Devising a TraceObject Class for Improved Runtime Monitoring of ooRexx Applications

Rony G. Flatscher
Vienna University of Economics and Business
Vienna, Austria
rony.flatscher@wu.ac.at

Till Winkler
Vienna University of Economics and Business
Vienna, Austria
till.winkler@wu.ac.at

Abstract

The dynamic programming language ooRexx includes a runtime monitoring keyword instruction named TRACE. This keyword instruction produces a trace line for each traced statement. Although additional trace information was added, ooRexx 5.0.0 is missing important information, such as tracing threads, whether a method is currently guarded, the guard locks state, and the interpreter instance. With the introduction of a specific trace-related class, TraceObject, it is now possible to maintain all ooRexx-related trace information for each traced statement in a proper directory that can be analyzed at runtime. This article will introduce the *TraceObject* class and its properties, including the ability to collect all produced trace objects for inspection and analysis. It focuses on the trace line (the trace object's string) representation that can be formatted to include the missing aspects for analyzing dynamic, multi-threaded, guarded methods on multi-instance ooRexx applications.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Runtime Verification, Multi-threaded, ooRexx, REXX

ACM Reference Format:
Rony G. Flatscher and Till Winkler. 2024. Devising a TraceObject Class for Improved Runtime Monitoring of ooRexx Applications. In *Proceedings of the 7th ACM International Workshop on Verification and Monitoring at Runtime Execution (VORTEX '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3679008.3685543

1 Introduction

With the introduction of the REXX [1] language on IBM mainframes in 1979, the TRACE keyword instruction for runtime monitoring of REXX programs became available. This built-in TRACE functionality allows programmers to turn on and off tracing at runtime, control the details of the produced trace line, and enter or leave interactive mode to inspect the runtime state further.

Influenced by Smalltalk and its message paradigm [4], IBM released an object-oriented follow-on to the REXX language called Object REXX in 1996 [7]. Despite being backward compatible with REXX, it added the ability to define classes, methods, attributes, and implemented message expressions, which get used to interacting with objects. The produced trace line was extended to allow for tracing message expressions and the resolution of environment symbols. However, the runtime monitoring information lacked information on essential properties in the context of dispatching messages dynamically on different threads.

IBM donated the source code of Object REXX to the non-profit special interest group "Rexx Language Association (RexxLA)," which used it to create an open-source version with the name "open object Rexx (ooRexx)" in 2004 [6]. In the years to follow, ooRexx got a new kernel and a new native application programming interface, which allows for creating and using multiple Rexx interpreter instances in the same process, which can share Rexx objects and dispatch messages on multiple threads. However, the produced trace lines when monitoring at runtime the execution of Rexx programs would not reflect all the relevant context information like the interpreter instance, the thread a message gets dispatched on, the state of methods, the state of guard locks, and the like.

Tracking the concurrent execution of applications can generally be difficult, and potential errors might be hard to understand. For example, the average troubleshooting time for concurrent MySQL projects is 2.58 times higher than for their sequential counterparts [5]. To understand the dynamics of multi-threaded and object-oriented programs, programmers must be provided with information that allows them to identify threads, invocations, and attribute pools used, if necessary. Complex runtime mixtures, for instance, like the Java bindings for ooRexx [8] and employing JavaFX with it [2], use separate Rexx interpreter instances to control different JavaFX scenes. In such scenarios, providing context information to identify the instances and their scope is necessary to understand the dynamics and the problems that may result.

This article first introduces the TRACE keyword instruction for runtime monitoring REXX programs in section 2 and explicates the essential trace options with a little nutshell example. Subsequently, the future release candidate ooRexx 5.1 (r12845) and its implementation of the *TraceObject* class, which provides all relevant context information during tracing, is discussed in Section 3.

2 The TRACE Keyword Instruction

The REXX language defines a TRACE keyword instruction and a built-in function called TRACE() with options to start and stop tracing, the tracing details, and whether a trace should stop for interactive inspection. When a REXX program gets executed, its trace option will be set to *'normal'* by default, informing the user of failing commands with appropriate trace lines. The name of the



This work is licensed under a Creative Commons Attribution 4.0 International License.

