Identifying Potential Deadlocked Instructions in a Multi-threaded ooRexx Program

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

In dynamic runtime environments, like ooRexx applications, guard-locked multi-threaded applications may be deadlocked for many reasons. Breaking deadlocked code may yield the location of a deadlocked instruction but may not allow for identifying other deadlocked threads such that the causes may be difficult or even impossible to locate. With the introduction of collectible *TraceObject* instances in ooRexx 5.1, a new infrastructure becomes available that can be put to work for identifying guard lock-related deadlocks on any number of threads. This article introduces the principles for creating *TraceObject* trace logs that can be externalized (e.g., encoded as JSON or XML text files) and internalized later as the originally ordered collection of trace objects for analyzing and processing purposes. The ooRexx guard lock-based multi-threading rules can then be used to develop an algorithm to identify all deadlocked program instructions "post immobilization," analyzing the *TraceObject* trace logs and supplying information about the affected objects and threads.

CCS Concepts

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

Keywords

Runtime Verification, Multi-threaded, ooRexx, TraceObject, Trace Log, Deadlock, Guard Lock

ACM Reference Format:
Rony G. Flatscher and Till Winkler. 2024. Identifying Potential Deadlocked Instructions in a Multi-threaded ooRexx Program. 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/10.1145/3679008.3685546

1 Introduction

ooRexx (an acronym for open object Rexx) is an open-source, dynamic, and dynamically typed programming language that was originally created by IBM to add object-oriented programming concepts to the then-popular IBM REXX scripting language [2]. Influenced by SmallTalk, ooRexx implements the message paradigm that conceptually reduces programming with objects to communicating with them by sending them messages [4][5]. Conceptually, the receiving objects then will look for a method by the same name and invoke it, supplying any arguments received with the message and returning any result the method supplies to the caller. To allow for messages, ooRexx supports message expressions that consist of the receiver, the message operator ~ (tilde), and the message name. ooRexx supports method chaining and method cascading (using two tildes, ~~, as the message operator) [7].

ooRexx allows for multi-threaded execution of methods that can be either initiated by sending messages asynchronously or with the keyword instruction REPLY. Using the the keyword instruction will return from a method invocation and reschedule the remaining instructions to be dispatched on a different thread. Access to methods of a specific class (scope) is controlled by a guard lock, which, by default, is used to execute methods of a specific class (scope) sequentially. The guard state of a method can be changed from guarded to unguarded and vice versa, either by defining the method guarded or unguarded, but also at runtime with the GUARD keyword statement that may carry a condition using object variables (attributes) as control variables [3].

Running multi-threaded ooRexx programs may arrive at states that cause deadlocks like in any other programming language. As the synchronization is carried out by the ooRexx runtime system and the dependency of holding the guard lock of methods of a specific class (scope) can be influenced by programmers, it may not be easy to find the locations that are involved in a deadlock. As with other programming languages, for instance Java, approaches for detecting deadlocks through runtime analysis are required [1].

This article first introduces in section 2 the ooRexx guard lock-based access of methods and explains the guard lock rules in a little bit more detail to become able to assess and understand potential deadlocks. Using the REPLY keyword statement, a simple deadlock example will be depicted. Section 3 briefly introduces the new *TraceObject* class in ooRexx 5.1.0 that, for the first time, makes all relevant runtime (monitor) related information available. In section 3.1, *TraceObject*'s built-in ability to monitor and collect trace objects at runtime is exploited to create trace logs that can be analyzed and annotated for deadlock locations. Section 3.2 suggests running programs that deadlock, getting at their trace log, and analyzing it to add runtime-related annotations to determine the blocked instructions. Finally, section 4 concludes this article.

2 Multi-threading in ooRexx

Multi-threading in ooRexx can be kicked off by sending messages asynchronously or by employing the REPLY keyword statement in a method.



