ORX_ANALYZE.CMD - A PROGRAM FOR ANALYZING DIRECTIVES AND SIGNATURES OF OBJECT REXX PROGRAMS

Rony G. Flatscher

Department of Management and Information Systems

Vienna University of Economics and Business Administration

„7th International Rexx Symposium“, Austin/Texas, May 13th-15th, 1996

ABSTRACT

Once one gets accustomed to Object Rexx, large projects (applications) are very feasible to be coded in Object Rexx, which gains tremendeous additional power by the object oriented features and the various scopes defined for it.

The more classes and methods get defined, the more the handy „`REQUIRES`“ directive gets used, the more it becomes important to have an overview of the implemented classes, methods and routines.

In the case of nested „`REQUIRES`“-directives, which allow one to merge code from other programs by incorporating it into the „local environment“, it is possible for programmers of large projects to loose the overview about the scopes.

`ORX_ANALYZE.CMD` is an Object Rexx program which analyzes Object Rexx programs.

As the main emphasis lies in the structure of the analyzed programs, all relevant information of procedures, routines, classes and methods is gathered and stored with objects available for further use. As `ORX_ANALYZE.CMD` resolves „`REQUIRES`“ directives

too, it becomes possible to further analyze the gathered data with respect to programs the analyzed program depends on. Resolution of „REQUIRES" directives occurs recursively.

The classification tree for storing the found data as well as some additional data structures used to interrelate the gathered objects will be explained.

1 INTRODUCTION

Writing Object Rexx programs is simple. A few line of codes may constitute a powerful script due to the features available thru Object Rexx. After the programmer got accustomed to the object oriented paradigma, taking advantage of it usually means that full-fledged programs get written in Object Rexx, exploiting features like the „::REQUIRES" directive to have other Object Rexx programs loaded into the local environment and using classes and routines defined there.

If there are several Object Rexx programs involved in a project which in turn may be dependent on another set of Object Rexx programs the danger arises that one looses the overview of all parts working together. This is more so true if code has to be maintained after a couple of months at a time where the overview for an older project might have been faded or forgotten altogether. This may lead to the undesirable situation that a lot of effort (time and money) has to be invested in order to reclaim lost overview or to get acquainted with programs written by someone else, e.g. knowing which classes and routines are available from which Object Rexx programs in the program to be maintained.

A program which was able to analyze Object Rexx programs and capable of analyzing those programs loaded into the local environment via „::REQUIRES" directives would help alleviate the aforementioned problem by effectively allowing a programmer to format and dump the relevant data in a form he or she wishes.

2 „ORX_ANALYZE“ OBJECTS

„ORX_ANALYZE.CMD“ was written for analyzing Object Rexx programs. It is basically an Object Rexx parser which collects information on certain key aspects and stores them in collections of Object Rexx objects and in the local environment for further processing.

In the case that an Object Rexx program uses the „::REQUIRES“ directive the appropriate files are recursively loaded and analyzed the very same way, taking care in the process that no circularity phenomenon occurs (a program requiring an already analyzed program). In any case all classes available via the present Object Rexx classification tree, starting with the class object „.object“ are analyzed too.

```
def
  def_class
  def_file
  def_method
  def_procedure
     def_label
  def_requires
  def_routine
```

Figure 1: Classification tree for classes defined in ORX_ANALYZE

„ORX_ANALYZE“ defines a classification tree as depicted in figure 1. With the help of this classification all the relevant information for Object Rexx programs are stored.

In this section every single class is described alphabetically and its methods and object variables are explained. Please note that in most cases the instance and class object variables are made directly accessible by defining *attribute* methods for them. *Attribute methods* represent in effect *object variables*.

Class „DEF“

This class serves as the root and defines those methods and object variables which are common to most of its subclasses. Figure 2 shows the structure of this class.

The following class methods and class object variables are defined:

„max_name_length" - contains the maximum length of the names of the DEF-object found in the analyzed program which gets stored.

„total_objects" - contains the total number of DEF-objects.

„User_Slot" - a directory which is not used by ORX_ANALYZE. It is intended for allowing programmers to store (e.g. statistical) data about *all* the DEF-objects stored.

These class definitions get inherited by all subclasses, so they are available in every subclassed class object.

The following instance methods and instance object variables are defined:

„DumpAttributes" - returns an empty string, is implemented in some subclasses and returns a string describing the found attributes.