VORTEX '24, September 19, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1119-0/24/09
https://doi.org/10.1145/3679008.3685543

TRACE option starts with different letters, such that one can only supply the first letter instead of the full name of the TRACE option.

2.1 TRACE Normal

The trace option *'normal'* is set by default, generating a trace line in case a (external) command fails. Listing 1 demonstrates the three basic REXX instruction types: assignment, keyword, and command instruction. A command instruction is neither an assignment nor a string that starts with a REXX keyword and will be handed over to the operating system for execution by the REXX interpreter [9]. The command's return code is available upon return, setting a REXX variable *RC*. While the command *'echo'* exists on Windows and Unix systems, the explicitly addressed command handler *nope* in line 8 does not and, therefore, will cause its command to fail, which causes appropriate trace lines to be produced at runtime. In Listing 2, the three trace lines inform the user of the failing command.

Listing 1: Three basic REXX instruction types

```
1 a="hello, world"       -- an assignement instruction
2 say 'a='''a''''        -- a SAY keyword instruction
3 say "TRACE option in effect:" trace() -- 'N'-'Normal'
4 say
5 'echo' a               -- a known system command
6 say "return code:" rc -- return code
7 say
8 address nope 'nixi' a -- 'nope' not available
9 say "return code:" rc -- return code
```

Listing 2: TRACE output of code in Listing 1

```
a="hello, world"
TRACE option in effect: N

hello, world
return code: 0

     8 *-* address nope 'nixi' a -- 'nope' not available
       >>>   "nixi hello, world"
       +++   "RC(30)"
return code: 30
```

2.2 TRACE All, Results, Intermediates

TRACE can display each statement before its execution to monitor the execution of REXX statements so that the control flow can be studied. The TRACE option *'Results'* will also create a trace line for each expression the statement produced, which is the recommended option. If one wishes to have all details of an expression traced, the TRACE option *'Intermediates'* can be used.

Listing 3 depicts a REXX program demonstrating the trace lines produced depending on the TRACE option in effect. It loops over the options *'All'*, *'Results'*, and *'Intermediates'*; the output is shown in Listing 4. As can be seen, the trace lines for *'Intermediates'* make it possible to analyze and understand the evaluation of the expression *a=100+random()*, which uses the REXX built-in function *RANDOM()*, generating a different random number on each run and thus giving a different value for the variable a.

Listing 3: Looping through TRACE options

```
1 do opt over 'All', 'Results', 'Intermediates'
2    say "setting TRACE to'"opt"':"
3    currOpt=trace(opt) -- TRACE (function)
4    a=100+random() -- line # 4
5    say "a:" a     -- line # 5
6    TRACE N        -- line # 6 (keyword)
7    say "---"
8 end
```

Listing 4: TRACE outputs of code in Listing 3

```
setting TRACE to'All':
     4 *-*   a=100+random() -- line # 4
     5 *-*   say "a:" a     -- line # 5
a: 1088
     6 *-*   TRACE N        -- line # 6 (keyword)
---
setting TRACE to'Results':
     4 *-*   a=100+random() -- line # 4
       >>>     "436"
     5 *-*   say "a:" a     -- line # 5
       >>>     "a: 436"
a: 436
     6 *-*   TRACE N        -- line # 6 (keyword)
---
setting TRACE to'Intermediates':
       >F>     TRACE => "N"
       >=>     CURROPT <= "N"
     4 *-*   a=100+random() -- line # 4
       >L>     "100"
       >F>     RANDOM => "514"
       >O>     "+" => "614"
       >>>     "614"
       >=>     A <= "614"
     5 *-*   say "a:" a     -- line # 5
       >L>     "a:"
       >V>     A => "614"
       >O>     " " => "a: 614"
       >>>     "a: 614"
a: 614
     6 *-*   TRACE N        -- line # 6 (keyword)
---
```

Listing 5 depicts an ooRexx program that defines a class, a *'guarded'* and an *'unguarded'* method, creates two instances, and sends messages (*hey* and *ho*) synchronously to the two instances and then asynchronously by employing the *start* message that each ooRexx object can understand. The receiving instances will then locate and invoke the methods with the same name as the message. The TRACE implementation of ooRexx adds trace lines that allow for tracing the beginning of an invocation (trace prefix *>I>*) and its end (trace prefix *<I<*), as can be seen in Listing 6. Studying the trace lines in Listing 6, it becomes clear that no information is present that would allow for identifying threads, invocations, or attribute pools in use, as well as information about the guard state of a method or the guard lock state. This information is, however, vital in analyzing and understanding the dynamics of a multi-threaded, message-based, object-oriented, dynamic program in complex deployment scenarios.

