Base Language - Feature #4347

add runtime support for STOP-AFTER block option

10/15/2019 09:28 AM - Constantin Asofiei

Status:	Review	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Eduard Soltan	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

Related issues:

Related to Base Language - Feature #3751: implement support for OO 4GL and st...	**Closed**
Related to Base Language - Feature #4373: finish core OO 4GL support	**New**

History

#1 - 10/15/2019 09:28 AM - Constantin Asofiei

The STOP-AFTER option requires runtime - see the BlockManager.stopAfter stubs.

#2 - 10/15/2019 09:29 AM - Constantin Asofiei

- Related to Feature #3751: implement support for OO 4GL and structured error handling added

#3 - 10/30/2019 09:55 AM - Greg Shah

- Related to Feature #4373: finish core OO 4GL support added

#4 - 05/31/2022 10:14 AM - Constantin Asofiei

We need tests to see what happens if there are nested blocks with STOP-AFTER - as I think the STOP should target the block with the STOP-AFTER clause which expired. And this needs to cleanup any other nested STOP-AFTER stopper threads.

#5 - 05/31/2022 10:19 AM - Greg Shah

I would like to avoid having a thread for each STOP-AFTER block. I think we need to sort the timeouts and process them all on 1 thread.

#6 - 05/31/2022 10:22 AM - Constantin Asofiei

Greg Shah wrote:

> I would like to avoid having a thread for each STOP-AFTER block. I think we need to sort the timeouts and process them all on 1 thread.

OK, makes sense, I think it can be done, we just need to figure out the cleanup (when block terminates, cleanup all nested blocks when targeting an outer block, etc).

#8 - 04/22/2024 03:12 PM - Greg Shah

- Assignee set to Eduard Soltan

#9 - 04/22/2024 03:12 PM - Greg Shah

- Status changed from New to WIP

#10 - 04/24/2024 04:19 AM - Alexandru Lungu

Eduard and I had a chat yesterday on this task an I took some "thinking" time on my own on this matter:

#4347-6 is not a trivial thing from my POV - in fact, it is the hardest part of this task. I think the 4GL statements are atomic, so in 4GL the functionality is something like:

```
do stop after 3:
// if 3 seconds already passed, do stop.
message "a".
// if 3 seconds already passed, do stop.
message "b".
// if 3 seconds already passed, do stop.
// ...
end.
```

To replicate this in Java, we need all **converted** statements to check if there was a time-out or not, so it can raise stop. To do that, we can either:

- Cherry-pick some places in FWD run-time: BlockManager is the obvious one, but there may be others like AbstractQuery.next (converted statement) or BDT.assign (converted statement), etc.
- Use AOP and annotate the methods used exclusively from the conversion side.

No need to use parallel thread. For each session, we can keep a stack of deadlines and the tightest deadline (current-time + after clause). For the hotspots we choose, we need to check if the tightest deadline was passed, so we raise the STOP condition. We can do that in BlockManager for each session context.

Either way, we still need to handle, as Constantin mentioned:

- Nested blocks with STOP AFTER. I don't think this is necessarily hard - if a certain time-out occurred, simply raise STOP, wherever the execution stands (in a top-block or a nested-block).
- Edge cases:
  - Database (schema or session) triggers.
  - UI statements including WAIT-FOR or UI triggers.

Most important from my POV:

- What happens when a long running IO operations happens (including database interaction). Is this the case in #8573?
- If so, we can't cancel a DB query AFAIK (or maybe we can with some specific JDBC settings). How is 4GL dealing with this? Even if 4GL is waiting for the query to end, in FWD we don't have cursor walking, but special queries to prefetch data (like ProgressiveResults), which are not that granular.

My end-game here: if the urgency of #4347 is due to #8573 and #8573 is timing out due to a long running query, maybe we should optimize that first, right?
If we need to handle this **now**, I suggest doing some detective work for the edge cases and IO operations. After, we can deliver a more relaxed implementation where only BlockManager is honoring the STOP AFTER.

