ooRexx 5 Yielding Swiss Army Knife Usability

Rony G. Flatscher
Department of Information Systems and Operations
WU (Vienna University of Economics and Business)
Vienna, Austria
rony.flatscher@wu.ac.at

Günter Müller
Institute of Informatics and Society
University of Freiburg
Mannheim, Germany
Fraunhofer SIT, Darmstadt, Germany
mueller@iig.uni-freiburg.de

***Abstract*— The new version 5.0 of the message based object-oriented programming language ooRexx ("open object-oriented REXX") is easy to learn, yet powerful. This article introduces some of the new language features with nutshell examples that at the same time demonstrate its power when deployed in different operating system environments. The modern native API of ooRexx makes it in addition very easy to extend the language with new functionality and deploy it as a macro language for any C++-based application.**

Keywords—dynamic programming language, object-oriented, message based, open-source, multi-platform

I. INTRODUCTION

The programming language ooRexx [1] has gone a long way to become a "swiss army knife (SAK)" tool that can be run off an USB stick.

Forty years ago, in 1979 the programming language REXX [2] the most innovative example for human oriented programming for mainframes and for IBM's SAA strategy got released. Despite this forgone time, REXX is still alive in present day computing and the latest release is a quantum leap as far as easy learning, powerful experimentation and prototyping is concerned. It is easier to learn and to maintain for creating mainframe scripts than the then prevalent, arcane EXEC language. With IBM's SAA (System Application Architecture) strategy REXX was defined as the scripting/batch language for all IBM operating systems, such that the language is present up to today. The programming language REXX, considered to be important was submitted to ANSI and over time an ANSI REXX standard got created and published, which to this day is in effect.

With IBM's desktop operating system OS/2 that got created in cooperation with Microsoft in the 80'ies the REXX language appeared on IBM's extended OS/2 version as a scripting language, following the SAA strategy in place with IBM. Due to the importance of REXX then and the paramount importance of the object-oriented paradigm at that time, IBM created an experimental object-oriented version of REXX in its English Hursley lab which drew many concepts from the object-oriented, message-based Smalltalk programming language. IBM customers communicated their interest but also insisted that the object-oriented Rexx interpreter must be backwardly compatible with REXX such that existing REXX programs would continue to run unchanged.

IBM created a commercial version of "Object REXX" for OS/2 and released it as an option with OS/2 Warp 4 at the end of the 90s, at a time where the desktop war between IBM and Microsoft was about to be lost to Windows. In the years that followed IBM's Windows version of "Object REXX" was sold to its customers who migrated from OS/2 to Windows allowing existing REXX programs to continue to run on Windows, such that the switching costs for the customers could be controlled at a low level.

In the middle of the new century the non-profit special interest group "Rexx Language Association (RexxLA)" received the source code of "Object REXX" from IBM which led to the open-source release under the name of "open object Rexx (ooRexx) 3.0".

As it might be interesting and helpful for assessing the development of a useful tool over time this article will first give a brief historical overview in the section entitled "Brief History of 'ooRexx'". To learn about the features that constitute a "quantum leap" release the section "ooRexx 5 Features" sketch the most important new features, some of which get demonstrated with small nutshell programs to allow the reader to learn about and assess the language and some of its new features.

II. BRIEF HISTORY OF "OOREXX"

ooRexx 3.0 inner workings represented the history of C and C++ at that point in time. The ability for Assembler- or C/C++ programmers to interact with the Rexx interpreter were timid, though it was possible to create "external Rexx function libraries" in those languages, that would enhance the set of built-in functions of the Rexx programming language. This allowed the creation of function packages for interfacing with relational databases or for supporting socket programming when the Internet became prevelant using the "REXX-SAA" interface to Rexx. This interface restricts its data types to strings as the REXX dynamically typed programming language internally processed strings only.

The adaptation by Rick McGuire of the ooRexx kernel to different architectures (32-, 64-bit) and to different operating systems, created a new powerful, native interface to ooRexx comparable to JNI ("Java native interface"). In September 2009 ooRexx 4.0 got released with the new kernel and the new native interface which has been allowing C++ programmers to easily employ ooRexx as a scripting language.

