Resurrecting REXX, Introducing Object Rexx

Rony G. Flatscher (Rony.Flatscher@wu-wien.ac.at), Wirtschaftsuniversität Wien

Abstract: *This paper introduces and characterizes the dynamic languages REXX and Object Rexx in such a way that many of their notable concepts get described and demonstrated with short nutshell examples. This way these concepts are made available for reflection and can be discussed in depth at the workshop.*

1 Resurrecting REXX

This section introduces the programming language REXX' history and briefly its main concepts.

1.1 A Brief History of REXX

In 1979 IBM introduced the "**Re**structured E**x**tended E**x**ecutor" (REXX) language interpreter, which was deliberately created by the English IBM employee Mike F. Cowlishaw as a "human centric language" to replace eventually the then already awkward EXEC II batch language for IBM mainframes. As such this typeless language to this very day plays an incredible important role in controlling the operation and maintenance of IBM mainframes. Over the course of time REXX became the standard scripting language of operating systems like the Amiga OS or later OS/2, which originally was planned by IBM and Microsoft to replace DOS and the 16-Bit Windows system.

In the 80'ies IBM implemented the "System Application Architecture" (SAA) in its company, with the aim to standardize the creation of its software systems. E.g. The SAA "Common User Access" (CUA) guidelines defined the function key 1 (F1) to be assigned the semantics of being the help key.[1] 1987 IBM defined REXX to be the strategic procedural (scripting) language for all of IBM's operating systems.

In the 80'ies the popularity of REXX in medium-sized and large companies induced the creation of commercial versions of REXX, that allowed them a good living. In the same timeframe free and opensource REXX interpreters like Regina or imc/Rexx were

[1] Microsoft followed the SAA CUA guidelines for its Windows (graphical) user interface. An updated version of IBM's SAA CUA later defined how users of object-oriented systems should interact with objects on the user interface, which got implemented in the OS/2 "Workplace Shell" (WPS) user interface at the end of the 80'ies. The WPS got implemented with IBM's object-oriented framework "System Object Model" (SOM), so each object on the WPS was truly implemented as a WPS object! Much of IBM's input to the Object Management Group's (OMG) Common Request Broker Architecture (CORBA) standard stems from IBM's distributed SOM (DSOM) design and implementation experiences.

created and made available to the public. After a few years of work, which gathered many of the REXX interpreter authors, the REXX language got standardized by the American National Standards Institute (ANSI), the 'ANSI "Programming Language - REXX", X3.274-1996'.

The decimal arithmetics[2] of REXX as defined in the ANSI REXX standard was used in the 2000'ies to create compliant extensions to programming languages like Java or Eiffel, with plans to create commercial processors which would implement decimal arithmetics.

1.2 A Very Brief Introduction to REXX

The REXX language is case-insensitive[3] and possesses only a string datatype, where the length of the string is only confined by the addressable memory available to the process. Variables need not be defined, nor do they have to have an explicit value assigned to it[4]. If the string contains a sequence of characters that can be interpreted by humans as a valid decimal number, then it is possible to carry out decimal arithmetics with it, using a default of nine digits in the operation.

```
say "Hello world!"    /* yields: Hello world!                              */
say 1/3               /* yields: 0.333333333                               */

numeric digits 25     /* now use 25 significant digits in arithmetics      */
say "1"/3             /* yields: 0.3333333333333333333333333               */

"rm -rf *"            /* remove all files recursively                      */
```

Code 1: A Simple, Yet Interesting REXX Program

Literal string values representing a decimal number need not to be enquoted (REXX would do that automatically). Each REXX statements ends with a semi-colon (;), and if it is missing then REXX implies a semi-colon at the end of the line.

Should there be a REXX statement that is unknown to REXX (as the last statement in Code 1 above), then REXX will pass the statement's string to the program that had started REXX, which usually is the command shell. In the case of a Unix shell or a Windows system with the Cygwin-utils installed, the last statement in the REXX

[2] Before REXX became an official IBM product that customers could order and get support for, a beta version was deployed among selected customers, one being the Stanford Linear Accelerator Center (SLAC). At SLAC the language design itself was perceived positively, especially a particular feature which served them well in their calculational needs: REXX has been able to carry out decimal arithmetics with an arbitrary amount of digits. It has been noted that for these particular features SLAC has approached IBM and persuaded it to release REXX as a commercial product. So its first application was a mathematical one.

[3] Actually, a REXX interpreter will uppercase all symbols, which are not enclosed in double or single quotes. Multiple blanks between symbols will be reduced to one.