Listing 5: Class definition with two methods

```
1 t1=.test~new       -- instance
2 t2=.test~new       -- instance
3 t1~hey
4 t2~ho
5 say "-"~copies(50)
6 t2~start('hey')    -- async
7 t1~start('ho')     -- async
8 say "-"~copies(50)
```

```
 9
10  ::class test       -- some Rexx class
11  ::method hey       -- by default: guarded
12    say 'hey (guarded)'
13
14  ::method ho unguarded    -- unguarded
15    say 'ho (unguarded)'
16
17  ::options trace all
```

Listing 6: TRACE output of code in Listing 5

```
     1 *-* t1=.test~new      -- instance
     2 *-* t2=.test~new      -- instance
     3 *-* t1~hey
       >I> Method "HEY" with scope "TEST"
    12 *-* say 'hey (guarded)'
hey (guarded)
       <I< Method "HEY" with scope "TEST"
     4 *-* t2~ho
       >I> Method "HO" with scope "TEST"
    15 *-* say 'ho (unguarded)'
ho (unguarded)
       <I< Method "HO" with scope "TEST"
     5 *-* say "-"~copies(50)
--------------------------------------------------
     6 *-* t2~start('hey')   -- async
     7 *-* t1~start('ho')    -- async
     8 *-* say "-"~copies(50)
--------------------------------------------------
       >I> Method "HEY" with scope "TEST"
    12 *-* say 'hey (guarded)'
hey (guarded)
       >I> Method "HO" with scope "TEST"
    15 *-* say 'ho (unguarded)'
ho (unguarded)
       <I< Method "HEY" with scope "TEST"
       <I< Method "HO" with scope "TEST"
```

It would be possible to supply these essential but missing pieces of information in a trace line. However, when adding such information to the trace line, it would be beneficial to make all relevant trace information available as a directory object (mapped collection). This would simplify fetching multi-threaded and object-related runtime information, as parsing a trace line string is no longer necessary. In addition, it would allow the relevant stack trace object to be made directly available for further inspection at runtime.

3 The ooRexx TraceObject Class

For ooRexx 5.1, the first authors devised and implemented a *TraceObject* class which subclasses the ooRexx *StringTable* class (a directory-like class) [3], allowing to store all relevant trace information with indexes that are descriptive strings:

- `ATTRIBUTEPOOL`, an identifier (a string prefixed with A)
- `HASSCOPELOCK`, true or false (if true, depicted with an asterisk in a string)
- `INTERPRETER`, an identifier (a string prefixed with R)
- `INVOCATION`, an identifier (a string prefixed with I)
- `ISGUARDED`, true or false (if true, depicted with a G, else with U in a string)
- `NUMBER`, a sequential number
- `OPTION`, the trace option in effect
- `RECEIVER`, the receiver object
- `SCOPELOCKCOUNT`, a number (a string prefixed with L)
- `STACKFRAME`, the *StackFrame* object
- `THREAD`, an identifier (a string prefixed with T)
- `TIMESTAMP`, a *DateTime* object of the trace object creation
- `TRACELINE`, the trace line to display (a string)
- `VARIABLE`, the traced variable (a *StringTable* with the entries `NAME`, `VALUE`, `ASSIGNMENT`)

By default, the TRACELINE entry of a trace object will be displayed. When the ooRexx interpreter needs to use a string representation of an object, it will send the *makeString* message. Implementing a method named *makeString* allows control of the formatting of such a string. Therefore, an instance method called *makeString* is implemented in the *TraceObject* class. To determine the desired formatting of trace lines, an option class attribute gets defined, which can be set to *'Normal'* (default), *'Thread'* (thread identifier in the trace prefix), *'Standard'* (extended trace information in brackets), and *'Full'* (extended trace information in brackets) formatting.