In 2010 BSF4ooRexx [3] ("Bean Scripting Framework for ooRexx") realized a two-way bridge between ooRexx and Java, camouflaging Java as the dynamically typed, case-independent, message-based, object-oriented ooRexx. This allows ooRexx programmers without Java programming skills to take advantage of the Java runtime-environment as all of the Java class libraries and Java objects appear as ooRexx class libraries and ooRexx objects which get messages sent to them.

In the past five years ooRexx has been reworked and enhanced in many respects considerably [4][5], starting out

with switching the build tools from *autotools* to *CMake*, to increasing execution speeds in different areas, such that speed improvements of up to a factor of 20 have been realized!

III. ooRexx 5 Features

ooRexx is a message-based object-oriented programming language. The tilde character (~) serves as the explicit message operator, where the receiver object is left of it and the message name right of it. Should the message carry arguments then they will be comma-separated and enclosed in parentheses immediately following the message name.

Conceptually the receiver will fetch the message with its arguments, if any, and then look up its class for a method by the same name, invoke that method with the supplied arguments, if any, and return the result, if any. If a method cannot be found in the class of the receiver, then its superclass will be inspected and if necessary its superclass up to and including the ooRexx root class named *Object*. In the case that a method by the name of the received message cannot be found, the receiver object will raise an error "*object cannot understand message*" very much like Smalltalk.

A. New Features

Due to the manifold new features, this section aims at noting the major ones and briefly describe them:

- Creating ooRexx from source has become easier by changing the build system from *autotools* to *CMake*, which also eases the creation of external ooRexx function libraries considerably.
- The need to have administrative rights in order to install ooRexx has been lifted. This significant change took place in a manner that allows to create USB-runtimes to carry along, but also to have multiple ooRexx interpreters for different architectures and different ooRexx versions executing at the same time without impeding each other on the same machine.
- Performance work on the kernel achieved a considerably faster (up to a factor of 20) interpreter and among other things improved the multithreading algorithms to increase the stability and speed of scheduling threads and switching between ooRexx threads and threads of third party programs interacting with ooRexx via its native interface.
- Access to the runtime environment of ooRexx programs have been extended by making a "*local*" environment directory available to the package object that gets created for each ooRexx program. Storing objects in this environment directory will speed up looking up environment symbols (ooRexx symbols that start with a dot).

```
arr="one", "two", "three"  -- define an array
do item over arr  -- iterate over items
   say item       -- display item
end
```

Figure 1: New array notation

```
one
two
three
```

Figure 1b: Output of running program in fig.1

- Introduction of a "*package*" scope for methods restricting access from within of an ooRexx package.
- Introduction of the boolean method "*isNil*" in the ooRexx root class "*Object*" which makes it considerably easier and faster to determine whether an object is ".*nil*" (an ooRexx sentinel object representing the semantics of "null" in the language).
- Introduction of name spaces: all ooRexx supplied routines and classes can now be referred to unambiguously with using the prefix "*rexx:*".

B. New Notations

Two notable new notations have been introduced into the language: a new array notation and a new variable reference notation.

a) New Array Notation

Like in many other programming languages arrays play an important role in ooRexx and get used in many places. ooRexx 5 introduces a new notation: a comma separated list of values in ooRexx will be regarded as an expression yielding an array containing the listed values, if not used as an argument list. This notation can be used even in an argument list as an array argument, if the comma separated enumeration of values is enclosed within parentheses. Fig. 1 demonstrates this notation.

b) VariableReference Notation

An interesting new feature added to the ooRexx programming language are variable references. It has become possible to get a variable reference and supply it as an argument to any routine which becomes thereby able to replace the value the variable refers to. The variable reference operator is either the smaller (<) or the greater (>) character which both serve as synonyms for each other. Fig. 2 exploits this new feature by allowing the routine *work* to change the variable *a* in the caller to refer to a different string ("*from the work routine*") upon return.