„Errors" - a list of errors found during parsing.

„Exits" - a directory containing all of the found EXIT-statements.

„IsFunction" - a boolean indicating whether program, label or routine returns a value and if so is set to .true.

„IsProcedure" - a boolean indicating whether a program, a label or a routine is used as a procedure (set if arguments are PARSEd or USEd).

„LineNr" - contains .nil if referring to data gathered from .environment, the line number in the source file otherwise.

„Local_Labels" - a directory of labels found in a program, a procedure, a routine or a method.

„Local_Procedures“ - a directory of procedures found in a program, a routine or a method.

„Name“ - name of the object in question.

„Returns“ - a directory of EXIT, RAISE, RETURN or SIGNAL statements found in a program, a procedure (function), a routine or a method.

„Signatures“ - a directory of statements which try to use or parse arguments found in a program, a procedure (function), a routine or a method.

„Type“ - returns a string indicating the type of the DEF-object; return values are implemented in subclasses.

„User_Slot“ - a directory which is not used by ORX_ANALYZE. It is intended for allowing programmers to store data about an individual DEF-object.

```
[::CLASS [def]              ]

The following object variable(s) was (were) found at CLASS scope:
      1 [max_name_length]
      2 [total_objects]
      3 [User_Slot]

CLASS METHOD(s):

      1 [::METHOD [INIT]                 class]
      2 [::METHOD [max_name_length]      class attribute]
      3 [::METHOD [total_objects]        class attribute]
      4 [::METHOD [User_Slot]            class attribute]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [Errors]
      2 [Exits]
      3 [IsFunction]
      4 [IsProcedure]
      5 [LineNr]
      6 [Local_Labels]
      7 [Local_Procedures]
      8 [Name]
      9 [Returns]
     10 [Signatures]
     11 [User_Slot]
```

```
INSTANCE METHOD(s):

      1 [::METHOD [DumpAttributes]      ]
      2 [::METHOD [Errors]               attribute]
      3 [::METHOD [Exits]                attribute]
      4 [::METHOD [INIT]                ]
      5 [::METHOD [IsFunction]           attribute]
      6 [::METHOD [IsProcedure]          attribute]
      7 [::METHOD [LineNr]               attribute]
      8 [::METHOD [Local_Labels]         attribute]
      9 [::METHOD [Local_Procedures]     attribute]
     10 [::METHOD [Name]                 attribute]
     11 [::METHOD [Returns]              attribute]
     12 [::METHOD [Signatures]           attribute]
     13 [::METHOD [Type]                ]
     14 [::METHOD [User_Slot]            attribute]
```

Figure 2: The root class named "DEF"

Class „DEF_CLASS“

„DEF_CLASS“ is a subclass of „DEF“. Instances of this class store all relevant class information. Figure 3 shows the structure of this class, which merely defines methods for its own purpose.

ORX_ANALYZE creates a sentinel object from this class, accessible via the environment symbol „.missing.class“ to indicate that while analyzing a program for some reasons a target class was not found. This may happen e.g. if ORX_ANALYZE is run with the option of not resolving requires-directive (see below).

The following instance methods and instance object variables are defined:

„dumpAttributes“ - overrides parent's method and returns a string containing all attributes found with the stored class.

„ExposeClass“ - a directory containing all class object variables.

„ExposeInstance“ - a directory containing all instance object variables.

„External“ - .nil or the string going with the „EXTERNAL“ attribute of a class-directive.

„Inherit“ - .nil or a string containing the list of additional superclasses to inherit from (multiple inheritance).

„IsMetaClass“ - .true if class is a metaclass, .false else.

„IsUsedAsSuper“ - .true if class is subclassed, .false else, indicating a leaf class.

„ListOfSuperClasses“ - a list of DEF_CLASS objects being used as superclasses for this class.

„Local_Class_Methods“ - a directory of the *class* DEF_METHOD objects found.

„Local_Instance_Methods“ - a directory of the *instance* DEF_METHOD objects found.

„MetaClass“ - .nil or a string with the metaclass class-id.

„MetaClassObject“ - a DEF_CLASS object representing this class’ metaclass.

„MetaUsedBySet“ - a set containing all DEF_CLASS objects using this class as their metaclass.