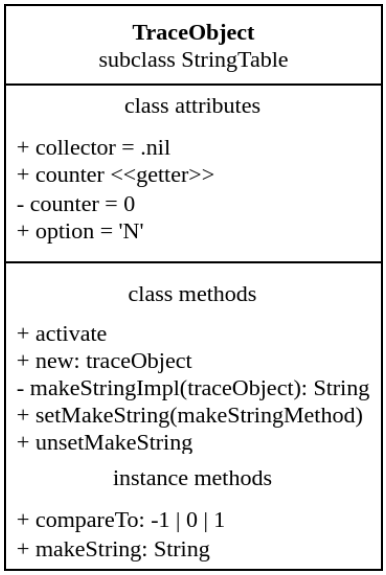


Figure 1: The TraceObject class

To set the trace line to *'Thread'* formatting, one must assign the value *'Thread'* to the class attribute *option*, which expects a Rexx object that understands the *append* message. As all ooRexx *OrderedCollection* classes like the *Array* class, understand the *append* message, they can be used to collect all created trace objects as the *TraceObject* class will send them as an argument with the *append* message. This way, one can collect and save trace objects for later analyses or reformatting purposes. There is even a specific option, *'Profiling'* or *'Probing'*, that would inhibit the output of the trace lines such that collecting trace objects would not distort the output of running programs and would allow the collection of trace objects.

3.1 Collecting TraceObjects

For each trace, a *TraceObject* gets created, and if the class attribute *collector* is not *.nil*, it gets appended to it. If one outputs trace objects later, they will get formatted according to the setting of the class

attribute *option* at that time. This makes creating trace logs easy and allows different formatting options later for creating the trace lines to ease analysis. It would even be possible to develop a tailored *makeString* method and supply, e.g., the sequence number with the time stamp in addition to the trace line.

Listing 7: Collecting and displaying trace objects

```
arrTraceLog=.array~new  -- array for collecting
.TraceObject~collector=arrTraceLog
.TraceObject~option='Profile'
t=.test~new         -- create object
say t~show_m1       -- synchronous message
msg=t~start(show_m1) -- asynchronous message
say "msg~result:" msg~result  -- waits for result
say

.TraceObject~collector=.nil    -- stop appending
say ".TraceObject~counter:" .TraceObject~counter
say "arrTraceLog has" arrTraceLog~items "trace obj.:"
.TraceObject~option='Thread'  -- set format option
do traceObject over arrTraceLog
  say traceObject    -- string value by makeString
end

::class test
::method m1       unguarded
  trace results
  return 'hello, my beloved world!'

::method show_m1
  trace results
  reply self~m1    -- return and new thread
  say "about to return from show_m1 ..."
```

Listing 7 depicts an ooRexx program that collects all trace objects that get created while running the methods M1 and SHOW_M1 (the REPLY keyword statement causes the remaining statements of the method to run on a separate thread) of the TEST class. Setting the class attribute option to *'Profiling'* inhibits any trace line output while collecting the trace objects. The program demonstrates creating a TEST instance and sending it the message SHOW_M1 synchronously and asynchronously on a new thread (fetching its returned message object that allows for waiting on a result). The program then stops collecting trace objects from this point on by setting the class attribute *collector* to *.nil* and changes the class attribute option to format the string rendering of trace objects to include the thread id in its thread prefix and then displays all collected trace objects one by one in a loop. The instance method *makeString* will get invoked to create a string rendering of each trace object honoring the setting of the class attribute *option*. Listing 8 depicts the program's output in Listing 7 above.