C. Directives

Directives in ooRexx are placed at the end of a program and are led in with two consecutive colons (*::*). Before an

```
a="hello!" -- refers to string "hello!"
call work >a
say a      -- refers to string "from the work routine"

::routine work
  use arg >tmp -- "tmp" now represents variable "a"
  tmp="from the work routine"
```

Figure 2: Variable reference notation

ooRexx program gets executed by the interpreter, it will first syntax check the program and thereafter carry out all directives, before running the program with the statement in the first line. This way the environment for a program can be set up prior to running it.

A typical usage of directives in ooRexx is related to defining classes with the *::CLASS* directive, their methods with the *::METHOD* and access to their attributes with the *::ATTRIBUTE* directive. As a result the interpreter will create the classes with their methods and attribute getter and setters methods at load time and makes them available well before the ooRexx program gets executed with the first statement at the top of the program.

a) "::REQUIRES" Directive

The "*::REQUIRES*" directive denotes a Rexx program that is required. The interpreter when at the stage of processing the directives will call the Rexx program and upon return makes its public routines and public classes accessible to the program. If such a required Rexx program was already required, then the interpreter will simply make its public routines and public classes available to the Rexx program that requires it.

To support the new namespace feature the ::REQUIRES directive got enhanced allowing optional subkeyword NAMESPACE followed by the namespace string, an example would be: "*::requires pgm.rex namespace xyz*". Any public routine or public class can then be qualified with the prefix "*xyz:*".

b) New Directive "::ANNOTATION"

ooRexx 5 employs the directive concept to allow for annotating ooRexx packages, classes, methods and routines with the new "*::ANNOTATION*" directive supplying an annotation name and its value. All annotations in an ooRexx program can be fetched by sending the *annotations* message to the respective objects at runtime, which will return *StringTable* objects that store the annotation name/value pairs.

c) New Directive "::RESOURCE*"*

The new "*::RESOURCE*" directive allows one to define string data consisting of any number of lines until the needle "*::END*" is encountered. This new feature is meant for allowing to store and retrieve string data at runtime, e.g. help text, (multiline) SQL statements or base64 encoded binary data and much more. The interpreter will store all such string resources in a *StringTable* that can be accessed with the *.resources* environment symbol. Each entry will use the name of the resource as the *index* value and store all its strings as an array for its *item*. Fig. 3 demonstrates this extremely useful new feature with two resources. Fig. 3b depicts its output.

D. Keyword Instructions

ooRexx 5 consists of 30 keyword instructions which are fully documented in "*rexxref.pdf*". The following notable changes have been incorporated:

- "*ADDRESS*" Keyword Instruction

  REXX can be used as a scripting language and as such makes it easy to direct commands to the execution environment using the *ADDRESS* keyword statement. The ANSI-REXX standard defines an optional variant that allows one to redirect the standard files *stdin*, *stdout*, and *stderr* to REXX associative arrays ("stem variables"). ooRexx 5.0 implements for the first time this optional feature and in addition to stem variables allows collection and stream objects to be used in place of *stdin*, *stdout* and *stderr*.

  Fig. 4 demonstrates an ooRexx program that supplies the command "*set | sort*" to the operating system, where the command "*set*" will retrieve the environment variables and pipe them using stdout to the sort filter which reads the data using stdin. The sorted data then are output using stdout. Fig. 4b demonstrates an excerpt of the output to stdout. The ooRexx variable RC ("return code") makes the return code of the command directly available to the ooRexx

```
say "greetings:"
say .resources~greetings
   -- fetch the string array turn it to a string, decode
say .resources~secret~makeString~decodeBase64

::resource greetings -- note the empty lines

  Hello,
  REXX 2019!

::END

::resource secret    -- base64 encoded
b29SZXh4IGlzIGNvb2whIDop
::END
```

Figure 3: The "::resource" directive

```
greetings:

  Hello,
  REXX 2019!

ooRexx is cool! :)
```

Figure 3b: Output of running program in fig. 3

```
"set | sort"    -- command to environment
say "RC="rc     -- display return code
```