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/10.1145/3679008.3685546

Messages in ooRexx can be dispatched (sent) synchronously on the same thread and asynchronous on a separate thread, spinning off multi-threading execution. The ooRexx *Message* class offers the methods *"send"* for synchronous and *"start"* for asynchronous dispatch. The message object can be used to inspect the current state of execution of an asynchronous message, whether it has been completed, or whether an error condition got raised during execution, fetching the result, or simply waiting (synchronizing) until an asynchronous dispatched method concluded its work.

Alternatively, the root class *Object*'s *"start"* method can be used to dispatch a message asynchronously, returning the created message object to allow for monitoring and interacting with it.

When interacting with an ooRexx object from different threads, methods of the same class ("scope") are guarded by default, such that only the guarded method holding the guard lock gets executed, thereby serializing the execution among all guarded methods of the same class ("scope"). Such serialization implies sequential access to attributes (object variables) by different methods of the same class as well. This simple but effective mechanism ensures that by default, multi-threaded execution of guarded methods of the same class (scope) have their object variables (attributes) protected from concurrent updates that could otherwise jeopardize their integrity.

Methods can be explicitly defined as unguarded, such that their execution on a different thread will always be possible, even concurrently to guard lock controlled, i.e., guarded methods.

The ooRexx REPLY keyword instruction available in methods can be used to return control to the sender, optionally returning a result such that the sender can continue to run on the same thread. After the executed REPLY keyword statement, the remaining method instructions will be rescheduled for execution on a new thread. If that method's state was unguarded, then it may have to wait on the guard lock if trying to acquire it, as, in the meantime, another guarded method in the same class on a different thread could successfully acquire the guard lock.

The ooRexx GUARD keyword statement enables a programmer to change the state of a method at runtime with the GUARD OFF instruction from guarded to unguarded and with the GUARD ON instruction from unguarded to guarded. The GUARD keyword instruction can only be executed if possessing the guard lock. It is possible to add to the GUARD keyword instruction a condition that must be met before being allowed to be carried out, using attributes as control variables in that condition. A GUARD instruction waiting on a condition to turn true will be reevaluated each time any of the attributes evaluated in the condition change their value from a method running in another thread.

Possible causes for guard lock-related deadlocks among methods of a specific class can be:

- an unguarded method cannot obtain the guard lock for running a GUARD instruction because another method does not release it,
- a long-running (e.g., forever loop) guarded method possessing the guard lock does not yield it with GUARD OFF such that other guarded methods of the same class waiting on the guard lock on different threads do not get it,
- the condition with "GUARD ON|OFF WHEN condition" does not turn true because no other method changes any of the attribute values used in "condition" such that the condition gets re-evaluated.

Listing 1 depicts a very simple example of a deadlocking ooRexx program, originally devised by Jean Louis Faucher for his experimental ooRexx interpreter named "executor" [6]. In this example, the keyword instruction REPLY in line 10 causes a deadlock.

Listing 1: "deadlock.rex" ooRexx program

```
1  o = .Clz~new
2  call syssleep 0.5
3  o~m2 -- wake-up m1
4  say "done."
5
6  ::class Clz
7  ::method init   -- guarded method
8      expose a    -- exclusive access
9      a = 0
10     reply
11
12     self~m1     -- still guarded
13
14 ::method m1     -- guarded method
15     expose a    -- exclusive access
16     a = 1
17     guard off   -- unguard m1 method
18     say "m1 method: unguarded"
19     guard on when a <> 1
20     say "m1 method: guarded"
21
22 ::method m2     -- guarded method
23     expose a    -- exclusive access
24     a = 2       -- change attribute
```

Using the standard runtime monitoring capabilities of ooRexx employing the keyword instruction "TRACE results" yields the output shown in Listing 2.