[4] If using a variable without an explicit assigned value, then REXX will use the (uppercased) name of the variable as the (string) value instead.

program will cause the deletion of all files and directories from the current directory! In the IBM mainframe world this standard behaviour of REXX has been used e.g. for creating macros for programming editors in REXX where the REXX coders just pass the editor's commands verbatimely to the editor that invoked the REXX editor macro.

In addition in REXX it is possible to define procedures and functions, invoke other Rexx programs as procedures/functions and invoke "external REXX functions" that are implemented in assembler, C, C++ or the like, allowing to extend its functionality quite considerably.

As there are no explicit datatypes (other than string) in REXX there is also no array (or any other collection-like) datatype. Still, by introducing a particular convention in naming variables, it is possible to get at the effects of associative arrays: if a variable name contains a dot[5], everything after the dot can be regarded to be a string "index" and the sequence of characters up to and including the first dot is called a `stem`. The character sequence after the first dot is named `tail`.

The strings delimited by dots in a variable name are regarded as variables by REXX, hence a program like the one in Code 2 can be used to represent an array type where the "0-element" indicates how many entries are available, that start with an index starting with the decimal number "1".

```
file.1='max.txt'     /* variable "FILE.1" is assigned a value        */
file.2="pia.txt"     /* variable "FILE.2" is assigned a value        */
file.0=2             /* variable "FILE.0" is assigned a value        */

do i=1 to file.0     /* will loop twice with control variable "I"    */
   say file.i        /* "I" will be substituted with "1" and "2"     */
end
/* yields the following output:
   max.txt
   pia.txt
*/
```

Code 2: Using a Stem Variable as an Integer-indexed Array.

To use decimal numbers as integers indices is just a convention to mimickry an integer array. In effect any string value can be used as a value after a dot, as can be seen from `Code 3`.

```
Austria.Tirol=500000             /* population of Tirol              */
austria.tirol.innsbruck=120000   /* population of the city Innsbruck */

a="TIROL.INNSBRUCK"              /* define the tail value            */
say Austria.a                    /* displays: 120000                 */
```

[5] In REXX a symbol may be constructed by alphanumerical characters, the exclamation mark, the question mark, the underline character and the dot. A REXX variable is a symbol that must start with a letter, exclamation mark, question mark or underline character, and may be followed by all characters that are valid for a REXX symbol.

```
a="TIROL"; b='INNSBRUCK';        /* define "index" values           */
say austria.a.b                  /* displays: 120000                */

say AusTriA.TiRoL.InnsBruck      /* displays: 120000                */
```

Code 3: Using a Stem Variable as an Associative Array.

2 Introducing Object Rexx

Object REXX[6] was originally created by IBM. It possesses an object model which is strongly influenced by Smalltalk (including metaclasses) and also by successive OO-developments (e.g. multiple inheritance). Object REXX, like its predecessor REXX is implemented as an interpreter.

2.1 A Brief History of Object Rexx

Large companies that have had REXX in operation because of using IBM mainframes for their business administration needs started to embrace the IBM OS/2 operating system for their Personal Computer (PC) needs at the end of the 80'ies. IBM's "extended edition" of OS/2, which contained IBM communication software, a relational database and more, also included REXX as the SAA procedural language. In that timeframe a huge object-oriented wave in the market place caused IBM to deploy [D]SOM and even a fully object-oriented user interface (Workplace Shell, WPS) which itself was implemented with the SOM framework.

In 1988 a group of English IBM engineers started work on creating an object-oriented version of REXX. Preliminary results were presented at different IBM user groups for discussion, among them the influential special interest group (SIG) named SHARE, which organizes the large IBM customers. The SHARE SIG was in favor of a true object-oriented version of REXX, but strongly requested that it should be backwardly compatible with REXX, such that no developed REXX programs in those large companies would need to be rewritten. It took almost nine years from the inception of the first project team, a couple of experimental designs and implementations, until a commercial version of what became known as "Object REXX" was made available as a product as part of "OS/2 Warp" in 1997[7].

Being a fully object-oriented implementation Object REXX internally works object-oriented and follows the notion of Smalltalk, that "everything is an object".

[6] Here "REXX" in capital letters refers to the IBM version(s), "Rexx" in mixed case to versions that are from different vendors or that are available in opensource form.

[7] Originally IBM's work on an object-oriented version of REXX was conducted in England under the lead of Simon Nash, then the project was transferred to the United States where finally a design and implementation under the lead of Rick McGuire succeeded.

