -- return the proc id

getProc: procedure             -- execute the command, parse its output
  parse arg cmd
  cmd                          -- let the shell execute the passed command
  proc=""
  do while queued()>0
     parse pull proc           -- pull procid from Rexx queue, make sure its emptied
  end
  return proc
```
Script 29: MPRISremoteServer (part 2/2) offering services via DBusServer

This script is started on the remote machine (Notebook) and acts as proxy for vlc. Commands that are sent to this server are forwarded to its local running instance of vlc via its own DBus session bus connection.
On the other machine, vlc mediaplayer has to be started and a file for playback has to be selected. As the goal of this usecase is to continue playing the same file at the same position on the remote machine, the file has to be available on, respectively for both computers.

Forwarding a local url will only work if a file with the same name exists under the same local path on the target computer. For testing purpose it is probably the easiest way to copy a file to the same path on both machines. During the tests, the author made use of a DLNR Server as well, in that case the remote path is available for both computers anyway and no further copying is necessary.
If a file was selected and is playing and the local machine has a Bluetooth adapter connected, the clientscript can be started.
The Bluetooth adapter queries available Bluetooth devices and listens on the system bus for discovery as well as for property changes of a signal issued by `org.bluez`.
The first approach listened for the RSSI value of a device. (Script 29)

```
::method DeviceFound
  expose treshvalue toward away active
  use arg name, signalinfos
  rssi = signalinfos~['RSSI']
  counter = calculateaction(rssi treshvalue toward away)
  toward = counter[1]
  away = counter[2]
  if (toward>4) then do
    if (active) then do
      say "nothing to do at the moment"
    end
    else do
     say "start local player"
      active=.true
    end
   end
 if (away >4) then do
   say "stop local player"
   say "stop remote player"
   active=.false
   end

::routine calculateaction
   parse arg rssi treshvalue toward away
  if (rssi>treshvalue) then do
    toward = toward+1
    away = 0
  end
  else do
    away = away+1
    toward = 0
  end
  return .array~of(toward, away)
```
Script 30: First implementation of DeviceFound

A given threshold for the signal strength was defined in advance. If the signal strength surpasses this value, the main computer does nothing if it is already playing a music file. If the value drops below the threshold, music playback stops and title and playback position of the music file are queried and passed to the other machine.
The remote computer receives this information and continues playback of the file. If the device to track comes back in range of the main computer, the script stops playback on the remote computer and resumes playing on the main computer.
In this very simple setup there is no need to identify a certain Bluetooth device,[73] (it would be

73 Identifying a certain device is possible via the address information within the signal, the script example provided in the next section demonstrates how a certain id can be stored, retrieved and asserted for equality with a current id.

if there are many devices in range) nor is it necessary to implement the approaching and departing detection on both computers.[74]
Testruns demonstrated that the RSSI value is emitted very unevenly, therefore the script was changed to only react upon a certain number of incidents occurred subsequently. Dependent on the device, the signalstrength as well as its frequency fluctuate very heavily. Signals are invoked multiple times a second, or once in a minute. As the signal strength also depends on the quality of both Bluetooth devices, an accurate threshold value has to be guessed and obtained via the trial and error method for each device individually. Script 28 demonstrates the first implementation, only if an event occurs four times subsequently and passes the threshvalue, an action will follow.

As the intention of these code snippets is to provide a comprehensive, working example without much setup effort, a limited version is presented:
Only the signals `DeviceFound` and `DeviceDisappeared` are listened to.
The first part of `MPRISremoteclient` (script 31) sets up the `SignalListener` class and establishes a connection to the media players (local and remote vlc DBus object) and to the Bluetooth device (bluez DBus object). The latter is started to listen for available devices with the instruction: `adapter~StartDiscovery`. With this instruction, the `SignalListener` class is ready to intercept the signals emitted by bluez. (See part 2 script 32)

The second part of the script contains the method `DeviceFound`. As described earlier this method is invoked by the signal very often and is of main interest in this example. In this method, the current status of the local music player is defined with a boolean value to react upon this call or do nothing. If this method is invoked and the music player is currently not playing, information from the remote music player gets queried, it gets stopped and playback resumes on the local machine. This work is done in a method called `switchPlaybackStatus`. This method assesses, if the playback of the file works on the local machine with the statement: `if (vlclocal~Metadata['xesam:url']\=url)` The url of the music file is passed to the media player and if it is not able to process this information (eg.: file is not available) then `['xesam:url']` will not contain the same url, passed to it.[75]
If the method `DeviceDisappeared` is invoked by the signal, the playback on the local machine stops and vlc starts to play on the remote machine.

74 Of course this would enhance the accuracy of the position detection.
75 It would be better to check the availability of the music file prior to passing it over to vlc.

```
signal on halt

  parse arg serveraddress
  if (serveraddress="") then do
  say "Please add server address as argument eg. 192.168.0.24"
  exit -1
  end
  conn = .dbus~system       -- get a system-bus connection
  bluez = conn~getObject('org.bluez','/') -- get the bluez dbus object
  adr = bluez~DefaultAdapter           -- query default bluetooth adapter
  adapter = conn~getObject('org.bluez',adr)  -- get bluetooth adapter object

  conn~listener("add",.rexxSignalListener~new(serveraddress)) -- add listener
  conn~match("add","type='signal',path="adr",interface='org.bluez.Adapter'")
  /* instruct bluetooth adapter to query for bluetooth devices*/
  adapter~StartDiscovery
  say "running till enter is pressed"
  parse pull quit

halt:
  adapter~StopDiscovery
  conn~close
  say "end"
  exit -1

::class RexxSignalListener
::attribute active
::attribute url
::attribute position
::attribute vlcremote
::attribute vlclocal

::method init
  expose active vlclocal vlcremote
  use arg serveraddress
  remoteconn=.dbus~new('tcp:host='serveraddress',port=23000,family=ipv4;unix' -
                       ':path=/tmp/dbus-test;')
  vlcremote=remoteconn~getObject('oorexx.vlc','/oorexx/vlc')
  vlclocal=.dbus~session~getObject('org.mpris.MediaPlayer2.vlc', -
                                   '/org/mpris/MediaPlayer2')
  if (vlclocal~PlaybackStatus\='Playing') then active = .false
  else active = .true -- music plays therefore no action is required
```

Script 31: MPRISremoteClient (part 1/2) listen for Bluetooth devices and connect to MPRISremoteServer.

This setup is very simple and works as expected. Only if a very strong Bluetooth receiver and sender are involved it might happen that `DeviceDisappeared` is never called, even when heading away from the computer a few meters.[76]
For testing purpose it is also possible to turn off the Bluetooth device to let the music play on the remote machine and to start the Bluetooth device again to resume playing on the local machine.

76 The built-in Bluetooth adapter of my desktop PC-mainboard has a range of approximately 50 centimeters, whereas an other Bluetooth adapter reaches a few meters.