Listing 2: TRACE output of "deadlock.rex" (Listing 1)

```
    >I> Routine "deadlock.rex"
  1 *-* o = .Clz~new
    >I> Method "INIT" with scope "CLZ"
  8 *-* expose a   -- exclusive access
  9 *-* a = 0
    >>>   "0"
 10 *-* reply
    <I< Method "INIT" with scope "CLZ"
    >>>   "a CLZ"
  2 *-* call syssleep 0.5
    >I> Method "INIT" with scope "CLZ"
 12 *-* self~m1    -- still guarded
    >I> Method "M1" with scope "CLZ"
 15 *-* expose a   -- exclusive access
 16 *-* a = 1
    >>>   "1"
 17 *-* guard off  -- unguard m1 method
 18 *-* say "m1 method: unguarded"
    >>>   "m1 method: unguarded"
m1 method: unguarded
 19 *-* guard on when a <> 1
    >K>   "WHEN" => "0"
    >>>   "0"
  3 *-* o~m2 -- wake-up m1
    >I> Method "M2" with scope "CLZ"
```

Although a relatively simple example, it is very difficult from Listing 2, if not impossible, to understand why and where the program in Listing 1 above deadlocks.

3 The TraceObject Class

Figure 1 depicts the *TraceObject* class introduced with ooRexx 5.1, which makes all runtime monitoring-related information of ooRexx programs available for each traced instruction. *TraceObject* subclasses *StringTable*, a directory-like class.

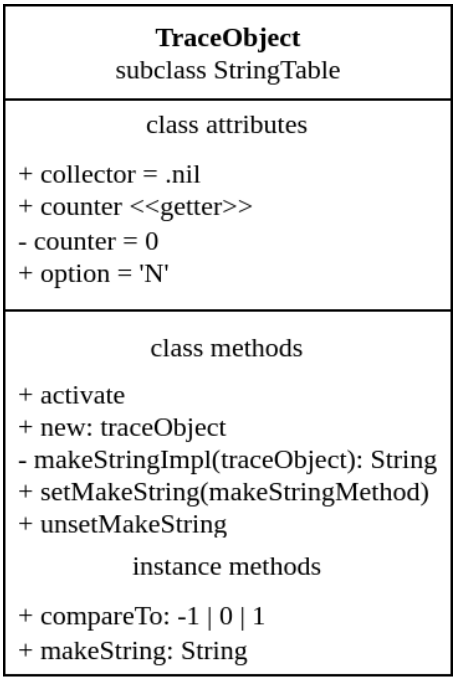


Figure 1: The TraceObject class

Each *TraceObject* instance may contain the following entries:

- `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`)

If the monitored instruction is part of a method, additional entries will get added, informing us about the object that invoked the method, the (class) scope of the method represented by its attribute (object variable) pool id, and more.

3.1 Trace Logs

The runtime monitoring for analyzing potential deadlock locations depends on the ability to create trace logs, i.e., collecting all generated trace objects during the execution of a program that becomes immobilized due to deadlocks.

Directly calling a program that deadlocks may not be a good idea, as it might be the case that the caller will get immobilized as well if the called program does not return. Listing 3 outlines a Rexx program that will be able to run a deadlocking Rexx program and collect all of its trace objects to create a trace log that can be analyzed. The program will accept the name of a Rexx program to be runtime monitored, read that program source into an array, and append the directive instruction `::OPTIONS TRACE RESULTS` to the source, which will activate runtime monitoring among the entire program. The program then dynamically creates a Routine object from this edited source that can be run anytime.

The class attribute "option" of the *TraceObject* class gets set to *"Probe"* to inhibit any output of trace lines from this moment on. Also, the class attribute *"collector"* of the *TraceObject* class gets an array assigned, which will get all created trace objects appended to it from that moment on. This array represents the trace log of any traced program as long as *TraceObject*'s *"collector"* attribute refers to that array object.