Listing 8: Output of Listing 7

```
hello, my beloved world!
about to return from show_m1 ...
about to return from show_m1 ...
msg~result: hello, my beloved world!

.TraceObject~counter: 24
arrTraceLog has 24 trace obj.:
       >I1> Method "SHOW_M1" with scope "TEST"
    25 *-1* reply self~m1    -- return and new thread
       >I1> Method "M1" with scope "TEST"
    21 *-1* return 'hello, my beloved world!'
       >>1>   "hello, my beloved world!"
       <I1< Method "M1" with scope "TEST"
       >>1>   "hello, my beloved world!"
       <I1< Method "SHOW_M1" with scope "TEST"
       >I2> Method "SHOW_M1" with scope "TEST"
    26 *-2* say "about to return from show_m1 ..."
       >>2>   "about to return from show_m1 ..."
       <I2< Method "SHOW_M1" with scope "TEST"
       >I3> Method "SHOW_M1" with scope "TEST"
    25 *-3* reply self~m1    -- return and new thread
       >I3> Method "M1" with scope "TEST"
    21 *-3* return 'hello, my beloved world!'
       >>3>   "hello, my beloved world!"
       <I3< Method "M1" with scope "TEST"
       >>3>   "hello, my beloved world!"
       <I3< Method "SHOW_M1" with scope "TEST"
       >I2> Method "SHOW_M1" with scope "TEST"
    26 *-2* say "about to return from show_m1 ..."
       >>2>   "about to return from show_m1 ..."
       <I2< Method "SHOW_M1" with scope "TEST"
```

3.2 TraceObject Formatting Options

As this article is concerned with gaining new insights at runtime by supplying vital additional information to the trace line, the remainder will discuss and demonstrate the information provided according to the different (formatting) options. It starts with adding thread-related information to the trace prefix, which can be done by assigning the *TraceObject* class attribute *option* the string *"Thread"* (.TraceObject option="Thread").

Listing 9 shows the output of running the program in Listing 5 above with the class attribute *option* set to *"Thread"* and can be compared to the output in Listing 6 above. As can be seen, the thread information gets inserted as the second to last part of the trace prefix. The threads and the runtime behavior analyzed concerning them can be identified with this information.

Listing 9: Output of Listing 5 in thread format

```
     1 *-2* t1=.test~new      -- instance
     2 *-2* t2=.test~new      -- instance
     3 *-2* t1~hey
       >I2> Method "HEY" with scope "TEST"
    12 *-2* say 'hey (guarded)'
hey (guarded)
       <I2< Method "HEY" with scope "TEST"
     4 *-2* t2~ho
       >I2> Method "HO" with scope "TEST"
    15 *-2* say 'ho (unguarded)'
ho (unguarded)
       <I2< Method "HO" with scope "TEST"
     5 *-2* say "-"~copies(50)
--------------------------------------------------
     6 *-2* t2~start('hey')   -- async
     7 *-2* t1~start('ho')    -- async
     8 *-2* say "-"~copies(50)
--------------------------------------------------
       >I3> Method "HEY" with scope "TEST"
    12 *-3* say 'hey (guarded)'
hey (guarded)
       >I4> Method "HO" with scope "TEST"
    15 *-4* say 'ho (unguarded)'
ho (unguarded)
       <I3< Method "HEY" with scope "TEST"
       <I4< Method "HO" with scope "TEST"
```

Setting the formatting to *"Standard"* will yield the output in Listing 10 for the same program. The extended bracketed trace prefix

now includes the information about the thread and the invocation identifier, and in the case of messages, the guard state of the invoked method, the guard lock count, and an asterisk if the guarded method possesses the guard lock and is therefore allowed to run. The guard lock will not control unguarded methods, and as such, unguarded methods are always runnable.