```
-- very simple implementation only listens if a device disappears and appears

::method DeviceDisappeared
  expose active
  use arg i, directory
  name = directory['Name']
  say 'device' name 'out of range or turned off'
  self~switchPlaybackStatus(.true)  -- true means switch to remote

::method DeviceFound
  expose active
  use arg i, directory
  name = directory['Name']
  say 'device' name 'in range, signal strengh:' directory['RSSI']
  if \active then self~switchPlaybackStatus(.false)  -- switch to local

::method switchPlaybackStatus
  expose vlclocal vlcremote active
  use arg toremote
  if toremote then do
   active=.false
   url = vlclocal~Metadata['xesam:url']
   position = vlclocal~Position
   say 'playing 'url 'at' position 'on remote computer'
   vlclocal~Stop
   vlcremote~Play(url, position)
   end
  else do
   say 'start local player'
      ar = vlcremote~Stop
      url = ar[1]
      position = ar[2]
      say 'file to play:' url
      vlclocal~OpenUri(url)
      call syssleep 0.5
      if (vlclocal~Metadata['xesam:url']\=url) then do
      say 'Error loading music file! ' -
          'maybe' url 'is not available on this machine?'
      end
      else do
      vlclocal~Seek(position)
      vlclocal~Play
      end
      active=.true
  end

::method unknown
::requires DBUS.CLS           --get access to DBus
```

Script 32: MPRISremoteClient (part 2/2) listen for Bluetooth devices and connect to MPRISremoteServer.

Of course it would be possible to enhance the functionality by only reacting on given devices. This might even be necessary for some setups as multiple Bluetooth devices are sending their signals.

4.2 Script Example: Interaction with Hardware over the System Bus

The next example is about testing services that are reachable over the system bus. In order to provide a script that can somehow be useful and at the same time introduce interesting DBus services. The usecase is defined as following:

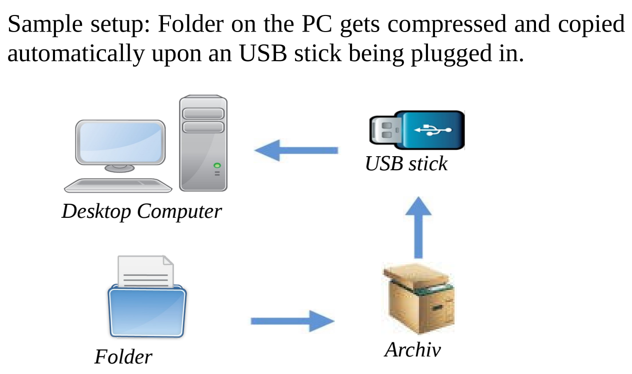

Sample setup: Folder on the PC gets compressed and copied automatically upon an USB stick being plugged in.
Desktop Computer
USB stick
Folder
Archiv

Figure 8: Automated backup to an USB stick

You are working on documents that are worth to be backuped from time to time. It is always nice to have some scripts that are doing this work automatically as making backups of files involves some steps. First of all the backup needs a comprehensive name to be retrievable among different versions that has nonetheless be unique to avoid overwriting existing backups. The next requirement is not to waste storage space senselessly. Therefore it is useful to take advantage of compression. Additionally it is very useful to not have backups distributed to anybody, therefore some kind of authentication mechanism is necessary.

With ooRexx and DBus these requirements are easily meetable. An easy, hence comprehensive approach to create unique filenames is to use a time-stamp and concatenate it to the name. (for example backup_3006142200) The compression can easily be effected via command line instructions. There are many different compression programs available that are well documented.[77] To overcome the third defined requirement, the authentication, an unique id, every USB stick has, can be used to compare it with registered ids. This registration “database” can easily be created with a text-file. If the script is set to registration mode, the id of an inserted stick is queried and stored in this file. From then on, every time an USB stick is inserted, its id is compared with these registered ids. If the ids do not match nothing happens, else the stick is getting mounted and the file in question is copied over.

This setup will make use of `org.freedesktop.UDisks`, accessible on the system bus. Generally for interaction with any kind of storage device, `org.freedesktop.UDisks` provides an interface for interaction with DBus. The interface describes itself as “D-BUS service to access and manipulate storage devices”. Many current Linux operating systems implement `UDisks` version 2, which much to the regret of various programmers is not fully compatible with earlier versions. A version iteration of a somehow low-level program could render existing scripts useless, same happened to the author's originally developed scripts. Much to his luck, the setup process of the scripts to fit the new convention is easily done. In the original setting, the signal `DeviceAdded` of the interface `org.freedesktop.UDisks` was listened to. The new script now connects to the path `org.freedesktop.UDisks2`, listens on the interface

77 http://linuxwiki.de/tar, wiki.ubuntuusers.de/7z, accessed on 2 September 2014.

`org.freedesktop.DBus.ObjectManager` and the signal name is called `InterfacesAdded`. (udisks.freedesktop.org, 2014)

Upon an USB device is being attached to a computer, a lot of actions happens on both buses. This can be monitored with the tool `dbus-monitor`. There are multiple signals informing about the presence of the newly attached device and thus offering instructions for the operating system. On the session bus there are 13 signals from the interface `com.ubuntu.Upstart0_6`, a few signals from `org.kde.KDirNotify` and more than 20 signals from `org.kde.Solid.Device`. This is necessary for the desktop environment to react accordingly. For example the notifications window of the operating system opens and informs that a new device was added and asks the user what to do with media found on the device. As we are interested in only the device, the monitoring tool has to be instructed to change the bus it listens to by invoking it with the switch `-system` (default value is `-session`). The signals spread over the system bus within the interface `org.freedesktop.DBus.ObjectManager` are common for all different desktop environments and therefore are ideally suited to be used by a script that runs on every Linux system with DBus enabled. The emitter of the signal can be tracked down via the following information:

```
sender=:1.17 -> dest=(null destination) serial=4673
path=/org/freedesktop/UDisks2;
interface=org.freedesktop.DBus.ObjectManager; member=InterfacesAdded
```

Signal captured by dbus-monitor -system

In order to obtain all necessary information about interface names and signal names, both DBus debuggers were used as their special abilities are of interest. D-Feet allows to inspect the signature of the Signal easily via its GUI. Following the path identified through the signal shows how the information appears upon insertion of the usb device

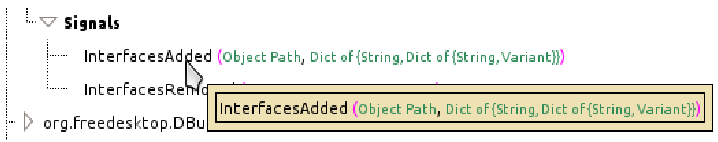

Signals
InterfacesAdded (Object Path, Dict of {String, Dict of {String, Variant}})
InterfacesRemo
org.freedesktop.DBu
InterfacesAdded (Object Path, Dict of {String, Dict of {String, Variant}})

Figure 9: D-Feet UDISK2 method view

Now as the name and the object type of the signal is known, qdbusviewer is used to connect to the signal and lists everything that happens upon an interface is added.

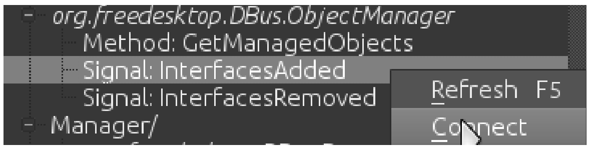

org.freedesktop.DBus.ObjectManager
Method: GetManagedObjects
Signal: InterfacesAdded
Signal: InterfacesRemoved
Manager/
Refresh F5
Connect

Figure 10: qdbusviewer - connect to signal

The DBus signal displayed via qdbusviewer contains interesting information.