Incarnating on the OS/2 operating system that was loaded with object-oriented infrastructures and had the [D]SOM framework available, this initial OS/2 version of Object REXX was fully intergrated with [D]SOM. Therefore it became possible to send Object REXX messages to WPS objects, like folders, files, fonts, devices, etc. It was even possible to create an Object REXX class that specialized a WPS folder (implemented in C++) that for instance would add password protection in a few lines of code (approximately 50 lines of code!). Unfortunately, Object REXX shared the fate of OS/2, which started to fade when IBM lost its "deskop battle" to Microsoft, after having ended the year long co-operation at the beginning of the 90'ies.

Independent of the OS/2 implementation, IBM created a commercial version of Objet REXX for AIX and Windows. Especially the Windows version generated notable income for IBM's PC software segment as all the large customers that had to migrate from OS/2 to Windows needed a Windows based REXX interpreter in order to take the numerous REXX programs into the new environment, being able to run them unchanged. At the end of the beginning of the new millenium experimental ports to Linux and Solaris were created at IBM.

Finally, in 2004 the non-profit-oriented, international SIG "Rexx Language Association" `[RexxLA]` and IBM entered into talks about receiving the IBM source code of Object REXX and making it available under RexxLA's responsibility as opensource software, maintain and develop it further. The introduction of "Open Object Rexx" `[ooRexx]` took place at RexxLA's 2005 International Rexx Symposium which took place in Los Angeles. Among the core opensource development team one can find the original father of Object REXX, Rick McGuire, who has taken on the (technical) lead role again in furthering the development of ooRexx.

2.2 A Very Brief Introduction to Object Rexx

Object REXX, a.k.a. ooRexx was created after the IBM researchers studied many object-oriented programming languages, most notably Smalltak and to a lesser extent C++ (in which the ooRexx interpreter itself is implemented).

ooRexx allows defining (meta-)classes, using reflection, creating one-off objects, mandating the use of explicit message operators for sending messages to objects, that look for methods by the name of the received message, as well as creating "floating"[8] methods and employing a runtime environment that is realized as a stack

[8] "Floating" methods are not associated with a specific class and create an own scope for sharing attributes ("object variables"). They can be easily created by defining them before the first class gets defined in a program. The environment symbol `.methods` is used to get access to such methods.

of at least four[9] directory objects being looked up on behalf of ooRexx programs, as well as being able to execute objects in a multithreaded manner. ooRexx comes with a relatively small (but quite well thought of) set of classes organized in a very flat classification tree. Table 1 depicts all classes that are installed with ooRexx.

Class Name			Comment
`Object`			Root class, fundamental
	`Alarm`		Send message asynchroneously
	`Array`		Collection class, no predefinition of size or dimension necessary, ordered
	`Class`		Metaclass, fundamental
	`Directory`		Collection class, index is a string, one object per index, no order implied
	`List`		Collection class, ordered
	`Message`		Fundamental class
	`Method`		Fundamental class
	`Monitor`		Monitors messages sent to objects
	`MutableBuffer`		Comparable to Java's `StringBuffer`
	`Queue`		Collection class, ordered
	`Relation`		Collection class, index is any object, multiple objects per index possible, no order implied
		`Bag`	Index and associated object are the same object
	`Stem`		Represents "classic Rexx" stems
	`Stream`		Stream (e.g. file) input/output
	`String`		Object's string values are not mutable
	`Supplier`		Iterator for collection classes
	`Table`		Collection class, index is any object, one object per index, no order implied
		`Set`	Index and associated object are the same object

Table 1: Object Rexx Classification Tree (Indented Classes Are Subclasses).

The program depicted in Code 4 contains two Rexx statements, one using the Rexx built-in *function* "reverse", and one that employs a *message* with the name "reverse" instead.

```
say reverse("aloha")     /* the reverse function returns: ahola    */
say "aloha"~reverse      /* the reverse message returns:  ahola    */
```

Code 4: Using a Message to Invoke a Method with the help of the Receiving Object.

[9] For each program an anonymous directory object is created that maintains important definitions for the runtime like public classes or routines, for each session a named directory object (`.local`) is created that contains e.g. the stream objects representing the standard files `stdin`, `stdout` and `stderr`, in addition a named directory object (`.environment`) that contains all of the ooRexx classes, and an anonymous directory object for the runtime environment to store additional runtime information.

As can be seen from Code 4 ooRexx mandates the use of an explicit message operator, the tilde character: `~`.[10] Left of the message operator is the object to which the message is sent to, right of it is the name of the message to be sent. If the message carries arguments, then the message name is immediately followed by round parenthesis in which the (list of) argument(s) is given.

