Debugger Specification

A Reengineering Case Study

Microsoft Research (Redmond) 1999-2000

Authors: Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, Margus Veanes

Slides courtesy of Mike Barnett and Margus Veanes

(slightly edited by E.Börger from the slides presented to the ASM'2000 Workshop, Ascona (CH), March 20-24, 2000)

See Chapter 3.1

(Requirements Capture by Ground Models) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

The Goal: Reengineering of a Debugger

From a command line debugger (30 K loc C++) which works in a stack-based COM execution env

- controlling execution: stepping through managed code, setting breakpoints, etc
- inspecting program state: viewing threads, stack frames, variables, etc

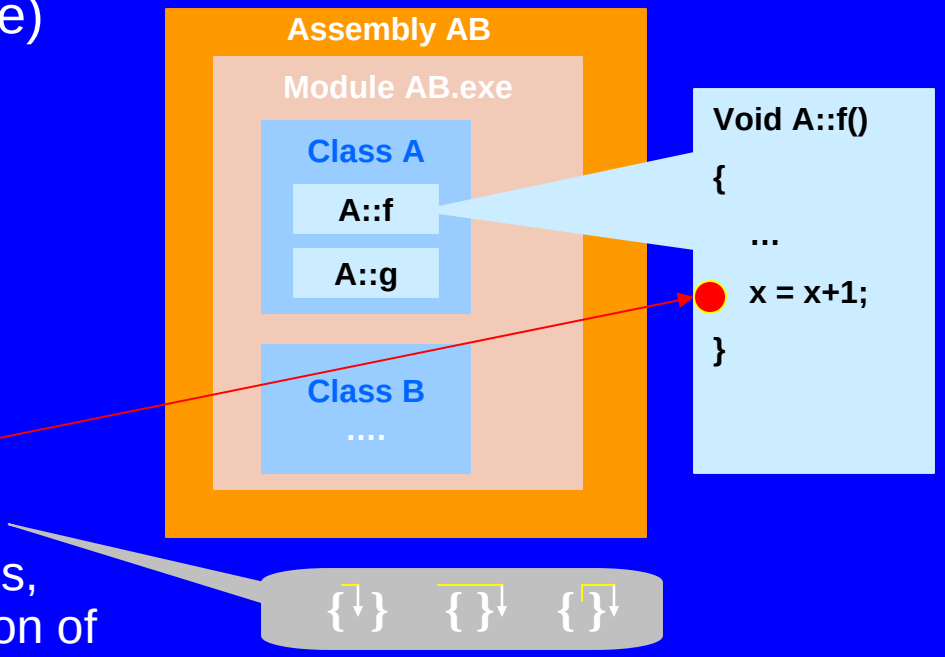
3 Debugger Models have been abstracted, forming a refinement hierarchy which specifies the APIs (Application Programming Interface) by which debugger components interact:

- **Control** Model (Level1)
- **Object** Model (Level2)
- **Ground** Model (Level3) Size: 4 K loc

The 3 models, being executable by AsmHugs, have been used for experimentation. Reflecting the reengineering task, their definition was obtained in reverse order.