```
  Arguments: [ObjectPath:
/org/freedesktop/UDisks2/drives/PNY_ATTACHE_OPTIMA_07840AA00335],
"Serial" = ["07840AA00335"], "Id" = ["PNY-ATTACHE-OPTIMA-07840AA00335"]
```

First signal from UDisks2 (excerpt)

This informs about where the added device is registered within UDisk2 and the label `“Serial”` and `“Id”` are revealing that every USB device can be uniquely identified, in this case the id is a 31-letter long string. The rest of the signal information basically informs about the capabilities of the device (cropped away in this example).

A second signal arrives:

```
  Arguments: [ObjectPath:
/org/freedesktop/UDisks2/block_devices/sdb],
 "Drive" = [ObjectPath:
/org/freedesktop/UDisks2/drives/PNY_ATTACHE_OPTIMA_07840AA00335],
"org.freedesktop.UDisks2.PartitionTable" = ["dos"]
```

Second signal from UDisk2 (excerpt)

UDisks2 registers the device as block device under `/sdb`. The “`Drive”` variable indicates, that it is the attached device. It is also interesting to know that the device has a dos partition table.

A third signal is emitted, which indicates that the device is formatted to FAT32, which is common for a USB stick to be compliant with many different devices like TVs, MusicPlayer and so on. It also indicates that the device in question is registered on the path `org/freedesktop/UDisks2/block_devices/sdb1`.

```
  Arguments: [ObjectPath:
/org/freedesktop/UDisks2/block_devices/sdb1], "Drive" = [ObjectPath:
/org/freedesktop/UDisks2/drives/PNY_ATTACHE_OPTIMA_07840AA00335],"Id
Usage" = "filesystem" "IdType" = ["vfat"], "IdVersion" = ["FAT32"],
org.freedesktop.UDisks2.Partition" = ["Number" = 1, "Type" = "0x0c"]
```

Third signal from UDisks2 (excerpt)

Now as we know how the USB device can be identified and where the device is registered by the operating system, this information can be further processed within the script.

As in this usecase we are only interested to copy data on specific USB devices, its id has to be stored and retrieved somewhere. An easy approach is to use a text file. Writing to and reading from textfiles is a simple task for ooRexx, this can be effected by defining a stream object with `file=.stream~new('usb.devices')` and write to that file by issuing `file~lineout('text to write ...')`. It is also possible to create a stream object on the fly by simply useing `lineout('usb.devices','text to write ...')`. As Rexx uses different pointers for reading and writing, new lines are appended to the end of the file, whereas reading starts at the beginning of the file. This enables the program to store and retrieve multiple devices for backup without needing any reference in which line of the text file the new entry is added and in from which line the program should start looking for entries.

Another information is necessary as well, the file to be copied. In this usecase there is only one file which is worth being backuped. ooRexx offers arguments for streams to define their behavior, valid arguments are WRITE REPLACE or BOTH REPLACE, the latter indicates it is available for read, write and replace. (Ashley, Flatscher, Hessling, & Mcguire, 2009) The command `file=.stream~new("backup.file")~~open("both replace")` makes the stream available and allows to replace existing entries if a new one is written.

So first of all, upon insertion of the USB device, its id is looked up in the textfile. If it is not stored, the user is asked if this device is dedicated to serve as a backup store.

Although a graphical user interface is not necessary for such a simple task, defining the path to a file over the command line is very uncomfortable. There are different approaches to realize a GUI for ooRexx, one possibility is to use ooDialog for example.[78] Another approach is to use BSF4ooRexx. With this package it is very easy to deploy Java classes within ooRexx code, the instructions are mapped to ooRexx such that using Java Classes does not differ much from using regular ooRexx objects. A Filechooser for defining the backupfile can easily be created by importing its Java class and use it like the original Java class. (Flatscher, 2012)

A Filechooser as demonstrated in script 31 (Flatscher, 2009, p9) is especially useful as defining the full object path of a file might result in a long string which would be very unhandy to remember and to enter for a human.

```
fc = .bsf~new('javax.swing.JFileChooser')
fc~setDialogTitle('please select file to backup')
 if fc~showOpenDialog(.nil)=0 then do
  filepath = fc~getSelectedFile~getCanonicalPath
 end
```
Script 33: Filechooser from Java

As there are three signals emitted subsequently on the same interface, the first idea was to use a counter within this method that allows to take further action only if all required information was collected from the signals.

Especially the first signal is interesting as it informs about what device was added (this is important for querying the id) and the last signal, offering the object path to the usb device partition. This approach worked in the first place but for unknown reason, the signal triplet is not guaranteed to be emitted every time. After approximately 30 attempts, the first signal is not emitted anymore. The signature of the signal indicates `(o, a{sa{sv})`.The first approach only used the first part (object path) of the complicated signal signature, establishes a DBusProxy to the service referenced by this path and query its properties for further information.

Due to the fact that the first signal might not be emitted, it is not possible anymore to listen for the first argument of the signal, the object path. The second and the third signal both provide a Drive variable, being the object path of the device. This attribute enables to create a connection and therefore the possibility to query the id of the device. Another attribute that both signals have common is `IdUsage`. The filesystem identifies itself over the attribute `'filesystem'`.

Therefore the following approach was chosen which also demonstrates how to cope with complicated signatures. The first element is the path, this is simply the object path, the next one is a dict. With a do over loop it is possible to “unwrap” the contents. A say command allows to identify the name of the keys (in this example `'IdUsage'` is interesting). With knowledge of this key, the dict can be queried with `dict[index]['IdUsage']`. Script 33 demonstrates this information collection process.

78 ooDialog http://www.oorexx.org/docs/oodialog/oodialog.pdf, accessed on 2 September 2014.

```
::method InterfacesAdded
  expose systempath infoready
  use arg path, dict

  do index over dict
     if (index='org.freedesktop.UDisks2.Block') then do
       if (dict[index]['IdUsage'] = 'filesystem') then do
           drive = dict[index]['Drive']
           spath = path
           infoready = .true
       end
     end
  end
```
Script 34: UDISKS2 Interfaces Added

If sufficient information is collected, (drive for querying the id of the device and spath for being able to mount the filesystem) the parameter infoready is set to .true and the script continues with appropriate action.

Dependent on the settings of the operating system, an USB device is mounted automatically or has to be mounted by the user. If the operating system does not mount the device automatically, this can easily be done with DBus.

With UDisks in its first version, it was necessary to provide information about the filesystem for the mounting procedure. During the assessment of the signals, it was already stated that the device under path `/sdb` has a `dos` partition-table. The device under `/sdb1` states `IdType = vfat`, `IdVersion = FAT32` and under the interface `org.freedesktop.UDisks2.Partition`, `type = "0x0c"`.

Looking up the reference reveals that `0x0c` references a FAT32 file system with a logical block addressing mode[79]