Conceptually, the object receives the message and looks for a method by the same name, starting out in the class from which the object got instantiated, searching through all superclasses until the root class `Object` is reached. The first found method will be invoked by the object that received the message (returning the result, if one was created by the method). If a method by the same name as the received message cannot be found by the object, then the program is stopped with the (Smalltalk-like) error message: *"object cannot understand message"*.[11]

Both statements in `Code 4` yield the same result. As a matter of fact, the ooRexx interpreter will internally reformulate the first ("classic Rexx") statement to its object-oriented form as shown in the second statement, so both statements actually invoke the same ooRexx string method named "reverse", returning a new string object that has the characters in reverse order to the receiving string object.

```
.Dog   ~new("Sweety") ~bark   /* create a dog, let it bark                */
.BigDog~new("Grobian")~bark   /* create a big dog, let it bark            */

::class  Dog
::method init  /* constructor method */
  expose name  /* establish direct access to attribute (object variable)   */
  use arg name /* retrieve argument, assign it to attribute                */
::method name attribute       /* define set and get attribute methods     */
::method bark
  say self~name":" "Wuff Wuff"

::class  BigDog subclass Dog
::method bark
  say self~Name":" "WUFFF! WUFFF!! WUFFF!!!"

/* yields the following output:
   Sweety: Wuff Wuff
   Grobian: WUFFF! WUFFF!! WUFFF!!!
*/
```

Code 5: Creating and Instantiating Different Kind of Dogs.

The Object Rexx program in `Code 5` uses directives (led in by two consecutive colons: `::`) to define classes and its methods. Directives instruct the ooRexx interpreter to carry them out, before the program starts executing at line number one. This way it

[10] In ooRexx the message operator is called "`twiddle`".

[11] ooRexx has the following `UNKNOWN` mechanism built in: if an object cannot find a method by the name of the received message, but it found a method `UNKNOWN` while searching through the classes, then it will invoke that method, which will receive as its first argument the name of the unknown message and an array object as its second argument containing the arguments supplied (sent) with the message, if any.

is assured that before a program starts, all needed resources are made available by the interpreter. In this case the directives command the interpreter to create class and method objects, and then assign the method objects to the class objects. In ooRexx a method named `init` serves as the constructor[12] method that gets invoked at creation time; whatever arguments one passes to the `new` method gets passed on to the constructor.

The `BigDog` class specializes the class `Dog` and re-implements (overrides) the method `bark`, as clearly :-) big dogs bark differently from normal dogs. The program will create first an instance of a normal `dog` supplying the name which gets processed in the constructor method `init` and then is sent the message `bark`, then repeating this sequence of messages with the `BigDog` class.

Object Rexx allows for multiple inheritance as well, as the probably self-explaining program in `Code 6` illustrates.

```
/* Multiple Inheritance */
.RoadVehicle     ~new("Truck")  ~drive
.WaterVehicle    ~new("Boat")   ~swim
.AmphibianVehicle~new("SwimCar")~show_off

::CLASS  Vehicle        /* define the vehicle base class                    */
::METHOD name ATTRIBUTE /* let interpreter define a getter and setter method  */
::METHOD init           /* define constructor method                        */
  self~name=ARG(1)      /* use the setter method to set the attribute's value */

::CLASS  RoadVehicle     MIXINCLASS Vehicle
::METHOD drive          /* define a road vehicle method                     */
  SAY self~name": 'I drive now...'" /* use the attribute getter method       */

::CLASS  WaterVehicle    MIXINCLASS Vehicle
::METHOD swim           /* define a water vehicle method                    */
  SAY self~name": 'I swim now...'"  /* use the attribute getter method       */

::CLASS AmphibianVehicle SUBCLASS RoadVehicle INHERIT WaterVehicle
::METHOD show_off       /* demonstrate multiple (implementation) inheritance  */
  self ~~drive ~~swim   /* using cascading messages (two twiddles)           */

/* yields the following output:
  Truck: 'I drive now...'
  Boat: 'I swim now...'
  SwimCar: 'I drive now...'
  SwimCar: 'I swim now...'
*/
```

Code 6: Multiple (Implementation!) Inheritance.

The class `Vehicle` serves as the base class, its subclasses `RoadVehicle` and `WaterVehicle` serve as mixinclasses. The `AmphibianVehicle` specializes at the same time `RoadVehicle` and `WaterVehicle` (in exactly that order which will be used for method resolution). The method `show_off` demonstrates the usage of cascading

[12] ooRexx also possesses a destructor method which is simply named `uninit`.

messages represented by two consecutive twiddles, indicating that both messages are directed at the same object, referred to by `self`, which references the object that conceptually invoked the method.