Listing 3: "runner.rex": create trace logs

```
1  parse arg fn              -- fetch file name argument
2  arr=getFileAsArray(fn)
3  -- global trace
4  arr~append("::OPTIONS TRACE RESUSLTS")
5  r=.routine~new(fn, arr) -- create routine
6  .traceObject~option='P' -- Probe: no trace lines
7  traceLog=.array~new   -- array collecting trace obj.
8  .traceObject~collector=traceLog  -- trace obj.
9  msg=r~start("call")      -- call asnchronously
10 do while \msg~completed, time('elapsed')<2 -- 2 secs
11   call sysSleep 0.1
12 end
13 say "msg~completed:" msg~completed -- 0 (deadlocked)
14 .traceObject~collector=.nil   -- stop collecting
15 .TraceObject~setMakeString(getMyMakeStringMethod())
16 do traceObject over traceLog
17    say traceObject
18 end
19  -- ... cut ...
```

The program in Listing 3 sends the *start* message to the *Routine* object with the message name *"call"* as an argument. The receiving *Routine* object will resolve the method *start* in its root class *Object*, which will create a *Message* object with the *Routine* object as the receiver and *call"* as the name of the message to send on a separate thread. After waiting up to two seconds, the returned message object will be used later to get state information to allow all threads to complete if the program succeeds. Collecting the trace objects gets stopped after waiting a maximum of two seconds by setting the *TraceObject* class attribute *"collector"* to *.nil*.

The array object now contains the entire trace log of running the program. Such trace logs can be used for analysis or for externalizing the contained trace objects, e.g., as XML or JSON, for documentation purposes.

Listing 4 gives an example of a possible XML rendering of a *TraceObject* instance, that includes additional information, e.g., retrieved from the trace object's *StackTrace* entry to allow for extensive post-run analysis.

Listing 4: A possible XML encoding of a TraceObject instance

```
<traceObject>
    <option>P</option>
    <number>21</number>
    <timestamp>2024-07-04T15:16:53.181000</timestamp>
    <interpreter>1</interpreter>
    <thread>3</thread>
    <invocation>4</invocation>
    <name>M1</name>
    <type>METHOD</type>
    <lineNr>19</lineNr>
    <executableId>FFFFFDF0_49D4DE5F</executableId>
    <package>deadlock.rex</package>
    <receiver>a CLZ</receiver>
    <receiverid>FFFFFDF0_49D3C37F</receiverid>
    <isGuarded>1</isGuarded>
    <currentGuardState>0</currentGuardState>
    <attributePool>1</attributePool>
    <target>a CLZ</target>
    <targetType>a CLZ</targetType>
    <targetId>FFFFFDF0_49D3C37F</targetId>
    <targetPackage>deadlock.rex</targetPackage>
    <scopeLockCount>2</scopeLockCount>
    <hasScopeLock>1</hasScopeLock>
    <isWaiting>1</isWaiting>
    <scope>CLZ</scope>
    <scopePackage>deadlock.rex</scopePackage>
    <traceline>   &amp;gt;K&amp;gt;   "WHEN" =&amp;gt; "0"</traceline>
</traceObject>
```

3.2 Analyzing Trace Logs

This section first discusses determining where deadlocks occur and how to identify and remedy the cause. The next section will define an algorithm that can automatically analyze and annotate the trace log, marking all potential deadlock occurrences, no matter how simple or complicated the affected programs' trace logs are.

3.2.1 Analyze, Identify and Eliminate Deadlocks. Using *TraceObject*'s class method *setMakeString(...)* makes it possible to define customized trace line formats. In the context of this article, the following information will be displayed in the bracketed extended trace prefix:

- `T1` represents the thread with the ID value of 1
- `I3` an invocation with the ID 3
- `G` indicates the method to be guarded
- `A1` communicates the attribute pool's ID to be 1 (all A1 occurrences represent the same instance and class scope)
- `L0` communicates the guard lock count to be 0
- `asterisk (*)` indicates who holds the guard lock

If a guarded method invokes another method, the invocation ID will increase, as will the lock number and the guard lock will be inherited. The implementation of this customized *makeString* method gets fetched and assigned in line 14 in Listing 3 above.