Listing 10: Output of Listing 5 in standard format

```
[T2 I2]            1 *-* t1=.test~new      -- instance
[T2 I2]            2 *-* t2=.test~new      -- instance
[T2 I2]            3 *-* t1~hey
[T2 I3 G L0  ]     >I> Method "HEY" with scope "TEST"
[T2 I3 G L1 *]  12 *-* say 'hey (guarded)'
hey (guarded)
[T2 I3 G L0  ]     <I< Method "HEY" with scope "TEST"
[T2 I2]          4 *-* t2~ho
[T2 I4 U L0  ]     >I> Method "HO" with scope "TEST"
[T2 I4 U L0  ]  15 *-* say 'ho (unguarded)'
ho (unguarded)
[T2 I4 U L0  ]     <I< Method "HO" with scope "TEST"
[T2 I2]          5 *-* say "-"~copies(50)
-------------------------------------------------
[T2 I2]          6 *-* t2~start('hey')   -- async
[T2 I2]          7 *-* t1~start('ho')    -- async
[T2 I2]          8 *-* say "-"~copies(50)
-------------------------------------------------
[T3 I5 G L0  ]     >I> Method "HEY" with scope "TEST"
[T3 I5 G L1 *]  12 *-* say 'hey (guarded)'
hey (guarded)
[T4 I6 U L0  ]     >I> Method "HO" with scope "TEST"
[T4 I6 U L0  ]  15 *-* say 'ho (unguarded)'
ho (unguarded)
[T3 I5 G L0  ]     <I< Method "HEY" with scope "TEST"
[T4 I6 U L0  ]     <I< Method "HO" with scope "TEST"
```

If complex ooRexx applications need to be monitored and analyzed at runtime, the *"Full"* formatting option can be used; its output for the program in Listing 5 above can be studied in Listing 11. The information supplied in the bracketed extended trace prefix adds the Rexx interpreter instance identifier and the attribute pool identifier (attribute pools with the same identifier belong to the same instance of the same execution scope, which denotes the respective class object). An example of such a complex runtime mix is taking advantage of the Java bindings for ooRexx [8] and employing JavaFX [2] with it, as the Java bindings will use separate Rexx interpreter instances for controlling different JavaFX scenes. Sharing Rexx objects among the different Rexx interpreter instances is possible in such scenarios. Identifying the instances and their scopes uniquely may be necessary to understand the dynamics.

Listing 11: Output of Listing 5 in full format

```
[R1 T2 I2]            1 *-* t1=.test~new      -- instance
[R1 T2 I2]            2 *-* t2=.test~new      -- instance
[R1 T2 I2]            3 *-* t1~hey
[R1 T2 I3 G A1 L0  ]   >I> Method "HEY" with scope "TEST"
[R1 T2 I3 G A1 L1 *] 12 *-* say 'hey (guarded)'
hey (guarded)
[R1 T2 I3 G A1 L0  ]   <I< Method "HEY" with scope "TEST"
[R1 T2 I2]            4 *-* t2~ho
[R1 T2 I4 U A2 L0  ]   >I> Method "HO" with scope "TEST"
[R1 T2 I4 U A2 L0  ] 15 *-* say 'ho (unguarded)'
ho (unguarded)
[R1 T2 I4 U A2 L0  ]   <I< Method "HO" with scope "TEST"
[R1 T2 I2]            5 *-* say "-"~copies(50)
-------------------------------------------------
[R1 T2 I2]            6 *-* t2~start('hey')   -- async
[R1 T2 I2]            7 *-* t1~start('ho')    -- async
[R1 T2 I2]            8 *-* say "-"~copies(50)
-------------------------------------------------
[R1 T3 I5 G A2 L0  ]   >I> Method "HEY" with scope "TEST"
[R1 T3 I5 G A2 L1 *] 12 *-* say 'hey (guarded)'
hey (guarded)
[R1 T4 I6 U A1 L0  ]   >I> Method "HO" with scope "TEST"
[R1 T4 I6 U A1 L0  ] 15 *-* say 'ho (unguarded)'
ho (unguarded)
[R1 T3 I5 G A2 L0  ]   <I< Method "HEY" with scope "TEST"
[R1 T4 I6 U A1 L0  ]   <I< Method "HO" with scope "TEST"
```

3.3 Collecting and Formatting TraceObjects

In this section, Listing 12 depicts a program that employs the GUARD keyword statement for synchronization purposes in an unguarded method. ooRexx's multithreading is guard lock based such that guarded methods can only run if they hold the guard lock. Unguarded methods must acquire the guard lock to execute the GUARD keyword statement; if using GUARD ON, unguarded methods change their state to guarded from that moment on in that particular invocation. On the other hand, guarded methods execute only if they possess the guard lock but can change their state in that invocation to unguarded, issuing the GUARD OFF keyword statement.