„MixinClass“ - .nil or a string containing the superclass class-id.

„Public“ - .true if the PUBLIC attribute is set, .false else.

„SetOfSubclasses“ - a set containing all DEF_CLASS objects using this class as their superclass.

„SubClass“ - .nil or a string containing the superclass class-id.

„SuperClassObject“ - .nil if no superclass or a DEF_CLASS object representing this class’ superclass.

„Type“ - returns the string „::CLASS“

```
[::CLASS [def_class]        subclass   [def]]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [ExposeClass]
      2 [ExposeInstance]
      3 [External]
      4 [Inherit]
      5 [IsMetaClass]
      6 [IsUsedAsSuper]
      7 [ListOfSuperClasses]
      8 [Local_Class_Methods]
      9 [Local_Instance_Methods]
     10 [MetaClass]
     11 [MetaClassObject]
     12 [MetaUsedBySet]
     13 [MixinClass]
     14 [Public]
     15 [SetOfSubclasses]
     16 [SubClass]
     17 [SuperClassObject]


INSTANCE METHOD(s):

      1 [::METHOD [dumpAttributes]      ]
      2 [::METHOD [ExposeClass]          attribute]
      3 [::METHOD [ExposeInstance]       attribute]
      4 [::METHOD [External]             attribute]
      5 [::METHOD [Inherit]              attribute]
      6 [::METHOD [INIT]                ]
      7 [::METHOD [IsMetaClass]          attribute]
      8 [::METHOD [IsUsedAsSuper]        attribute]
      9 [::METHOD [ListOfSuperClasses]   attribute]
     10 [::METHOD [Local_Class_Methods]  attribute]
     11 [::METHOD [Local_Instance_Method attribute]
     12 [::METHOD [MetaClass]            attribute]
     13 [::METHOD [MetaClassObject]      attribute]
     14 [::METHOD [MetaUsedBySet]        attribute]
     15 [::METHOD [MixinClass]           attribute]
     16 [::METHOD [Public]               attribute]
     17 [::METHOD [SetOfSubclasses]      attribute]
     18 [::METHOD [SubClass]             attribute]
     19 [::METHOD [SuperClassObject]     attribute]
     20 [::METHOD [Type]                ]
```

Figure 3: Class "DEF_CLASS" for storing class related information

Class „DEF_FILE"

„DEF_FILE" is a subclass of „DEF". Instances of this class store all relevant file information. Figure 4 shows the structure of this class, which merely defines methods for its own purpose.

Instances of this class carry all the information gathered from analyzing a specific file and therefore serve as a central repository for undertaking further analysis. For the classes found in the Object Rexx environment a pseudo DEF_FILE object is created, named „.environment".

The following instance methods and instance object variables are defined:

„LOC" - number of lines found in the program. Note, this number represents the true amount of lines of code, i.e. empty lines are removed from the count, lines concatenated with a semicolon (;) are split and lines split over multiple lines with a comma (,) are rejoined.

„Local_Classes" - a directory containing all DEF_CLASS objects representing the classes defined in the file.

„Local_Leaf_Classes" - a set containing all DEF_CLASS objects which have no subclasses defined in the file.

„Local_Metaclasses" - a directory containing all DEF_CLASS objects representing metclasses defined in the file.

„Local_Methods" - a directory containing all DEF_METHOD objects representing the „floating" methods defined in the file.

„Local_Root_Classes" - a set of DEF_CLASS objects which have no superclass other than the Object Rexx class „Object" defined in the file.

„Local_Routines" - a directory of DEF_ROUTINE objects defined in the file.

„Required_by" - a list of DEF_FILE objects referring to the files requiring this file.

„Requires_files" - a list of DEF_FILE objects referring to the files this file requires.

„ShortName" - a string containing the filename only, i.e. a possible drive letter and the path is removed from the given name.

„Total_loc" - number of lines as is, including empty lines and the like.

„type" - returns the string „FILE".

„Visible_Classes" - a directory containing DEF_CLASS objects for all public classes defined in files loaded via the require-directive.