Issuing the command "runner.rex deadlock.rex" uses the program depicted in Listing 3 to run the deadlocking program. In Listing 3, line 14 fetches a customized *makeString* method and supplies it as an argument to *TraceObject*'s class method *setMakeString(...)*, the loop in lines 15 through 17 will output all trace objects with this customized method. The output is shown in Listing 5.

Listing 5: Output of "runner.rex deadlock.rex"

```
[T1 I1]                >I> Routine "deadlock.rex"
[T1 I1]                <I< Routine "deadlock.rex"
[T2 I2]                >I> Routine "deadlock.rex"
[T2 I2]              1 *-* o = .Clz~new
[T2 I3 G A1 L0  ]      >I> Method "INIT" with scope "CLZ"
[T2 I3 G A1 L1 *]    8 *-* expose a   -- exclusive access
[T2 I3 G A1 L1 *]    9 *-* a = 0
[T2 I3 G A1 L1 *]      >>>   "0"
[T2 I3 G A1 L1 *]   10 *-* reply
[T2 I3 G A1 L1 *]      <I< Method "INIT" with scope "CLZ"
[T2 I2]                >>>   "a CLZ"
[T2 I2]              2 *-* call syssleep 0.5
[T3 I3 G A1 L1 *]      >I> Method "INIT" with scope "CLZ"
[T3 I3 G A1 L1 *]   12 *-* self~m1    -- still guarded
[T3 I4 G A1 L1  ]      >I> Method "M1" with scope "CLZ"
[T3 I4 G A1 L2 *]   15 *-* expose a   -- exclusive access
[T3 I4 G A1 L2 *]   16 *-* a = 1
[T3 I4 G A1 L2 *]      >>>   "1"
[T3 I4 G A1 L2 *]   17 *-* guard off  -- unguard m1 method
[T3 I4 G A1 L1  ]   18 *-* say "m1 method: unguarded"
[T3 I4 G A1 L1  ]      >>>   "m1 method: unguarded"
[T3 I4 G A1 L1  ]   19 *-* guard on when a <> 1
[T3 I4 G A1 L2 *]      >K>   "WHEN" => "0"
[T2 I2]                >>>   "0"
[T2 I2]              3 *-* o~m2 -- wake-up m1
[T2 I5 G A1 L1  ]      >I> Method "M2" with scope "CLZ"
```

The first two trace lines reflect that the *Routine* object gets created on the main thread (T1) and the first invocation (I1). The next trace line indicates that on thread 2 (T2) with invocation 2 (I2), the program now executes asynchronously on a new thread. Invocation I3 represents running the constructor method INIT, which is a guarded method. The REPLY keyword instruction returns to the caller at I2, allowing it to proceed running, while the remainder of the INIT method instructions get dispatched on the new thread T3, sending itself the message M1, which will invoke in I4 method M1 on T3. After setting the attribute *a* to one, M1 uses the keyword statement GUARD OFF to change its guard state to unguarded and give up the guard lock. This causes the first evaluation of the "guard on when a <>1" to fail, as can be seen by the result "0" (false) line 24 in Listing 5 above. Subsequently, attribute *a* needs to be changed from another thread to have the "guard on when a<>1" statement is re-evaluated again. Meanwhile, in the main program in I2 on T2, the object gets the message M2 sent, causing it to invoke I5 for running method M2, which hangs as no more runtime monitoring trace objects get created.

Looking at the trace log in Listing 5 above, it becomes clear that the M2 guarded method in the A1 scope does not gain the guard lock for A1 and waits for it. At the same time, looking closer at method M1 for the A1 scope, one can see that the WHEN condition in I4 on T3 does not get re-evaluated because attribute *a* in scope A1 serves as a control variable and does not get changed. This is because, for the A1 scope in I5 on T2, the instruction is waiting on acquiring the guard lock before becoming able to change the attribute named *a*. Going up the invocation chain for the A1 scope from I4 on T3 brings us to I3 on T3, where the message M1 got sent off, revealing that the method possesses the guard lock for A1, inhibiting I5 on T2 from gaining it. One solution in this case is to

release the guard lock in I3 on T3 before sending the M1 message such that later method M2 becomes able to acquire the guard lock and subsequently becomes able to change *a* in scope A1, allowing in I4 on T3 the re-evaluation of the WHEN condition once M2 returns thereby releasing the guard lock for A1. Another possible solution, in this case, is to define method M2 as unguarded so it does not get blocked and can change attribute *a* in scope A1 in I5 on T2.