#11 - 04/24/2024 04:30 AM - Constantin Asofiei

Alexandru, the STOP AFTER needs to use a similar mechanism we have for CTRL-C support (which actually raises a STOP condition in ChUI). This will interrupt the target Conversation thread once the timeout exceeded and it will perform a rollback/rollup until the targeted DO STOP AFTER block is reached, via a STOP condition.

If you run this test in ChuI:

```
do on quit undo, leave:
   do on stop undo, leave:
     do stop-after 1:
       pause 3.
     end.
     message "no stop".
  end.
  message "no quit".
end.

message "done".
```

you will see that a STOP condition is being raised for both CTRL-C and the STOP-AFTER timeout.

#12 - 04/24/2024 08:34 AM - Greg Shah

To replicate this in Java, we need all **converted** statements to check if there was a time-out or not, so it can raise stop. To do that, we can either:

- Cherry-pick some places in FWD run-time: BlockManager is the obvious one, but there may be others like AbstractQuery.next (converted statement) or BDT.assign (converted statement), etc.
- Use AOP and annotate the methods used exclusively from the conversion side.

I disagree here. We don't want to do this. It will make a mess of the code and will also add new processing that is a performance drag.

No need to use parallel thread. For each session, we can keep a stack of deadlines and the tightest deadline (current-time + after clause).

Actually, a timer thread is exactly what is needed. And as Constantin notes, we have the interruption mechanism already available to generate the STOP. This will work for all cases (base lang, UI, database, processing on the client or the server). See Control.interrupt().

This timer thread can:

- Be a single thread for all sessions. In this case, we must save some context local state to identify the session to be interrupted along with the scheduled deadline. OR
- Be a single thread per session that handles all timers for that context. In this case, the thread itself would have the context needed to know which session is being interrupted.

  - Database (schema or session) triggers.
  - UI statements including WAIT-FOR or UI triggers.

Most important from my POV:

- What happens when a long running IO operations happens (including database interaction). Is this the case in #8573?
- If so, we can't cancel a DB query AFAIK (or maybe we can with some specific JDBC settings). How is 4GL dealing with this? Even if 4GL is waiting for the query to end, in FWD we don't have cursor walking, but special queries to prefetch data (like ProgressiveResults), which are not that granular.

The Control.interrupt() works for all of these cases.

#13 - 05/13/2024 09:56 AM - Eduard Soltan

- Status changed from WIP to Review

Committed on 4347a, revision 15204.

#14 - 05/13/2024 03:04 PM - Eric Faulhaber

Greg or Constantin: please review.

#15 - 05/14/2024 02:16 AM - Constantin Asofiei

Review for 4347a rev 15204:

- we need a SessionListener to clean up the StopAfterTimer.stopAfterTimerOuts map. Currently this is not done.
- stopAfterTimerOuts needs to be synchronized
- what happens if there are nested blocks with STOP-AFTER and i.e. the outer block has 1 second and the inner blocks 10 seconds?
- two or more timers can start executing 'in parallel' - cancel is not a problem, but:
  - the linked list for the Task may be problematic.
  - calling Control.blockInterrupt(); twice, in parallel, from two timers is possible. I think we need to ensure we call it only once.
- there are some formatting issues with StopAfterTimer - new line between field definitions, after extends - please take a look
- I think if we need to have a way to disable STOP-AFTER, via a directory configuration - the reason is if the STOP-AFTER values are tuned for OE performance, in FWD we may get into canceling tasks which should not be canceled.

#16 - 05/14/2024 02:49 AM - Eduard Soltan

Constantin Asofiei wrote:

> Review for 4347a rev 15204:
>
> - what happens if there are nested blocks with STOP-AFTER and i.e. the outer block has 1 second and the inner blocks 10 seconds?

In that case it is iterating over nested blocks using nextTask, and canceling the TimerTasks submitted to the timer corresponding to nested blocks.

- two or more timers can start executing 'in parallel' - cancel is not a problem, but:
  - the linked list for the Task may be problematic.
  - calling Control.blockInterrupt(); twice, in parallel, from two timers is possible. I think we need to ensure we call it only once.