„Visible_Routines" - a directory containing DEF_CLASS objects for all public routines defined in files loaded via the require-directive.

```
[::CLASS [def_file]         subclass   [def]]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [LOC]
      2 [Local_Classes]
      3 [Local_Leaf_Classes]
      4 [Local_Metaclasses]
      5 [Local_Methods]
      6 [Local_Root_Classes]
      7 [Local_Routines]
      8 [Required_by]
      9 [Requires_files]
     10 [ShortName]
     11 [Total_loc]
     12 [Visible_Classes]
     13 [Visible_Routines]

INSTANCE METHOD(s):

      1 [::METHOD [INIT]                ]
      2 [::METHOD [LOC]                  attribute]
      3 [::METHOD [Local_Classes]        attribute]
      4 [::METHOD [Local_Leaf_Classes]   attribute]
      5 [::METHOD [Local_Metaclasses]    attribute]
      6 [::METHOD [Local_Methods]        attribute]
      7 [::METHOD [Local_Root_Classes]   attribute]
      8 [::METHOD [Local_Routines]       attribute]
      9 [::METHOD [Required_by]          attribute]
     10 [::METHOD [Requires_files]       attribute]
     11 [::METHOD [ShortName]            attribute]
```

```
[::METHOD [Total_loc]            attribute]
[::METHOD [type]               ]
[::METHOD [Visible_Classes]      attribute]
[::METHOD [Visible_Routines]     attribute]
```

Figure 4: Class "DEF_FILE" for storing file related information

Class „DEF_LABEL“

„DEF_LABEL“ is a subclass of „DEF_PROCEDURE“. Instances of this class store all relevant label information. A label in the context of ORX_ANALYZE is not followed by any statements trying to retrieve arguments. The only method it defines is „type“ which returns the string „LABEL“.

Class „DEF_METHOD“

„DEF_METHOD“ is a subclass of „DEF“. Instances of this class store all relevant method information. Figure 5 shows the structure of this class, which merely defines methods for its own purpose.

The following instance methods and instance object variables are defined:

„Attribute“ - .true if an attribute method, .false else.

„Class_method“ - .nil if a „floating“ method, .true if a class method, .false if an instance method.

„dumpAttributes“ - returns a string containing all attributes found with the analyzed method.

„Expose“ - a directory containing the object variables defined or used with this method.

„ExposeAsString“ - returns a string representation of all of the object variables.

„Private“ - .true if a private method, .false else.

„Protected“ - .true if a protected method, .false else.

„type“ - returns the string „::METHOD“.

„Unguarded“ - .true if an unguarded method, .false else.

```
[::CLASS [def_method]       subclass   [def]]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [Attribute]
      2 [Class_method]
      3 [expose]
      4 [Private]
      5 [Protected]
      6 [Unguarded]

INSTANCE METHOD(s):

      1 [::METHOD [Attribute]            attribute]
      2 [::METHOD [Class_method]         attribute]
      3 [::METHOD [dumpAttributes]      ]
      4 [::METHOD [Expose]               attribute]
      5 [::METHOD [ExposeAsString]      ]
      6 [::METHOD [INIT]                ]
      7 [::METHOD [Private]              attribute]
      8 [::METHOD [Protected]            attribute]
      9 [::METHOD [type]                ]
     10 [::METHOD [Unguarded]            attribute]
```

Figure 5: Class "DEF_METHOD" for storing method related information

Class „DEF_PROCEDURE“

„DEF_PROCEDURE“ is a subclass of „DEF“. Instances of this class store all relevant procedure information. Figure 6 shows the structure of this class, which merely defines methods for its own purpose.

The following instance methods and instance object variables are defined:

„dumpAttributes“ - returns a string containing all exposed variables found with the keyword instruction „PROCEDURE“.

„Expose“ - a directory containing the variables exposed to the procedure.

„Type“ - returns the string „PROCEDURE“.

```
[::CLASS [def_procedure]    subclass   [def]]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [expose]
      2 [procedure]

INSTANCE METHOD(s):

      1 [::METHOD [dumpAttributes]      ]
      2 [::METHOD [Expose]               attribute]
      3 [::METHOD [INIT]                ]
      4 [::METHOD [Type]                ]

```

Figure 6: Class "DEF_PROCEDURE" for storing procedure related information

Class „DEF_REQUIRES“

„DEF_REQUIRES“ is a subclass of „DEF“. Instances of this class store the name of the file which is required. DEF_REQUIRES is a subclass of DEF and the only method it defines is „type“ which returns the string „::REQUIRES“.