The example program in Listing 1 above is very simplistic, using simply the REPLY keyword instruction to kick off multi-threaded execution. In the real world, ooRexx multi-threading deployments would be more elaborate and cause the trace logs to be much more complicated and longer. For instance, the same classes may be used to create numerous instances that employ multi-threading via REPLY keyword statements or asynchronous messages, making it practically impossible for humans to analyze such trace logs manually at all!

3.2.2 Identify and Annotate Deadlocks. Each object (instance) wishing to invoke a guarded method from a specific class needs to acquire the guard lock governing all guarded methods of that particular class ("scope") for that particular instance. Each object (instance) has its own pool of attributes ("object variables") defined for a particular class that can be shared among all the methods defined for that particular class such that that class defines the (class) scope and allows for guarding it with a guard lock. The *TraceObject* class will maintain an identifier that is unique per object (instance) and (class) scope and stores it under the name "ATTRIBUTEPOOL".

As guard lock-related, deadlocks can only occur if race conditions occur when attempting to acquire the guard lock of a specific scope from different threads, ordering the trace log by the attribute (pool) identifier and threads brings all related trace objects together. Within each group, ordering by invocation ID and within invocation groups by the sequence the trace objects got created therein allows a simplified algorithm to be applied to identify the trace objects that are waiting on receiving the guard lock: it first needs to determine the effective guard state of a method within such a group by looking for GUARD keyword instructions and evaluating their effect. As a trace object is an instance of a *StringTable*, it is possible to annotate each trace object by adding an entry that stores the effective guard state of a method and whether it is running or waiting.

At runtime, if a method is in the effective unguarded state, it is always runnable unless it carries out a GUARD keyword statement for which it needs to acquire the guard lock. If using the GUARD keyword statement with a condition, such a method will remain in the wait state if the condition does not yield true.

At runtime, a method in the effective guarded state is running if it has acquired the guard lock and is waiting on it otherwise. If entering a guarded method from a different thread without holding the guard lock and not having any other trace objects for that particular invocation available, then that instruction is waiting for the guard lock. After storing the annotation's effective guard state and wait state, the trace log needs to be reordered by the entry "NUMBER", which is the serial number of the trace object in the trace log.

Listing 6 depicts the trace lines after annotating the same trace log that gets created when running "runner.rex deadlock.rex" if using a tailored *makeString* method that shows the effective guard state of a method in lowercase ('u' for unguarded, 'g' for guarded) if it differs from the statically defined guard state. In addition, a capital 'W' is displayed with instructions waiting on the guard lock. This makes it easy to spot the instructions waiting on the guard lock at a specific point at runtime and their context.