So, the GUARD keyword statement can be used to change the guard state of a method at runtime to OFF or ON for that particular invocation. Using the subkeyword WHEN for testing attributes allows these to be used as control variables. Listing 12 displays such a program where in the unguarded method *wait* of the *Test* class, the attribute *lock* gets used as a control variable. Anyone who invokes the *wait* method will be put on hold in line number 28 until the control variable is set to *.false* on another thread.

To become able to analyze the dynamics in this case *TraceObject's* extended format *"Standard"* gets assigned to the class attribute *option* such that the extended trace prefix gets prepended to allow analyzing the execution of the program in Listing 12.

Listing 12: Collecting trace objects for guarded method

```
1  trace off       -- do not trace prologue
2  .TraceObject~option="P" -- Profile/Probe
3  .TraceObject~collector=.array~new
4  o=.Test~new     -- create an instance
5  say "starting worker asynchronously ..."
6  o~start("do")   -- dispatched on a new thread
7  say "about to wait for worker to end ..."
8  o~wait          -- synchronize with worker
9  say "waiting is over"
10 trace off
11 arr=.TraceObject~collector  -- get collection
12 .TraceObject~collector=.nil -- stop collecting
13 say
14 say 'setting: .TraceObject~option="Standard"'
15 .TraceObject~option="S"
16 say "displaying" arr~items "collected trace objects:"
17 do traceObj over arr
18    say traceObj
19 end
20
21 ::class Test
22 ::method init   -- constructor
23   expose lock   -- access attribute
24   lock=.true    -- set default
```

```
25
26 ::method wait unguarded
27   expose lock  -- access attribute
28   guard on when lock=.false
29
30 ::method do    -- do some work
31   expose lock  -- access attribute
32   t=random(1,999)/1000
33   say "do: working for" t "secs"
34   call sysSleep t
35   say "do: ending work"
36   lock=.false  -- release lock
37
38 ::options trace results  -- trace everything
```

Listing 13 shows the output of running the program in Listing 12 above. The generated trace objects get collected, and upon the return of the dispatched *wait* message in line number 8 in Listing 12, the main program sets the *TraceObject* class attribute *option* to "Standard", such that the extended trace prefix gets used for formatting the trace line string. It can be seen that the unguarded method runs on the thread with the id 1 (*T1*), in which it evaluates the GUARD instruction after being able to reserve the guard lock (the asterisk indicates who holds the guard lock then) repeatedly. The first time, the evaluation of the WHEN expression yields 0 (*.false*), which keeps the instruction on hold. The next time the *lock* attribute changes its value, the GUARD instruction in line number 28 gets scheduled for re-execution and, after getting the guard lock (thereby increasing the lock count to 1), re-evaluates the WHEN expression, which yields 1 (*.true*) as the *do* method has changed the *lock* attribute in between.

Listing 13: Output of Listing 12 in standard format

```
     1 *-* trace off      -- do not trace prolog
starting worker asynchronously ...
about to wait for worker to end ...
do: working for 0.45 secs
do: ending work
waiting is over

setting: .TraceObject~option="Standard"
displaying 24 collected trace objects:
[T1 I2 G L0  ]      >I> Method "INIT" with scope "TEST"
[T1 I2 G L1 *]   23 *-* expose lock  -- access attribute
[T1 I2 G L1 *]   24 *-* lock=.true   -- set default
[T1 I2 G L1 *]      >>>   "1"
[T1 I2 G L0  ]      <I< Method "INIT" with scope "TEST"
[T1 I3 U L0  ]      >I> Method "WAIT" with scope "TEST"
[T1 I3 U L0  ]   27 *-* expose lock  -- access attribute
[T1 I3 U L0  ]   28 *-* guard on when lock=.false
[T1 I3 U L1 *]      >K>   "WHEN" => "0"
[T2 I4 G L0  ]      >I> Method "DO" with scope "TEST"
[T2 I4 G L1 *]   31 *-* expose lock  -- access attribute
[T2 I4 G L1 *]   32 *-* t=random(1,999)/1000
[T2 I4 G L1 *]      >>>   "0.45"
[T2 I4 G L1 *]   33 *-* say "do: working for" t "secs"
[T2 I4 G L1 *]      >>>   "do: working for 0.45 secs"
[T2 I4 G L1 *]   34 *-* call sysSleep t
[T2 I4 G L1 *]      >>>   "0"
[T2 I4 G L1 *]   35 *-* say "do: ending work"
[T2 I4 G L1 *]      >>>   "do: ending work"
[T2 I4 G L1 *]   36 *-* lock=.false  -- release lock
[T2 I4 G L1 *]      >>>   "0"
[T2 I4 G L0  ]      <I< Method "DO" with scope "TEST"
[T1 I3 U L1 *]      >K>   "WHEN" => "1"
[T1 I3 U L0  ]      <I< Method "WAIT" with scope "TEST"
```