```
0x07 NTFS
0x0B FAT32, used by DOS & Win95
0x0C FAT32 using LBA mode to access to FAT32 partition
0x01 FAT12
0x04 FAT16, less than 32 MB
0x06 FAT16, greater than 32 MB
0x0E FAT16 using LBA mode to access to FAT16 partition
```

With UDisks2, it is not necessary anymore to provide any information for the mounting procedure, but of course it is necessary to call the service with the right object type defined according to the signature as DBus mandates. The service `Mount` on the interface `org.freedesktop.UDisks2.Filesystem` defines a `Dict (String,Variant)` as input parameter and an object path as return parameter. As there is no configuration necessary in this case, an empty dict can be provided. Following command mounts the device and returns its path within the filesystem of the operating system.

`mountpath=udisk~Mount(.directory~new~~put(.array~of(),""))`.

This path is defined automatically unless otherwise instructed within the `/etc/fstab` configuration file. If the device is referenced in this configuration file, the mount command is executed with administrative privileges and all parameters passed via DBus are ignored.[80]

79 Logical Block Addressing: http://www.dewassoc.com/kbase/hard_drives/lba.htm accessed on 2 September 2014.

80 http://udisks.freedesktop.org/docs/2.1.3/gdbus-org.freedesktop.Udisks2.Filesystem.html, accessed on 2 September 2014.

If the device was already mounted by the operating system, an additional Mount instruction raises an error and (dependent on the operating system) invokes a messagebox denoting that this device is already mounted. This is not only annoying but stops the script from execution. If a device was mounted automatically by the operating system, the property MountPoints stores its mountpath. Looking up the signature of this property reveals that the object path is stored as byte array. It seems strange on the first sight that this information is not stored as an objectpath. A full mount path consists of the path where the device is mounted within the file system and its name. It is possible that at least the name of the device is not compliant to the specification of an object path.[81]

The byte array returned from UDisk2 is described in its decimal representation. For example `{47,109,101,100,105,97,47,122,101,114,107,111,112,47,73,67,79,65,0}`. It is obvious on the second view that this array references an object path as the first byte is `'47'` referencing a `'/'` get repeated multiple times.

Translating this array to a string can easily be effected with a function introduced earlier, D2C (decimal to character) [82]

It is absolutely necessary to let the operating system time to automount the device, if queried too early, `udisk~MountPoints` is empty and at least one mount call disturbs the other one, therefore provoking an error. The instruction `call Syssleep 3` solves this issue.

```
mountpath = ""
call syssleep 3              -- wait for automount
mountpoints = udisk~MountPoints
if (mountpoints~size>1) then do
  do i over mountpoints
    mountpath = mountpath || d2c(i)
  end
end
else mountpath=udisk~Mount(.directory~new~~put(.array~of(),""))
```

Script 35: Mount a device over UDISK2

In order to be able to distinguish between different backups of the same file, a timestamp is used. ooRexx provides a function called `DATE()`. There are different possibilities to format a date representation. The default date representation is “day month year” (eg 10 Aug 2014). Using this information as backupfile name is not sufficient, as it might be necessary to create multiple backup files a day. Therefore not only the date, but also the time is necessary for the filename. Passing the parameter `DATE('Full')` returns the number of microseconds passed since 00:00:00.000000 on the 1 January 0001. Although this time-stamp is useful for exactly distinguishing backup versions, even those created within minutes, it is not easy for a human to cope with 17 digits numbers, nor is is easy to understand what date the 17 digits number actually represents. Therefore a granularity between these two date representations was envisaged. The author decided to concatenate the date in the form yyyymmdd and return the value of the function `TIME('S')`, denoting the number of seconds passed since midnight in the form of sssss (both separated with an underscore for easier readability). This results in a file name like 20140710_58098 for example, being distinguishable and yet comprehensive for a human.

81 The author tested this by changing the name of an USB device to `“ÖtestÄ”` on Windows OS. Linux now ignores the name and uses an id like `“0012-D687”`.

82 A test showed, that the return values were already converted to characters, therefore the correct mountpath line is mountpath || i. For demonstration purpose how to convert decimal values, the script was not changed in this description.

The decision for the package program was settled to zip. Zip is an inter-operational file storage and transfer format that is available for nearly every platform and is one of the most widely used compressed file formats since 1989. The ubiquity of zip files is further demonstrated as some common file formats are using zip as default like Javas .JAR, Microsofts .DOCX, PPTX, and Apaches .ODT to name only a few. (PKWARE, 2012). Although zip compression might not be the most efficient, most up to date compression format, it is ensured that .zip files are accessible for all common operating systems (Unix (including Linux), VMS, MSDOS, OS/2, Windows 9x/NT/XP/Vista/7/8, Minix, Atari, Macintosh, Amiga, and Acorn RISC OS).[83]
The command `zip [archive name] [file-to-compress]` compresses the file in an archive. An additional `cp` command line instruction copies the file to the USB device.[84]

Upon the first start of the program, you will get informed that there is no backup file defined and that the program should be started again with the switch -f for file.
If started with this switch, the script will pop up its filechooser and asks to choose the backup file. After this definition, the script will stop again and informs the user that there is no USB device registered so far. The script has to be started again with the switch -r.[85]
If everything is configured correctly, upon insertion of a registered device, the output of the script should look like:

```
###################################################
#### automatic backup program using DBusooRexx ####
###################################################

Backup file: /Bilder/Aquatica.jpg
Ready, ... waiting for registered USB-devices
registered device plugged in, starting the backup
zip -j /media/0012-D687/20140812_64325
/home/zerkop/Bilder/Aquatica.jpg
  adding: Aquatica.jpg (deflated 2%)
```

Example output of AutoBackupDBus.rexx

The following script demonstrate how to code the given example.

```
parse arg switch1 switch2

say '#'~copies(51)
say '#### automatic backup program using DBusooRexx ####'
say '#'~copies(51)
say
registermode = .false

if (switch1 = '-r' | switch2 = '-r') then registermode=.true
if (switch1 = '-f' | switch2 = '-f') then do
  fc = .bsf~new('javax.swing.JFileChooser')
  fc~setDialogTitle('please select file to backup')
  if fc~showOpenDialog(.nil)=0 then do
   call writebackupfile(fc~getSelectedFile~getCanonicalPath)
   end
  else say 'cancelled no (new) backupfile defined'
end
```

83 Documentation of man zip.
84 It is possible to instruct zip to copy the file directly, but if the device is auto-mounted, this attempt often resulted in failures.
85 It is also possible to start the script with both parameters `-r -f` to define the backup file and start the registration mode for the usb device simultaneously.