Listing 6: "runner.rex deadlock.rex" with annotation output

```
[T1 I1]                    >I> Routine "deadlock.rex"
[T1 I1]                    <I< Routine "deadlock.rex"
[T2 I2]                    >I> Routine "deadlock.rex"
[T2 I2]                  1 *-* o = .Clz~new
[T2 I3 G   A1 L0    ]    >I> Method "INIT" with scope "C
[T2 I3 G   A1 L1 *  ]    8 *-* expose a   -- exclusive acc
[T2 I3 G   A1 L1 *  ]    9 *-* a = 0
[T2 I3 G   A1 L1 *  ]      >>>   "0"
[T2 I3 G   A1 L1 *  ]   10 *-* reply
[T2 I3 G   A1 L1 *  ]      <I< Method "INIT" with scope "C
[T2 I2]                    >>>   "a CLZ"
[T2 I2]                  2 *-* call syssleep 0.5
[T3 I3 G   A1 L1 *  ]      >I> Method "INIT" with scope "C
[T3 I3 G   A1 L1 *  ]   12 *-* self~m1    -- still guarded
[T3 I4 G   A1 L1    ]      >I> Method "M1" with scope "CLZ
[T3 I4 G   A1 L2 *  ]   15 *-* expose a   -- exclusive acc
[T3 I4 G   A1 L2 *  ]   16 *-* a = 1
[T3 I4 G   A1 L2 *  ]      >>>   "1"
[T3 I4 G   A1 L2 *  ]   17 *-* guard off  -- unguard m1 me
[T3 I4 G u A1 L1    ]   18 *-* say "m1 method: unguarded"
[T3 I4 G u A1 L1    ]      >>>   "m1 method: unguarded"
[T3 I4 G u A1 L1    ]   19 *-* guard on when a <> 1
[T3 I4 G u A1 L2 * W]      >K>   "WHEN" => "0"
[T2 I2]                    >>>   "0"
[T2 I2]                  3 *-* o~m2 -- wake-up m1
[T2 I5 G   A1 L1   W]      >I> Method "M2" with scope "CLZ
```

In the case of "deadlock.rex," no further trace objects get created as the program is deadlocked because method M2 is waiting for the guard lock and can therefore not change attribute *a* to allow the "guard on when a <> 1" condition in M1 to be re-evaluated. Going up the invocation chain to the point where the message M1 gets sent in method INIT, one can see that the guard lock is not released before the message gets sent. Releasing the guard lock before sending the M1 message will allow M2 to acquire the guard lock later, as no one will hold it then.

In Listing 7, the guard lock is released with the "guard off" statement before the M1 message gets sent, which solves the deadlock and is the only change to the program shown in Listing 1.

Listing 7: "solved.rex" ooRexx program

```
1  o = .Clz~new
2  call syssleep 0.5
3  o~m2 -- wake-up m1
4  say "done."
5
6  ::class Clz
7  ::method init   -- guarded method
8      expose a    -- exclusive access
9      a = 0
10     reply
11     guard off   -- free guard lock
12     self~m1     -- now unguarded
13 ::method m1     -- guarded method
14     expose a    -- exclusive access
15     a = 1
16     guard off   -- unguard m1 method
17     say "m1 method: unguarded"
```

```
18      guard on when a <> 1
19      say "m1 method: guarded"
20  ::method m2     -- guarded method
21      expose a    -- exclusive access
22      a = 2       -- change attribute
```

Listing 8 shows the trace log output when running the program in Listing 7 via the program in Listing 3. It is interesting to note that at one particular point in time one instruction is waiting on the guard lock with a guard condition. Later, when method M2 executes, the change of the attribute *a* will cause the guard condition to be re-evaluated and to succeed. The final three trace lines with the trace prefix "<I<" indicate the return from three remaining invocations having taken place on the threads T2 and T3.