Isn't this something that wanted to be avoided? To have separate timer for each block.

#17 - 05/14/2024 03:12 AM - Constantin Asofiei

Eduard Soltan wrote:

> Constantin Asofiei wrote:
>
> > Review for 4347a rev 15204:
> >
> > - what happens if there are nested blocks with STOP-AFTER and i.e. the outer block has 1 second and the inner blocks 10 seconds?
>
> In that case it is iterating over nested blocks using nextTask, and canceling the TimerTasks submitted to the timer corresponding to nested blocks.

Thanks, I see it now.

> > - two or more timers can start executing 'in parallel' - cancel is not a problem, but:
> >   - the linked list for the Task may be problematic.
> >   - calling Control.blockInterrupt(); twice, in parallel, from two timers is possible. I think we need to ensure we call it only once.
>
> Isn't this something that wanted to be avoided? To have separate timer for each block.

Ah, you are right, there is a single thread where the tasks are posted. So there is no concurrency here. But, can anything wrong happen if you fire two tasks in a row, which ends up calling Control.blockInterrupt almost one after the other? The point here is to look for race conditions.

#18 - 05/14/2024 06:20 AM - Eduard Soltan

Constantin Asofiei wrote:

> Ah, you are right, there is a single thread where the tasks are posted. So there is no concurrency here. But, can anything wrong happen if you fire two tasks in a row, which ends up calling Control.blockInterrupt almost one after the other? The point here is to look for race conditions.

I guess such a condition could happen.
I am thinking about adding a lock in a Session implementation, and to allow execution of the run method of TimerTask only after lock acquiring. After acquiring the lock could be released if a on stop undo phrase is encountered or at the program termination if no such phrase is encoutered.

#19 - 05/14/2024 08:29 AM - Greg Shah

I'm reviewing the code now. After reviewing the 4GL docs, I do have some questions. I also added Marian to coordinate testcases and also he may know some of the answers.

Did you write testcases to check how OE works in these cases:

- 4GL syntax
  - "time limit" value
    - Test both literals and "complex" expressions (a variable and a calculation should be enough).
    - Is the value really required to be an integer? Can it be decimal? Can it be int64?
  - Can you have both an ON STOP phrase and a STOP-AFTER phrase on the same block?
  - Can you have catch blocks for Progress.Lang.Stop (or subclasses) in the same block as an ON STOP phrase or a STOP-AFTER phrase?
- Exceptions versus Conditions and abnormal control flow
  - The 4GL docs say that when the timeout occurs, both a STOP condition is raised and a Progress.Lang.StopAfter exception will be thrown. This (and the references to enabling the old STOP behavior using -catchStop 0) suggests they have moved to an exception-first approach with STOP where they throw Progress.Lang.Stop or one of its subclasses like Progress.Lang.StopAfter, Progress.Lang.UserInterrupt for CTRL-C and Progress.Lang.LockConflict for record lock timeouts) and probably have an implicit catch block for any blocks that have ON STOP phrases. Is that how it works?
  - Does one need to use ROUTINE-LEVEL ON ERROR UNDO, THROW. or BLOCK-LEVEL ON ERROR UNDO, THROW. to get this exception behavior or is it really present in later releases? (extra credit: which OE release brought this?)
  - Does STOP-AFTER add an implicit catch block for Progress.Lang.StopAfter to the block in which it is specified?
  - Does using STOP-AFTER provide any behavior for Progress.Lang.Stop instances that are not Progress.Lang.StopAfter?
  - Does it add an implicit ON STOP phrase in the case that exceptions are not being used (-catchStop 0)
  - Is STOP-AFTER essentially an ON STOP UNDO, LEAVE but only matching a timeout and not other kinds of STOP?
  - If you can have both an ON STOP phrase and a STOP-AFTER phrase on the same block, how do they interact?
  - If you can have catch blocks for Progress.Lang.Stop (or subclasses) in the same block as an ON STOP phrase or a STOP-AFTER phrase, how do they interact?
  - Does an ON STOP clause "catch" all Progress.Lang.Stop exceptions, no matter the subclass?
- Timers
  - Does the timer get reset at the top of every looping block that has STOP-AFTER?
    - Check this for both iterate and RETRY.
  - Does the timer get removed after the final exit from any block that has STOP-AFTER?
- Nested Blocks
  - Outer block is STOP-AFTER 3 and inner block is STOP-AFTER 8, does a timeout in the inner block properly target the outer block?
  - Outer block is STOP-AFTER 8 and inner block is STOP-AFTER 3, does a timeout in the inner block properly target the inner block?