Class „DEF_ROUTINE“

„DEF_ROUTINE“ is a subclass of „DEF“. Instances of this class store all relevant routine information. Figure 7 shows the structure of this class, which merely defines methods for its own purpose.

The following instance methods and instance object variables are defined:

„dumpAttributes“ - returns either a null string or a string containing „PUBLIC“, if this routine was defined to be public.

„public“ - .true if a public routine, .false else.

„type“ - returns the string „::ROUTINE“.

```
[::CLASS [def_routine]      subclass   [def]]

The following object variable(s) was (were) found at INSTANCE scope:

      1 [public]

INSTANCE METHOD(s):

      1 [::METHOD [dumpAttributes]      ]
      2 [::METHOD [INIT]                ]
      3 [::METHOD [public]               attribute]
      4 [::METHOD [type]                ]
```

Figure 7: Class "DEF_ROUTINE" for storing routine related information

2.1 Relating the DEF Objects with Each Other

So far we have seen what kind of information is stored for which type of analyzed token. DEF_FILE objects are certainly a central point of reference with respect to the Object Rexx related tokens stored there.

In addition ORX_ANALYZE defines a stem variable, which contains indices with various objects determining which relationships exist between the various DEF-objects, like which methods belong to which class. Some variables which might be useful for other programs are made available via entries into this stem.

The stem variable is the return value of ORX_ANALYZE after parsing the Object Rexx programs. If the stem is named „ctl.“ then the various indices allow for accessing:

ctl.eEnvClassSet - a set containing all DEF_CLASS objects representing the classes which are available in the Object Rexx environment.

ctl.eFileStart - the DEF_FILE object representing the file for which the analysis was started.

ctl.eFilesSeq - a list of DEF_FILE objects in the order they got analyzed as imposed by require-directives.

ctl.eFiles - a directory relating the filenames to the DEF_FILE objects.

ctl.eFileEnvironment - the DEF_FILE object representing the Object Rexx environment classes.

ctl.eMissingClass - the DEF_CLASS object (a sentinel) representing a missing class, e.g. if ORX_ANALYZE is to ignore requires-directives while parsing, classes may be missing which are defined in required programs.

ctl.eRootObject - the DEF_CLASS object representing the class object of the Object Rexx root class „Object“.

ctl.eRootClass - the DEF_CLASS object representing the class object of the Object Rexx metaclass „Class“.

The indices for the following relations are named such that the index into a relation is the DEF-object type following the „2“. E.g. for the relation „ctl.eClasses2Files“ the index would be of type DEF_FILE object, so given that the variable „aFile“ contains a DEF_FILE object the statement „anArray = ctl.eClasses2Files ~ allat(aFile)“ would return *all* DEF_CLASS objects defined with the DEF_FILE object stored in the variable „aFile“. So in a sense the following relations constitute little databases about the facts stored about the analyzed files:

ctl.eClasses2Files - relates DEF_CLASS objects to DEF_FILE objects.

ctl.eLabels2Files - relates DEF_LABEL objects to DEF_FILE objects.

ctl.eLabels2Objects - relates DEF_LABEL objects to DEF_FILE, DEF_PROCEDURE, DEF_METHOD or DEF_ROUTINE objects.

ctl.eMethods2Files - relates DEF_METHOD objects to DEF_FILE objects.

ctl.eProcedures2Files ~ relates DEF_PROCEDURE objects to DEF_FILE objects.

ctl.eProcedures2Objects - relates DEF_PROCEDURE objects to DEF_FILE, DEF_METHOD or DEF_ROUTINE objects.

ctl.eRequires2Files - relates DEF_REQUIRES objects to DEF_FILE objects.

ctl.eRoutines2Files - relates DEF_ROUTINE objects to DEF_FILE objects.

ctl.eToken2Files - relates objects of type DEF_CLASS, DEF_LABEL, DEF_METHOD, DEF_PROCEDURE, DEF_REQUIRES, DEF_ROUTINE to DEF_FILE objects.

ctl.eMethods2Class - relates a DEF_METHOD object to a DEF_CLASS object, irrespectible whether it represents a class or instance method.

ctl.eClassMethods2Class - relates a class method DEF_METHOD object to a DEF_CLASS object.

ctl.eInstanceMethods2Class - relates an instance method DEF_METHOD object to a DEF_CLASS object.