Listing 8: "runner.rex solved.rex" with annotation output

```
[T1 I1]                    >I> Routine "solved.rex"
[T1 I1]                    <I< Routine "solved.rex"
[T2 I2]                    >I> Routine "solved.rex"
[T2 I2]                  1 *-* o = .Clz~new
[T2 I3 G   A1 L0    ]     >I> Method "INIT" with scope "C
[T2 I3 G   A1 L1 *  ]   8 *-* expose a   -- exclusive acc
[T2 I3 G   A1 L1 *  ]   9 *-* a = 0
[T2 I3 G   A1 L1 *  ]     >>>   "0"
[T2 I3 G   A1 L1 *  ]  10 *-* reply
[T2 I3 G   A1 L1 *  ]     <I< Method "INIT" with scope "C
[T2 I2]                   >>>   "a CLZ"
[T2 I2]                  2 *-* call syssleep 0.5
[T3 I3 G   A1 L1 *  ]     >I> Method "INIT" with scope "C
[T3 I3 G   A1 L1 *  ]  11 *-* guard off  -- free guard lo
[T3 I3 G u A1 L0    ]  12 *-* self~m1    -- now unguarded
[T3 I4 G   A1 L0    ]     >I> Method "M1" with scope "CLZ
[T3 I4 G   A1 L1 *  ]  15 *-* expose a   -- exclusive acc
[T3 I4 G   A1 L1 *  ]  16 *-* a = 1
[T3 I4 G   A1 L1 *  ]     >>>   "1"
[T3 I4 G   A1 L1 *  ]  17 *-* guard off  -- unguard m1 me
[T3 I4 G u A1 L0    ]  18 *-* say "m1 method: unguarded"
[T3 I4 G u A1 L0    ]     >>>   "m1 method: unguarded"
[T3 I4 G u A1 L0    ]  19 *-* guard on when a <> 1
[T3 I4 G u A1 L1 * W]     >K>   "WHEN" => "0"
[T2 I2]                   >>>   "0"
[T2 I2]                  3 *-* o~m2 -- wake-up m1
[T2 I5 G   A1 L0    ]     >I> Method "M2" with scope "CLZ
[T2 I5 G   A1 L1 *  ]  23 *-* expose a   -- exclusive acc
[T2 I5 G   A1 L1 *  ]  24 *-* a = 2      -- change attrib
[T2 I5 G   A1 L1 *  ]     >>>   "2"
[T2 I5 G   A1 L0    ]     <I< Method "M2" with scope "CLZ
[T2 I2]                  4 *-* say "done."
[T2 I2]                   >>>   "done."
[T3 I4 G u A1 L1 *  ]     >K>   "WHEN" => "1"
[T3 I4 G   A1 L1 *  ]  20 *-* say "m1 method: guarded"
[T3 I4 G   A1 L1 *  ]     >>>   "m1 method: guarded"
[T2 I2]                   <I< Routine "solved.rex"
[T3 I4 G   A1 L0    ]     <I< Method "M1" with scope "CLZ
[T3 I3 G u A1 L0    ]     <I< Method "INIT" with scope "C
```

4 Conclusion

This article introduced the guard lock-related rules of multi-threaded ooRexx programs and possible reasons for deadlocks. The program in Listing 1 is a minimal Rexx program that exercises a deadlock, which may not be obvious. With the help of the *TraceObject* class that was introduced with ooRexx 5.1, it finally becomes possible to get the runtime monitor information needed to understand the state of ooRexx programs in each instruction when multi-threading is exercised.

The *TraceObject* class allows one to collect the generated trace objects to analyze the state changes manually and infer the locations where deadlocks occur. Once the deadlocked instructions are identified, it becomes possible to discover the underlying reasons so that the deadlocks can be avoided.

The trace logs must be analyzed programmatically to identify the deadlocked instructions in complex and massively multi-threaded programs. Knowing the guard lock-related rules, it is relatively easy to develop a simple algorithm that can annotate the trace logs with the effective guard states of methods for specific attribute pools and the blocked instructions waiting on the guard lock.

This should allow for debugging multi-threaded related deadlocks in complex deployment scenarios like using ooRexx in the context of JavaFX FXML-defined GUIs that get controlled by ooRexx programs.

The experimental findings will be used to create a tool that makes it easy for ooRexx programmers to analyze any of their programs, exploiting the functionality offered by the new *TraceObject* class in ooRexx 5.1.

References

[1] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. 2006. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. 41–50.

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

[3] Rony G. Flatscher. 2022. *Procedural and Object-oriented Programming: Debugging, Multithreading, Security Manager, Off the Records.* https://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/090_ooRexx_advanced_V05.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] Rene Jansen. 2011. *7 Reasons that Rexx Still Matters.* https://smartbear.com/blog/7-reasons-that-rexx-still-matters/

[6] jlfaucher. [n. d.]. *executor5.* https://github.com/jlfaucher/executor5

[7] 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