Due to place constraints the last three examples only demonstrate how a dynamic language like ooRexx can address totally different environments as if they were actually ooRexx environments, ie. ooRexx objects to which one can send (untyped!) ooRexx messages: a Windows COM/OLE/ActiveX component (the Internet Explorer) in `Code 7` and a Java environment (the program runs unchanged on Linux and Windows!) in `Code 8`.[13]

```
call orexxole.cls       /* get the COM/OLE/ActiveX support              */
   /* create an instance (a proxy) of the InternetExplorer             */
myIE = .OLEObject~New("InternetExplorer.Application")

myIE~Width = 1024       /* set the width in pixels                      */
myIE~Height = 768       /* set the height in pixels                     */
myIE~Visible = .True    /* now show the window                          */
myIE~Navigate("http://www.ooRexx.org")

say "sleeping 15 seconds..."
Call SysSleep 15
myIE~quit               /* now close the Internet Explorer              */
```

Code 7: Remote-controlling the Microsoft InternetExplorer on Windows.

```
call bsf.cls            /* get the Java support                         */
s=.bsf~bsf.import("java.lang.System")  /* import the Java System class  */
say "java.version:" s~getProperty('java.version')

/* yields the following output (maybe):
   java.version: 1.5.0_06
*/
```

Code 8: Using Java Objects as If they Were ooRexx Objects.

A last, maybe impressive example as it uses Sun's StarOffice/OpenOffice to remote-control its word processor component (mostly implemented with C++ components) from ooRexx. `Code 9` can be run unchanged on Linux and Windows.

```
call uno.cls   /* get the UNO (StarOffice/OpenOffice, OOo) support      */
oDesktop=UNO.createDesktop()     /* get OOo desktop service object      */
xComponentLoader=oDesktop~XDesktop~XComponentLoader
               /* open the blank *.sxw - file                           */
url = "private:factory/swriter" /* define the document URL               */
               /* the following statement spans two lines               */
xWriterComponent=xComponentLoader~loadComponentFromURL(url,"_blank", 0,  -
                                                       .UNO~noProps     )
xText=xWriterComponent~XTextDocument~getText -- get the OOo text object
xText~setString("Hello World, this is ooRexx speaking!") /* insert text */
```

Code 9: Remote-controlling StarOffice/OpenOffice in a Platform Independent Manner.[14]

[13] In all of these cases the ooRexx `UNKNOWN` mechanism is put to work to make it relatively easy to address alien/foreign environment from ooRexx in a very easy manner.

[14] If a statement needs to be split into more than one line, one can use a trailing dash or comma character to indicate to Rexx that the statement will be continued.

3 Summary

This paper attempted, according to the workshop's theme, to introduce two dynamic languages "for revival" that offer very interesting concepts and an interesting power to many real-world applications: Rexx and its object-oriented successor Object Rexx a.k.a. ooRexx. Although not widely known (anymore in the case of Rexx, not at all in the case of ooRexx), both languages - due to their implemented concepts - allow solving many very different use cases and varying problems efficiently.

One application area of such dynamic languages is clearly the scripting, remote-controlling of operating systems and applications that are or may be implemented in statically typed languages. The potential of the dynamic languages and possibilities of innovative applications are very high. It is likely that other dynamic languages could take advantage of some of the proved and innovative Rexx and ooRexx concepts.

4 References

[Cow90] Cowlishaw, M.F.: "The REXX Language", Prentice-Hall (Second edition), 1990.

[Flat04] Flatscher R.G.: "Camouflaging Java as Object REXX", in: Proceedings of the „2004 International Rexx Symposium“, Böblingen, Germany, May 3rd - May 6th, 2004.

[Flat05] Flatscher R.G.: "Automating OpenOffice with ooRexx: ooRexx Nutshell Examples for Write and Calc", in: Proceedings of the „The 2005 International Rexx Symposium“, Los Angeles, California, U.S.A., April 17th - April 21st, 2005.

[Fos05] Fosdick H.: "Rexx Programmer’s Reference", John Wiley & Sons, ISBN: 0-7645-7996-7, URL (as of 2006-04-01): `http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764579967.html`

[cetRexx] URL (as of 2006-04-01): `http://www.cetus-links.org/oo_rexx.html`

[ooRexx] URL (as of 2006-04-01): `http://www.ooRexx.org`

[Rexx] URL (as of 2006-04-01): `http://www.Rexx.org`

[RexxInfo] URL (as of 2006-04-01): `http://www.RexxInfo.org/`

[RexxLA] URL (as of 2006-04-01): `http://www.RexxLA.org`

[VeTrUr] Veneskey G.L., Trosky W., Urbaniak J.J.: "Object Rexx by Example", Aviar. URL (as of 2006-04-01): `http://www.oops-web.com/orxbyex/`