Figure 4: Command to environment

```
ACPath=C:\Program Files (x86)\Lenovo\Access Connections\
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Administrator\AppData\Roaming
… cut …
RC=0
```

Figure 4b: Output of running program in fig. 4 under Windows

```
out=.array~new -- create Rexx array
   -- command to environment: stdout gets redirected to array
address system "set | sort" with output using (out)
do i=1 to 3    -- display first three array entries
   say "out["i"]="out[i]
end
say "RC="rc    -- display return code
```

Figure 5: "address ... with" fetching stdout in ooRexx

```
out[1]=ACPath=C:\Program Files (x86)\Lenovo\Access Connections\
out[2]=ALLUSERSPROFILE=C:\ProgramData
out[3]=APPDATA=C:\Users\Administrator\AppData\Roaming
RC=0
```

Figure 5b: Output of running program in fig. 5 under Windows

program upon return from the command. This ooRexx program will run on Windows, Linux and MacOS unchanged.

To ease piping e.g. the stdout data into ooRexx the "ADDR*ESS … WITH*" variant can be employed. Fig. 5 first creates an ooRexx array object to receive the *stdout* data and then issues the above command. Instead of displaying the *stdout* data in the terminal it gets redirected into the supplied ooRexx array and can be immediately accessed in ooRexx. Fig. 5b shows the first two array elements which correspond to the first two lines in fig. 4b.

Fig. 6 Uses the operating system "*sort*" command but supplies the strings in an ooRexx array to *stdin* and fetches the "*sort*" *stdout* data into another ooRexx array for direct processing in ooRexx. Fig. 6b depicts the created output.

This way any operating system command can be supplied by ooRexx and if necessary any of the standard files *stdin*, *stdout* and/or *stderr* can be redirected to ooRexx objects.

- "*DO*" Keyword Instruction

The *DO* keyword instruction allows defining blocks which can be used for creating loops and iterating over collections. ooRexx collection classes have a proper iterator class named "*Supplier*" which allows iterating over all *index*/*item* pairs after creating a snapshot of the collection. To get such an iterator object from any ooRexx collection one merely needs to send the message "*supplier*" to the collection object. The new *DO* variant "***do with index*** *x* ***item*** *y* ***over*** *supplierObject*" eases iterating over supplier objects considerably: each loop will have variable "x" refer the current index value and variable "y" the associated item value.

- "*SELECT*" Keyword Instruction

ooRexx introduces the "*select case expression*" variant as defined in the *NetRexx* programming language created by the original author of REXX, Mike F. Cowlishaw, to allow writing Java programs in an easier to learn programming language that applies the proven "human oriented" principles. This variant of "*SELECT*" allows to evaluate the *expression* at the top of the block and simplifies the denoted "*when*"

```
in ="Tracy","Angie","Berta"            -- input data
out=.array~new                         -- output data
   -- command to environment use ooRexx arrays for stdin and stdout
address system "sort" with input using (in) output using (out)
say "RC="rc       -- display return code
do item over out  -- iterate over all items of array
   say item       -- display items
end
```

Figure 6: "address ... with" supplying stdin and fetching stdout in ooRexx

```
RC=0
Angie
Berta
Tracy
```

Figure 6b: Output of running program in fig. 6

conditions by explicitly listing the values for which they got defined instead of boolean expressions.

- "*USE*" Keyword Instruction

  The *USE* keyword instruction allows to fetch arguments by reference and define default values. ooRexx 5 introduces a new "*USE LOCAL*" keyword instruction variant which can only be used in method routines as it reverts the semantics of local variables to be regarded as attributes (object variables) instead. "*USE LOCAL*" allows a blank delimited list of variable names to be listed, which should be regarded as local variables. This new feature is very helpful for classes that possess many attributes (object variables) and to which method routines of the class need direct access. It is the inverse of the "*EXPOSE*" keyword instruction of method routines which usually is used to denote a blank delimited list of attributes that get directly accessed by the method routine.