```
filename = getbackupfilename()
if (filename=.nil) then exit 0
else say 'Backup file:' filename

signal on halt

if (stream('backup.device','c','query exists')\="") then do
  if (\registermode) then do
   say 'Ready, ... waiting for registered USB-devices'
   end
  else say 'Ready, ... waiting for USB-devices to register'
   end
else do
  if (registermode) then say 'Please attach device to register'
  else do
   say 'There is no USB device registered'
   say 'Please start the program again with the switch -r and plug in the device'
   exit -1
  end
end

conn=.dbus~system        -- DBus system connection established
listener=.rexxSignalListener~new(registermode) -- create the signal handler class
-- add the listener to the connection
conn~listener('add',listener) --,interface='org.freedesktop.DBus.ObjectManager')
conn~match('add',"type='signal',interface='org.freedesktop.DBus.ObjectManager'", -
           .true)

parse pull quit          -- wait until enter is pressed

halt:
conn~close
exit -1

::class RexxSignalListener
::attribute drive
::attribute registermode
::attribute infoready
::attribute spath

::method init
  expose infoready registermode
  use arg registermode
  infoready=.false
  stickregistered=.false

::method InterfacesAdded
  expose drive infoready registermode spath
  use arg path, dict
  do index over dict
    if (index='org.freedesktop.UDisks2.Block') then do
      if (dict[index]['IdUsage'] = "filesystem") then do
        drive = dict[index]['Drive']  -- necessary values, name of the drive
        spath = path                  -- path of the filesystem
        infoready = .true             -- ready if enough information collected
      end
        else infoready = .false
     end
  end

  if (infoready) then do
   id=.dbus~system~getObject("org.freedesktop.UDisks2",drive)~Id
   infoready = .false
     if  usbRegistered(id) then do
      if (registermode) then say 'this device is already registered, ' -
               'please start program without switch, or add another device'
      else call copytoUSB(path)
     end
  else do
    if (registermode) then call registerUSB(id)
    else say 'Device added, but not a backup device, ' -
             'start with switch -r to register'
   end
  end

::method unknown       -- necessary to intercept unknown signals
  use arg methName

::routine loadfile     -- loads the file where registered usb devices are stored
  devices =.list~new     -- returns a list of all device IDs
  do  while lines('backup.device')\==0
   devices~append(linein('backup.device'))
  end
  return devices

::routine usbRegistered  -- returns true if parameter id is already registered
  use arg id
  devices=loadfile()
  if (devices~hasItem(id)) then return .true
  else return .false

::routine registerUSB  -- stores the ID of the device in a file
  use arg id
  if (lineout('backup.device',id)==0) then do
   say 'device:' id 'added sucessfully'
   say 'upon next start of the program, this device is used for backup'
  exit -1
  end

::routine copytoUSB  -- the backup file is zipped and copied to the usb device
  use arg path              -- path of the usb device's filesystem
  mountpath=""
  udisk = .dbus~system~getObject('org.freedesktop.UDisks2',path)
  say 'wait a little bit if automounted'
  call syssleep 3
  mountpoints = udisk~MountPoints
  say 'registered device plugged in, starting the backup'
  if (mountpoints~size>1) then do
    do i over mountpoints
     mountpath = mountpath || i
    end
   end                      -- if not mounted automatically
  else mountpath=udisk~Mount(.directory~new~~put(.array~of(),""))
  name = date(s)'_'time(s) || '.zip'
  cmd  = 'zip -j' name getbackupfilename()
  cmd2 = 'cp' name mountpath ||'/'|| name
  say cmd
  say cmd2
  cmd                       -- executes zipping
  cmd2                      -- executes copying

::routine writebackupfile   -- stores the full path of the file to backup
  use arg filename
  file=.stream~new('backup.file')~~open('both replace')
  file~lineout(filename)
  file~close

::routine getbackupfilename -- retrieves the full path of the file to backup
  file=.stream~new('backup.file')
  if file~query("exists")="" then do
    say 'Sorry, you have to define a file for backup first'
    say 'Please start the program again with switch -f'
  return .nil
  end
  name = linein('backup.file')
  file~close
  return name

::requires DBUS.CLS
```
Script 36: Automated backup of files on registered USB devices

4.3 Script Example: Networkmanager

Networkmanager is a toolset designed for easy interaction with different kind of network devices.[86] A connection to this tool can be established over the `org.freedesktop.NetworkManager` DBus object. (Redhat, 2014) The following example describes the steps how to acquire the current IP address of the machine.

This also demonstrates an approach many DBus services are following: An “entry” DBus service returns a path which enables connection to an active DBus object on the same bus name. This “brachiation” around the DBus service object until the object path of the desired service is finally reached is common. In this example this querying process works as following.

After a connection to the `org.freedesktop.NetworkManager` DBus object has been established, the property `ActiveConnections` is queried. The return value is an array of objectpaths. Given that path, connection to the first active connection can be established. In order to know what device serves this connection, the property Devices has to be queried. This return value again is an array of objectpaths. Following this path finally leads to the device that is currently serving the active Internet connection which can then be queried for its ipv4 address. Script example 35 demonstrates this querying process.

The DBus specification of `Networkmanager` informs that the return value of the property `Ip4Address` is an UINT32 value containing the IP address in network byte order.

In order to convert the decimal IP representation to the common dot-decimal notation, there are different possibilities. The first attempt was to first translate the integer value into its hexadecimal representation with `hexaddress=d2x(decaddress)` and then separate the hex-address into four hexadecimal value pairs.[87] These value pairs were then converted back to their decimal value and concatenated in reverse order to provide the IP address in the common endianess. But unfortunately this approach did not always work as expected. For example if my current IP address is 4180922505 in its decimal representation, this value equals the hexadecimal value F933D089 and converted back to decimal F9 equals 249 and 33 equals 51, and so on, thus resulting in 137 208 51 249. But if your IP address contains a 0, this approach might not work properly.
Therefore another approach was used. Rexx offers a handy operator `%` that enables to divide a value and crop its fractional digits. The other required operator is the modulo operation `//`.
Given the decimal IP address, the value that remains after a division with 256 (modulo operation) represents the first part of the common IP address format. The decimal value is then divided through 256, its fractional digits cropped and again a modulo operation is effected. When all digits are calculated, the reverse order represents the current IP address.

86 https://wiki.gnome.org/Projects/NetworkManager
87 It would also be possible to use the ping command and pass the decimal representation, the program will return the common dot-decimal representation.

```
#!/usr/bin/rexx
/*demonstrates how IP address is acquired from org.freedesktop.NetworkManager
  converts decimal IP address to decimal-dot representation */

signal on halt

busname='org.freedesktop.NetworkManager'
conn=.dbus~system
nwmanager = conn~getObject(busname,'/org/freedesktop/NetworkManager')
activeconnectionspaths = nwmanager~ActiveConnections
-- get the path to the first active connection
activeconnection = conn~getObject(busname, activeconnectionspaths[1])
-- get the device path to the first active connection
activedevicespath = activeconnection~Devices
-- get the first active device
activedevice = conn~getObject(busname,activedevicespath[1])

-- Ip4Address returns an UINT32 decimal address
decaddress = activedevice~Ip4Address

numeric digits 20
d = decaddress//256                       -- modulo operation
c = decaddress%256//256                   -- division - no comma values
b = decaddress%256%256//256
a = decaddress%256%256%256//256

ipaddress = d||'.'||c||'.'||b||'.'||a    -- reverse byte order

say 'Your IP address is:' ipaddress

halt:
conn~close

::requires DBUS.CLS         -- get access to DBus
```
Script 37: Networkmanager, handle decimal IP Address

Execution of this script yields in following example output:

```
$ rexx Networkmanager.rexx
Your IP address is: 192.168.0.24
```

5. Outlook and Roundup

The aim of this document was to test the functionality of the newly created ooRexx DBus binding. Therefore testcases were defined with ooTest and sample scripts have been developed. The outcome of this work is twofold. Firstly, the testgroup for the ooTest framework is valuable for programmers of DBus services, additionally every properly programmed test script is valuable for the ooRexx community, as the testgroup can be tested upon every new release of an involved program without much effort. Especially changes on the programming language need not necessarily keep the whole system and most notably the addons in a stable state. The most severe errors are unobtrusive ones, which sometimes influence the result of a program. To overcome these issues, an extensive testgroup was created for assessing most functions provided by the binding, which was carried out multiple thousand times so far. The following output capture of ooTest shows the result of the final testgroup execution.