2.2 Defined Environment Symbols in the Local Environment

ORX_ANALYZE defines a couple of environment symbols in the Object Rexx supplied directory accessible via the environment symbol „.local“ to refer to objects which are heavily used by ORX_ANALYZE or programs which utilize the resulting DEF-objects. This way every Rexx program in the local environment is able to access the objects these environment symbols represent.

.bQueryEnvClassMethods - by default .true, causing ORX_ANALYZE to retrieve the methods defined with the object classes found in Object Rexx environment. The programmer may set the value for this variable *before* ORX_ANALYZE is called, e.g. with the statement „.local ~ bQueryEnvClassMethods = .false“.

The following environment symbols are defined for the programmer's convenience:

.Env.Object - the DEF_CLASS object representing the class object of the Object Rexx root class „Object“ (same as ctl.eRootObject).

.Env.Class - the DEF_CLASS object representing the class object of the Object Rexx metaclass „Class“ (same as ctl.eRootClass).

.Env.FileObj - the DEF_FILE object representing the Object Rexx environment (same as ctl.eFileEnvironment).

.missing.class - returns the DEF_CLASS object (a sentinel) representing a missing class, e.g. if ORX_ANALYZE is to ignore requires-directives while parsing, classes may be missing which are defined in required programs (same as ctl.eMissingClass).

3 INVOKING ORX_ANALYZE

For invoking ORX_ANALYZE the following syntax is used:

```
ORX_ANALYZE rexx_program[, switch]
```

By default ORX_ANALYZE parses the given Rexx program and resolves any requires-directives. It will return the stem variable containing the results.

If the second (optional) argument „switch“ has a value of „1“ ORX_ANALYZE will *not* resolve requires-directives. This may cause some errors in the result for classes which are referred but are defined in the required programs. In such a case, missing classes are indicated by the usage of the .missing.class object in place of the missing classes. Still, it will return the stem variable containing all the above described objects.

A value of „2“ for the second argument merely returns an array containing a „stripped“ version of the Object Rexx program, each array item representing a different line of code. The „stripped“ version does not contain any comments, nor blank lines, in addition multiple statements on a line delimited by a semi-colon are split into separate lines. If statements span multiple lines (indicated with a trailing comma at the end of a line of code) they are put back into one line.

4 CONCLUSIONS

By reading this paper it should have become clear what type of information is relevant if attempting to analyze the structure of Object Rexx programs. The results of the parsing, which may lead to analyzing additional Object Rexx programs if requires-directives are used, are stored in a class hierarchy developed for this purpose only and starting with the string „`DEF`“. All `DEF` objects are collected and put into different Object Rexx collection objects, mostly into relationship objects. The relationship objects allow for researching and manipulating all the found information and thereby allowing for analyzing the collected data even further.

The author wrote an additional program, `ORX_ANALYZE_ASCII.CMD` (with the name of the file to be analyzed as the only argument), which uses the gathered data for creating various statistics and reports, allowing for seeing at a glance which Object Rexx programs have which program scopes. Classes and routines are depicted with the files they are defined in, so it becomes easy to track down the places (programs) where they are defined and where they are used. The figures in this book are based on the output of that program, which merely worked on the data `ORX_ANALYZE.CMD` generated while analyzing itself. `ORX_ANALYZE_ASCII.CMD` will be made available with source on the internet as an example how to use the results of running `ORX_ANALYZE.CMD`.

5 REFERENCES

Online documentations of various beta versions of Object Rexx (the version used for this paper stems from February 16th, 1996).

Various postings on the internet newsgroup „`comp.lang.rexx`“, 1995-1996.

Flatscher, R.G.: „Local Environment and Scopes in Object Rexx“, in: Proceedings of the "7th International REXX Symposium, May 12-15, Texas/Austin 1996", The Rexx Language Association, Raleigh N.C. 1996.

Flatscher, R.G.: „Object Classes, Meta Classes and Method Resolutions in Object Rexx“, in: Proceedings of the "7th International REXX Symposium, May 12-15, Texas/Austin 1996", The Rexx Language Association, Raleigh N.C. 1996. .

Date of Article: 1996-06-02.

Published in: Proceedings of the "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association, Raleigh N.C. 1996.

Presented at: "7th International REXX Symposium", Texas/Austin, May 12th-15th, 1996, The Rexx Language Association.