E. New ooRexx Classes

The REXX and ooRexx programming languages follow the "keep the language small" philosophy. Adding new built-in functions or ooRexx classes are therefore rare and warrant learning about them and the motivation to add them to the language.

- New Utility Class "*VariableReference*"

  This class represents a variable reference and maintains the name of the referenced variable and the value that variable refers to.

- New Utility Class "*AlarmNotification*"

  ooRexx is a multi threaded programming language. The ooRexx "*Alarm*" class allows one to send an ooRexx message later, after a certain time has elapsed or a certain date and time has arrived. The "*AlarmNotification*" mixin class defines the abstract method "*triggered*" which allows one to learn when the *alarm* object sent off the message asynchroneously.

- New Utility Class "MessageNotification"

  ooRexx is a multi threaded programming language. The fundamental ooRexx class "*Message*" (messages are first order objects) can be used to create *message* objects. In the case a message gets executed asynchroneously sending it the "*start*" (instead of the "*send*") message, one can request to be notified when the asynchroneous message completed. The "*notify*" method in the *Message* class expects the object that wants to get notified about completion to implement the abstract "*messageCompleted*" method of the "*MessageNotification*" mixin class.

- New Utility Class "*EventSemaphore*"

  ooRexx is a multi threaded programming language. This ooRexx class allows multiple ooRexx threads to synchronize on it. Once the event semaphore gets posted, all blocked threads are able to run again.

- New Utility Class "*MutexSemaphore*"

  ooRexx is a multi threaded programming language. This ooRexx class allows multiple ooRexx threads to use a shared resource, guaranteeing that only one of the threads can execute at a single point in time while all other threads keep waiting on the semaphore block. Once the thread holding the lock ends one of the waiting threads gets the control and is able to execute.

- New Collection Class "*StringTable*"

  ooRexx like many programming languages supplies a directory like collection class, which in ooRexx is named "*Directory*". One interesting feature of it exploits the ooRexx *unknown* mechanism which allows one to intercept the error condition when a method cannot be found. In this case the ooRexx runtime environment will search for a method with the name *unknown* and if found, invokes it instead supplying the unknown message and arguments sent with it. This way a class can react specifically in the case of (expected) unknown messages. The directory class exploits this feature by creating or fetching entries dynamically. In the case of fetching an entry by the name of the unknown message it needs to check for methods serving as entries and execute them, returning whatever it returns. This incurs some overhead and is rarely used in ooRexx programs.

  To improve the runtime performance the new ooRexx class "*StringTable*" got introduced which forgoes the ability to lookup and execute method objects. As the ooRexx 5 interpreter employs this class internally, some performance gains can be attributed to this class.

IV. Roundup and Outlook

Since 2014 work on the open-source ooRexx 5.0 has aimed at improving functionality but still keeping it "human oriented" in its syntax and feature sets. In the process considerable execution speed improvements have taken place in all areas of the interpreter.

With the new ability to use ooRexx without administrative rights and allow different versions of ooRexx to stably run concurrently on a computer, one has become able to create "USB-versions" of ooRexx. This makes it possible to carry along one owns ooRexx interpreters for different architectures and operating systems together with matching ooRexx function packages like BSF4ooRexx (and Java) as well as ooRexx utilities on an USB stick. This way the easy to learn programming language ooRexx can be used as a SAK-tool ("swiss-army-knife" tool) from Windows, Linux, MacOS to even IBM mainframe "Linux on IBM Z".

References

[1] R. G. Flatscher. *Introduction to Rexx and ooRexx*. Vienna: facultas, 2013.

[2] M. F. Cowlishaw. *The REXX Language*. (2nd edition). Englewood Cliffs, New Jersey: Prentice Hall, 1990.

[3] R. G. Flatscher. "The 2010 Edition of BSF4ooRexx". 2010 International Rexx Symposium, Almere, Netherlands, 2010.

[4] R. G. Flatscher. "ooRexx 5.00 New Features". 2017 International Rexx Symposium, Amsterdam, Netherlands, 2017.

[5] R. G. Flatscher. "Menschenfreund". *'ix*, vol. 30, pp. 66-70, Nov. 2017.