```
Interpreter:     REXX-ooRexx_4.2.0(MT)_64-bit 6.04 28 Dec 2013
Addressing Mode: 64
ooRexxUnit:      2.0.0_3.2.0    ooTest: 1.0.0_4.0.0

Tests ran:           273
Assertions:          7608
Failures:            0
Errors:              0
Skipped files:       0

Test execution:     00:05:07.553638
```
Script 38: Result of final testgroup execution

It is often difficult to test DBusooRexx in detail as its application, an interplay of the underlying operating system, the IPC mechanism, the programing language and other programs using DBus has to be considered. Analogous to the phrase “too many cooks spoil the broth”, it is of high importance to identify and narrow down any error to its originator and separate it from other systems involved. Of course another party is involved as well, the programmer. Mistakes that I made during exploring DBusooRexx were often difficult do track down, one example was the numeric digits instruction of ooRexx, that in its default setting, restricts integer values to nine digits. (That affects all integer object types except int16 and uint16) If a service does not answer as expected, it was also not that easy to exactly know if the service was defined wrong or the object type of the message call was defined in a false way.

In addition to the testing procedure, the value of a system like DBus was stressed. Using sophisticated IPCs is inevitable for developing high quality services that can be used in different settings and are accessible to any other service. It can also be used to enable concatenation of different services, programmed in different languages. The script example with the media-player demonstrated, that ooRexx programs can easily interact with another application on another computer.

The intention of this work was not only to test the functionality of DBusooRexx, but also to judge about its ease of use. The ooRexx DBus binding offers a new, easy to use approach to get in contact with DBus without having to know much about the DBus at all. For programmers that are new to DBus, this binding might serve as an excellent starting point. This is also due to the fact that ooRexx provides a straightforward language approach that was adapted in DBusooRexx to make any DBus call look like familiar ooRexx code. In comparison to the approach of using the Java language binding and deploy it via BSF4ooRexx, (as described in the introduction) a much easier access has evolved. It might nonetheless be useful to use BSF4ooRexx to establish a connection to DBus, but this time the other way round. Java could use DBusooRexx for establishing connection to DBus. Another impressive functionality was demonstrated as well. A DBus server can be established with only a few lines of code and enables quick and easy connection and interaction. A DBusooRexx script on the server side can create a connection to its own session and system bus and forwards any message call, therefore acting as proxy between the buses of different computers.

During the tests some errors were found which got corrected. The most severe error occurred when listening on the system bus for an USB device as done in the backup script example. Every time a device was added, an error occurred that teared down Java and the Rexx interpreter due to wrong marshalling. Another failure that got corrected in the meantime concerning a threading problem that blocks message calls to DBus if a signal is connected to. This error resulted in scripts that sometimes work, sometimes not, depending on the time a message call is issued. These two errors were revealed during the usage of DBusooRexx with other services. As stated before, bugs like the wrong integer handling for int64 on a 32-bit operating system would probably not been uncovered in the wild as easily as using big integers is not that common for DBus services. The testgroup uncovered the bit-swapping and enabled its elimination. Same is true for double values where the digit limit could not get raised and therefore big double values could not get processed correctly. To my knowledge no service the author ever interacted with on the DBus uses a double value as return object type.

It was valuable that results from this testgroup were already used to correct all stated bugs, therefore this work already contributed a little bit to make DBusooRexx even better. The testgroup and the testserver are probably also useful for other purpose than serving for tests as there are many different approaches demonstrated how DBusooRexx can be used. That includes providing services and making usage of services.

Another thing what was intended to be included in the testgroup, but is missing is an assertion of the performance of DBusooRexx. That is how it actually performs. It would be interesting to compare different DBus language bindings. This also includes testing their ease of use, but especially their performance. A testsetup with bigger amount of data transferred between different services in different languages and assessing their performance sounds interesting and might result in a further work on this topic

Concerning the application of DBus it is very likely that DBus further evolves. Efforts are ongoing to integrate DBus much deeper into the system[88], which might further elevate its relevance and therefore rendering it essential for every programmer to cope with DBus. A project which aims at moving the DBus into the kernel was nearly finished at the time of writing. This new more efficient approach, named kdbus[89] has certain advantages (Corbet, 2014) Its main goal is to make the implementation of DBus more efficient. The k in the word kdbus refers to kernel, which is where the inter-process communication is moved to. Most sophisticated operating systems dispose of a well designed inter-process communication mechanism, for example Windows, MacOS and Android. Whilst Linux uses sockets, FIFOS and shared memory. DBus offers nice functions for handling messages and emit and react to signals, there are also bindings available for the most popular programming languages (now

88 https://code.google.com/p/d-bus/source/browse/kdbus.txt, accessed on 2 September 2014.

89 Unlike the name might indicate, it has nothing to do with KDE

including ooRexx), but DBus also has some drawbacks. One of the biggest is the way data is handled during the transmission as DBus is located in the user space. In its current version, DBus is not the best solution for sharing big amounts of data, the inefficiencies of DBus are slowing down enormously. A better implementation will make it necessary to hang in there and stay up to date with DBus. The usage should not differ much, but the capabilities and essentially the speed will be further enhanced and new features are going to be introduced as well.

References

Ashley, W. D., Flatscher, R. G., Hessling, M., & Mcguire, R., Miesfeld, M., Peedin, L., Wolfers, J. (2009). Open Object Rexx^{TM} Programming Guide. Retrieved August 12, 2014, from http://www.oorexx.org/docs/rexxpg/rexxpg.pdf

Ashley, W. D., Flatscher, R. G., Hessling, M., McGuire, R., Miesfeld, M., Peedin, L., Wolfers, J. (2009). Open Object Rexx Documentation. Retrieved August 12, 2014, from http://www.oorexx.org/docs/rexxref/rexxref.pdf

Corbet, J. (2014). The unveiling of kdbus. Retrieved August 12, 2014, from http://lwn.net/Articles/580194/

Cowlishaw, M. (2003). General Decimal Arithmetic Specification, (March), 1–74. Retrieved August 12, 2014, from http://speleotrove.com/decimal/decarith.pdf

Ennaime, M., Carré, R., Saman, J.-P., Derezynski, M., Welch, N., & Merry, A. (2012). MPRIS D-Bus Interface Specification. Retrieved August 12, 2014, from http://specifications.freedesktop.org/mpris-spec/latest/

Feldmann, S. (2003). An indoor Bluetooth-based positioning system: concept, implementation and experimental evaluation. Retrieved August 12, 2014, from http://projekte.l3s.uni-hannover.de/pub/bscw.cgi/S48c2249c/d27118/An Indoor Bluetooth-based positioning system: concept, Implementation and experimental evaluation.pdf

Flatscher, R. G. (2009). “ The 2009 Edition of BSF4Rexx.”, 1–14. Retrieved August 12, 2014, from http://wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_BSF4Rexx-20091031-article.pdf

Flatscher, R. G. (2011a). An Introduction to the D-Bus Language Binding for ooRexx. *The 2011 International Rexx Symposium*, 1–24. Retrieved August 12, 2014, from http://wi.wu.ac.at/rgf/rexx/orx22/201112-DBus4ooRexx-article.pdf

Flatscher, R. G. (2011b). D-Bus Language Bindings for ooRexx. Retrieved August 12, 2014, from http://rexxla.org/events/2011/presentations/201112-DBus4ooRexx.pdf

Flatscher, R. G. (2012). BSF4ooRexx - The Bean Scripting Framework for ooRexx. Retrieved August 12, 2014, from http://sourceforge.net/projects/bsf4oorexx

Flatscher, R. G. (2013). *Introduction to Rexx and ooRexx - from Rexx to Open Object Rexx (ooRexx)*. Retrieved September 3, 2014, from http://www.facultas.at/flatscher

Holtmann, M. (2006). Playing BlueZ on the D-Bus. *Proceedings of the Linux Symposium*. Retrieved August 12, 2014, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.5475&rep=rep1&type=pdf#page=421

Hossain, A., & Soh, W. (2007). A comprehensive study of bluetooth signal parameters for localization. *Personal, Indoor and Mobile Radio.* Retrieved August 12, 2014, from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4394215

IEEE. (1985). IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*. doi:10.1109/IEEESTD.1985.82928. Retrieved August 12, 2014, from http://www.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

Kahan, W. (1996). IEEE Standard 754 for Binary Floating-Point Arithmetic. Retrieved August 12, 2014, from http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps

Kikawa, M., Yoshikawa, T., Okubo, S., & Takeshita, A. (2010). A Presence-detection Method using RSSI of a Bluetooth Device, *2*(1), 23–31. Retrieved August 12, 2014, from http://lib-repos.fun.ac.jp/dspace/handle/10445/6000

Margiol, S. (2011). *Scripting the Linux D-Bus with ooRexx*. WU Vienna. Retrieved August 12, 2014, from http://wi.wu-wien.ac.at:8002/rgf/diplomarbeiten/Seminararbeiten/2011/201106_Margiol_DBus/ooRexx_Scripting_the_DBus-20110622.pdf

Miesfeld, M. (2009a). The ooTest Framework (Testing the ooRexx Interpreter). Retrieved August 12, 2014, from http://wi.wu-wien.ac.at:8002/rgf/rexx/misc/ooTest/ootest.pdf

Miesfeld, M. (2009b). *Using the ooTest Framework to Write ooRexx Test Cases* (p. 46). Retrieved August 12, 2014,from http://sourceforge.net/projects/oorexx/files/oorexxunit/4.2.0.Snapshot.06/ooTestQuick.pdf/download

Pennington, H., Carlsson, A., Larsson, A., Herzberg, S., McVittie, S., & Zeuthen, D. (2014). D-Bus Specification. Retrieved July 12, 2014, from http://dbus.freedesktop.org/doc/dbus-specification.html#

PKWARE. (2012). .ZIP File Format Specification. Retrieved August 12, 2014, from http://www.pkware.com/documents/casestudies/APPNOTE.TXT

Redhat. (2014). NetworkManager DBUS API. Retrieved August 12, 2014, from http://people.redhat.com/dcbw/NetworkManager/NetworkManager DBUS API.txt

Thompson, W. (2009). *Profiling and Optimizing D-Bus APIs*. Gran Canaria Desktop Summit. Retrieved September 3, 2014, from http://www.willthompson.co.uk/talks/profiling-and-optimizing-d-bus-apis.pdf.

udisks.freedesktop.org. (2014). UDisks Reference Manual. Retrieved August 12, 2014, from http://udisks.freedesktop.org/docs/latest/

Appendix

Troubleshooting

Installation Scripts

The installation package of the ooRexx DBus binding might misbehave under certain circumstances. In its first version, the installation process failed on an Linux system with German language settings due to a parsing issue that does not occur on English versions.