4 Conclusion

This article introduced the runtime monitoring abilities of the dynamic programming language REXX with its TRACE keyword instruction and its built-in TRACE() function, which has been available in its open-source successor "open object Rexx (ooRexx)" for runtime monitoring. It discussed vital runtime information to allow identifying the threads being used, the invocations to follow, the instances to trace in the context of guarded or unguarded methods, the possession of scope and instance-based attribute booleans and the state of their guard locks, and in complex deployments, the identification of Rexx interpreter instances used at runtime.

To make runtime monitoring and analysis of runtime context more powerful and efficient, a *TraceObject* class was devised and implemented. Each trace information will cause a numbered and time-stamped trace object to be created that can be appended to an optional collector object that needs to understand an append message. All trace information gets stored under predefined string indexes in each trace object that can be fetched from it for further inspection and processing, either at runtime or after creating a trace log off it in post-process scenarios.

A critical feature of the *TraceObject* class is the ease of formatting the trace line with different extended trace information by simply setting the value of its class attribute named *option* to one of *Normal*, *Thread*, *Standard*, or *Full*, which takes effect immediately. It is, therefore, possible to change the formatting at runtime. It is also possible to collect the trace objects, create a trace log, and display the trace objects with different formatting by changing the *TraceObject* class attribute *option* accordingly.

References

[1] Michael Cowlishaw. 1990. *The REXX language: a practical approach to programming.* Prentice-Hall, Inc. https://dl.acm.org/doi/abs/10.5555/93748

[2] Rony G. Flatscher. [n. d.]. *Introduction to Programming with ooRexx and BSF4ooRexx 2.* https://sourceforge.net/projects/bsf4oorexx/

[3] Rony G. Flatscher. 2024. Debugging Multithreaded ooRexx Programs. In *35th Rexx Language Symposium.* Brisbane, Australia and online. https://www.rexxla.org/presentations/2024/202403-03_debugging_MT_programming.pdf

[4] Rony G Flatscher and Till Winkler. 2024. Employing the Message Paradigm to Ease Learning Object-oriented Concepts and Programming. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO).* IEEE, 1244–1249.

[5] Jeff Huang and Charles Zhang. 2011. An efficient static trace simplification technique for debugging concurrent programs. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18.* Springer, 163–179.

[6] IBM. 2004. *Software withdrawal and service discontinuance: IBM Object REXX.* https://www.ibm.com/docs/en/announcements/archive/ENUSZP04-0453

[7] Rene Jansen. 2011. *7 Reasons that Rexx Still Matters.* https://smartbear.com/blog/7-reasons-that-rexx-still-matters/

[8] orexx, abaginski, aresti, aseik, bjskelly, jfaucher, manidu, perolovjonsson, rexx, rexxcoder, rvjansen, walterpachl, wb4alm, and eduijs. [n. d.]. *BSF4ooRexx.* https://sourceforge.net/projects/bsf4oorexx/

[9] RexxLA. 2022. *ooRexx Documentation 5.0.0: Reference* (2022.12.22 ed.). Rexx Language Association. https://sourceforge.net/projects/oorexx/files/oorexx-docs/5.0.0/rexxref.pdf

Received 2024-06-24; accepted 2024-07-24