If you've already written these, just point them out and explain the results here. If not, then we probably need these testcases to confirm how things work. Many of these things are statements or implications from the 4GL documentation. We have learned not to trust that documention, it is often incomplete and sometimes incorrect.

LE: Added a Timers section above with more questions.

#20 - 05/14/2024 09:21 AM - Greg Shah

Code Review Task Branch 4347a Revision 15204

1. Please explain why Control.blockInterrupt() adds a remote call and uses the remote form of interrupting (sending a message)? I would have expected that we would use the Control.interrupt() implementation (which has both local and remote interruption paths). As far as I know, Control.interrupt() is exactly what we need.

2. In BlockManager.stopAfter(long), please call wa.tm.getBlock() only once and save the result in a local var.

3. As noted in #4347-19, I think there is likely to be a lot of exception/condition processing that the 4GL implements. I suspect we can implement this properly in a couple of days if we have the 4GL behavior fully tested and documented. More importantly, the 4GL docs suggest that there is a difference between STOP conditions processed for different causes. This could explain why they implemented an "exception first" approach to handling these cases. With that in mind, I don't understand the current BlockManager implementation. Even beyond the exception approach, I suspect that the current implementation does not properly implement the control flow in many cases.

- I would expect that the timer would be a Finalizable
  - Reset at the iterate() or retry() of a block.
  - Removed at the finished().
- Don't we need to add processing in locations where we are catching ConditionException otherwise how do we implement the impicit LEAVE of a block in response to STOP-AFTER timeout?
- Doesn't any STOP condiiton processing need to differentiate STOP-AFTER timeouts from other STOP sources?
- In the finally blocks for BlockManager.processBody() and BlockManager.processForBody(), there is the if (reason == null && wa.tm.getBlock().hasStopAfter) conditional that removes the timer. We do need to remove the timer but I think this should be done using the Finalizable.

4. The following usage in BlockManager is expensive (in BM.stopAfter() and in the finally blocks of BM.processBody() and BM.processForBody()):

```
            SessionManager sm = SessionManager.get();
            Session session = sm.getSession();
```

For performance reasons, let's save the session in the BlockManager.WorkArea. We can do that the first time we encounter a STOP-AFTER (in BM.stopAfter()). Then we don't need to look it up ever again.

5. I would prefer if we keep a reference to the StopAfterTimer instance in the BlockDefinition. That way, all the BlockManager operations can directly reference things without calling StopAfterTimer.getStopAfterTimer() and we don't need to have the StopAfterTimer.stopAfterTimerOuts map. This is faster and easier to uderstand. It also avoids the synchronization issues.

6. Some minor things:

- If we need to keep "Control.blockInterrupt()@, we need to change the name because it is confusing. It can suggest that we are "blocking interrupts" (preventing them from occurring).
- If we need to keep Control.sendIntreruptMessage(), it should be spelled sendInterruptMessage().

#21 - 05/14/2024 09:34 AM - Greg Shah

Code Review (Part 2) Task Branch 4347a Revision 15204

7. Should we (can we?) name the Timer thread to make it clear which session it belongs to? Otherwise it is just going to be chaos.

8. I'm surprised the timer thread doesn't need a FWD context. On that thread we are calling multiple context-local iooperations. This doesn't seem right.

9. I don't undertstand the usage of Control.init() in the task's run() method. That adds the current thread to the stack which seems wrong.