```
copying DBUS Rexx package: cp -pfv dbus.cls /usr/bin
»dbus.cls“ -> »/usr/bin/dbus.cls“
copying DBUS Rexx shared library: cp -pfv libDBusooRexx64.so The NIL
object/libDBusooRexx.so
cp: angegebenes Ziel »object/libDBusooRexx.so“ ist kein Verzeichnis
updating shared library cache: ldconfig -n /usr/lib /usr/lib32
```

If you experience any trouble during the installation process, try to copy the required files manually. As you can see in the above error message, the target directory could not get identified correctly by the script. As the file `dbus.cls` was copied successfully, there is only need to move the file `libDBusooRexx64.so` to its appropriate folder. As I use a 64-bit, Debian-based system, I decided to copy the file to `/usr/lib`.

The installation of ooTest also showed some difficulties. The installation package contains a script called `setTestEnv`. It intends to make all required files for ooTest available by exporting them to the search path. The script is available for both, Windows OS and Linux OS. But as it was accidentally saved in a non-Linux compliant representation by its authors[90], it cannot be executed on an Unix system. This script has to be “repaired” first, which can be effected by issuing the command `dos2unix setTestEnv.sh`[91] which renders the script in an executable state. In my configuration the shell script, although executed successfully, does not work as expected. Therefore I copied the required files manually to a directory within the search path. Copying both, `OOREXXUNIT.CLS` and `ooTest.frm` to the directory `/usr/bin/` made them available everywhere.

DBus Debuggers and DBus Services

If the listed DBus debuggers are not able to either obtain the introspection data correctly in the case of D-feet or not able to invoke all methods in the case of qdbusviewer that does not necessarily mean that your ooRexx services do not work anymore. Try to code a simple “client” application that connects to the implemented service and check whether this works.
The commands `.IDBusPathMaker~publishAllServiceObjects(conn)` or `conn~serviceObject('add','default',.IDBusPathMaker~new(objectPath))` in the case of defining introspection without subclassing DBusServiceObject have to be added. But is is not possible to use both command in the same script as one path would get overwritten by the other.
If your DBus application does not work anymore but did so once, it might be helpful to investigate the service object you want to connect to. If the service was not written by yourself, it might have experienced changes over time and backward compatibility might not

90 The package was created on a Windows OS system which uses other line breaks than Linux.
91 This program is probably not installed per default, it can be obtained by `apt-get install dos2unix` on Debian based systems.

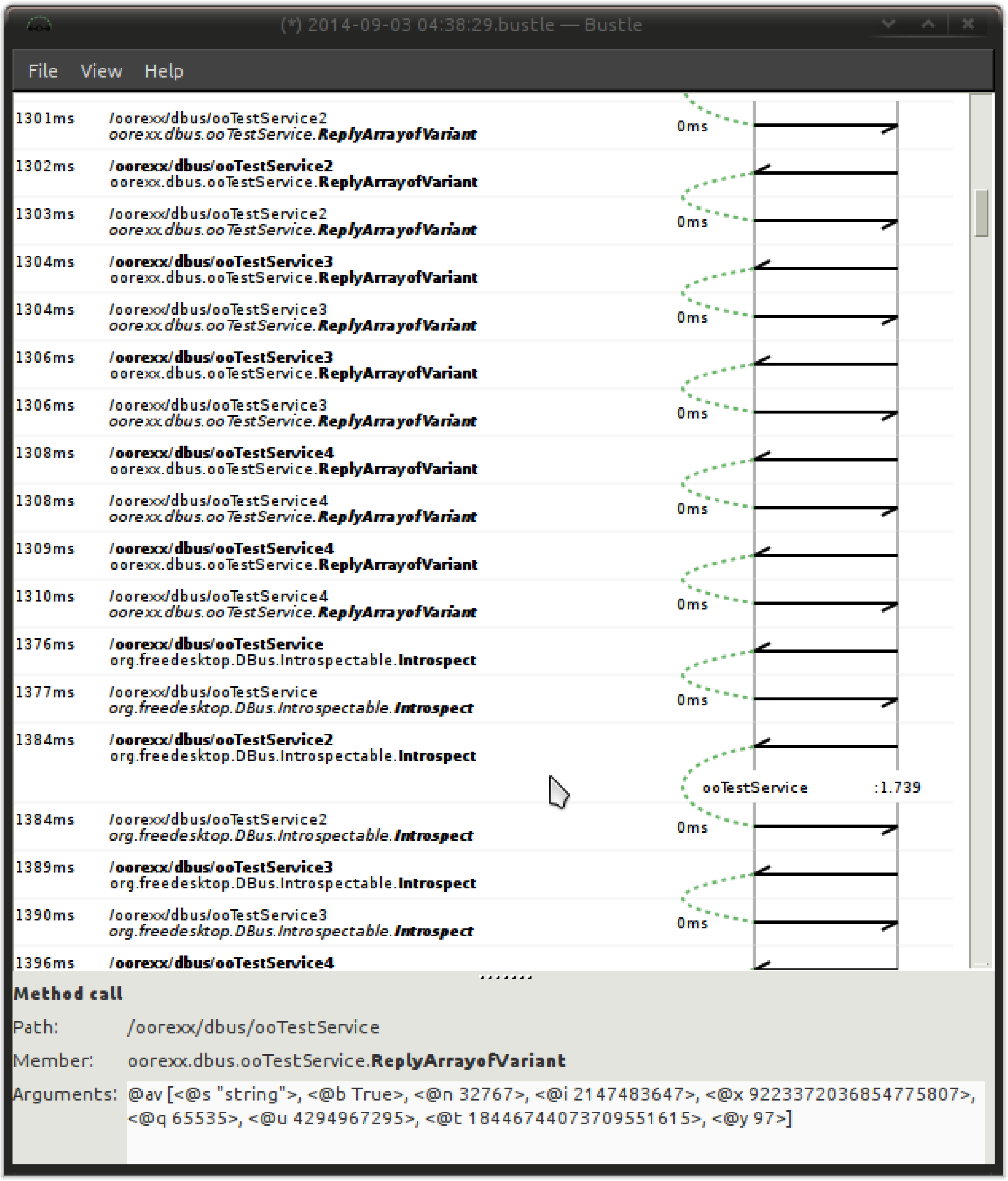
be given anymore. This has already been demonstrated with mpris that now evolved to version 2, rendering old scripts useless. In the case of mpris, most services were still available but the busname was changed and therefore none of the services could be reached anymore. The same can be observed with the `org.freedesktop.UDisk` interface, that also evolves to version 2. Although always making adaptations it is frustrating, carrying backwards compatibility is not a solution on the long sight and introducing improvements of existing DBus services is necessary and desirable as well. Therefore try investigating the services with a DBus debugger of your choice or use the command line command qdbus, followed by the object path of the service that is to be investigated and the interface name. (for example `qdbus org.kde.kwin.Screenshot /Screenshot`)

It was already described that the D-Bus debuggers might have difficulties in displaying all ooRexx Services correctly. This is most probably due to a failure within the introspection.
Try to use the method where the introspection data is created on the fly with command instructions, if the script then does not start up, any of the introspection lines were defined incorrect.
The command line utility dbus-monitor, which is part of the default DBus installation is also helpful as it displays all actions on the DBus. It is possible to monitor the Name acquiring process of the own service and other related information that might be useful.

DBus Profiler

An interesting program that can be used to inspect the activity of DBus services is called Bustle.[92] It is a nice tool that can be configured to listen for all DBus activity and allows to filter specified services. The output file representation of bustle allows to inspect DBus calls, their roundtrip time and carried values. Figure 11 shows an example capture of the running testgroup. In his presentation, the author of Bustle gives a nice overview how this tool can be used for profiling. (Thompson, 2009)
In this paper, Bustle is only used to visually present a testrun of the testgroup, no further inspection of the data was effected. If one of the calls is selected, the transported value get displayed and the programmer sees what value was transported. In this example the call to ReplyArrayofVariant was selected. The arguments are nicely presented. It is also visible what object types are defined with the variant.
The output also nicely demonstrates the alphabetic eradication of the testmethods within the testgroup.

92 Bustle http://www.willthompson.co.uk/bustle/, accessed on 2 September 2014.



```
1301ms   /oorexx/dbus/ooTestService2                                 0ms
         ooTestService.ReplyArrayofVariant
1302ms   /ooRexx/dbus/ooTestService2
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1303ms   /ooRexx/dbus/ooTestService2                                 0ms
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1304ms   /oorexx/dbus/ooTestService3
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1304ms   /oorexx/dbus/ooTestService3                                 0ms
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1306ms   /oorexx/dbus/ooTestService3
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1306ms   /oorexx/dbus/ooTestService3                                 0ms
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1308ms   /oorexx/dbus/ooTestService4
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1308ms   /oorexx/dbus/ooTestService4                                 0ms
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1309ms   /oorexx/dbus/ooTestService4
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1310ms   /oorexx/dbus/ooTestService4                                 0ms
         oorexx.dbus.ooTestService.ReplyArrayofVariant
1376ms   /oorexx/dbus/ooTestService
         org.freedesktop.DBus.Introspectable.Introspect
1377ms   /oorexx/dbus/ooTestService                                  0ms
         org.freedesktop.DBus.Introspectable.Introspect
1384ms   /oorexx/dbus/ooTestService2
         org.freedesktop.DBus.Introspectable.Introspect
                                                       ooTestService       :1.739
1384ms   /oorexx/dbus/ooTestService2                                 0ms
         org.freedesktop.DBus.Introspectable.Introspect
1389ms   /oorexx/dbus/ooTestService3
         org.freedesktop.DBus.Introspectable.Introspect
1390ms   /oorexx/dbus/ooTestService3                                 0ms
         org.freedesktop.DBus.Introspectable.Introspect
1396ms   /oorexx/dbus/ooTestService4
```

```
Method call
Path:       /oorexx/dbus/ooTestService
Member:     oorexx.dbus.ooTestService.ReplyArrayofVariant
Arguments:  @av [<@s "string">, <@b True>, <@n 32767>, <@i 2147483647>, <@x 9223372036854775807>,
            <@q 65535>, <@u 4294967295>, <@t 18446744073709551615>, <@y 97>]
```

Figure 11: Example output of Bustle (excerpt)