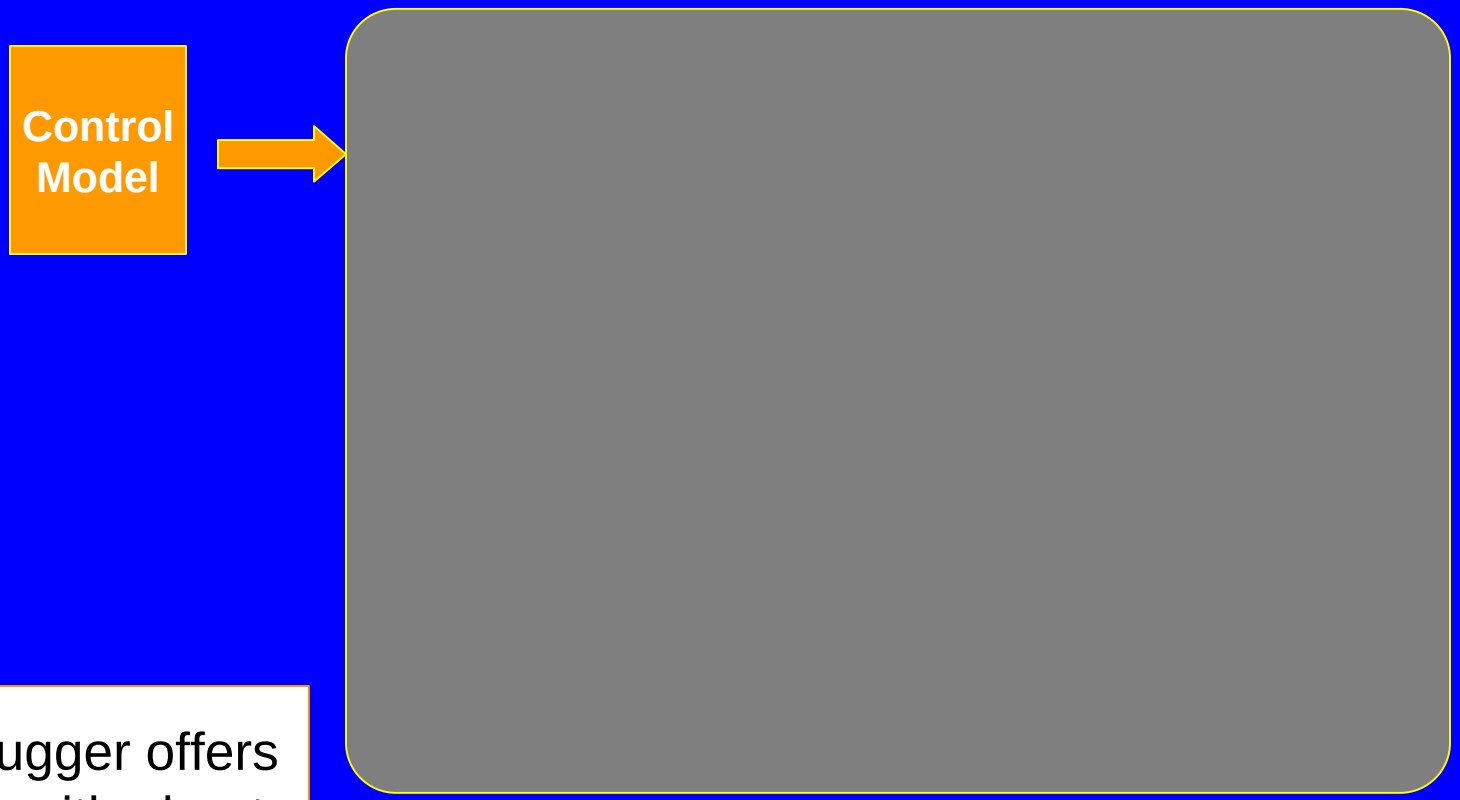
Role of the 3 models

- **Control** Model (Level1)
  - Models user modes: control moving between user, debugger and runtime system
- **Object** Model (Level2)
  - Reflects the runtime system model of static (compile time) program components
    - Assembly, module, class, function
  - Partial view of the dynamic model
    - Process, thread, chain, …
  - Refines related user commands:
    - Breakpoints, stepping cmds, execution control (inspection of run-time stack, frames)



Role of the 3 models cont.

- **Ground** Model (Level3)
  - Same core functionality as dbg
  - Faithful to dbg wrt usage of the dbg interface

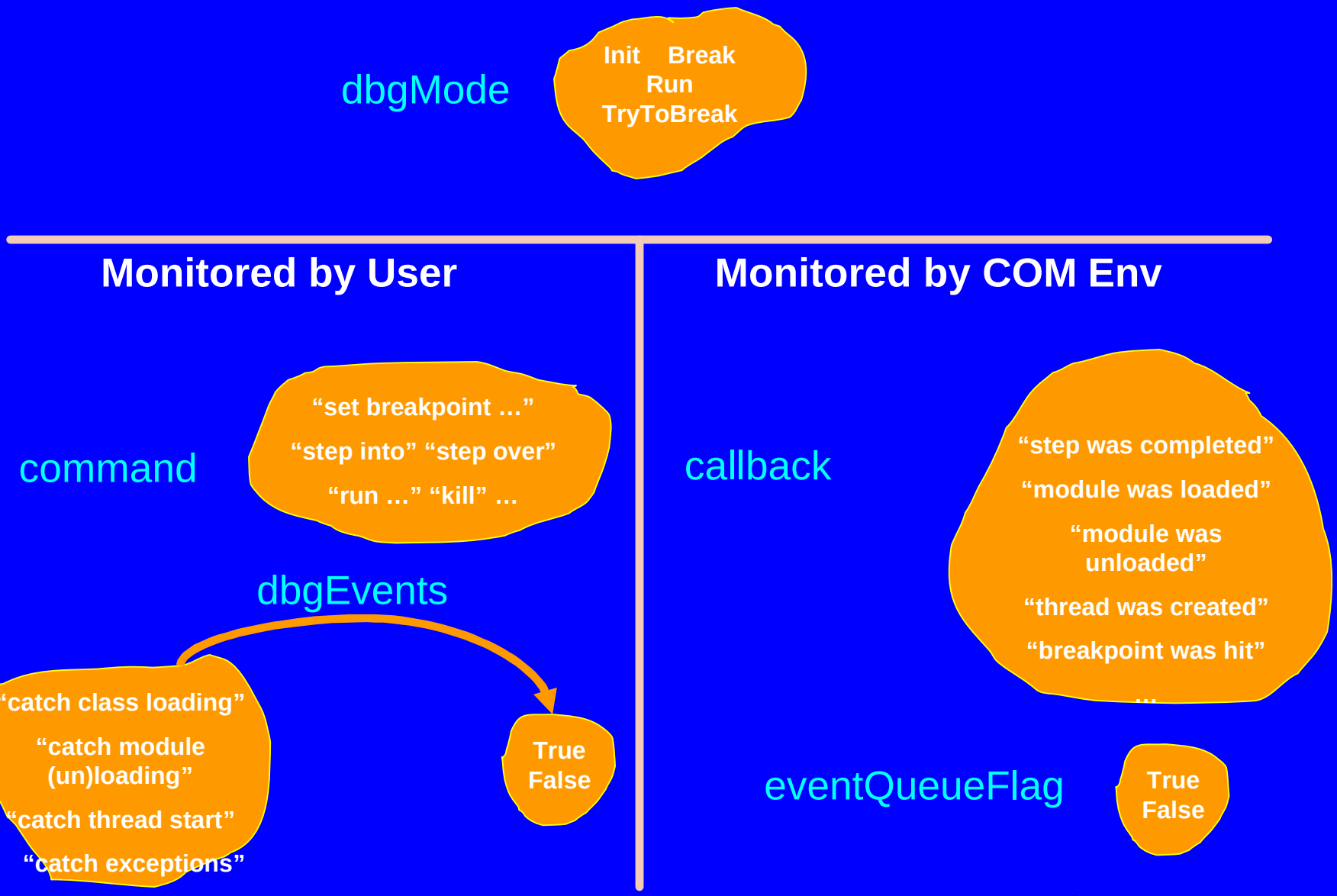


The debugger offers services with about 50 interfaces and 250 methods

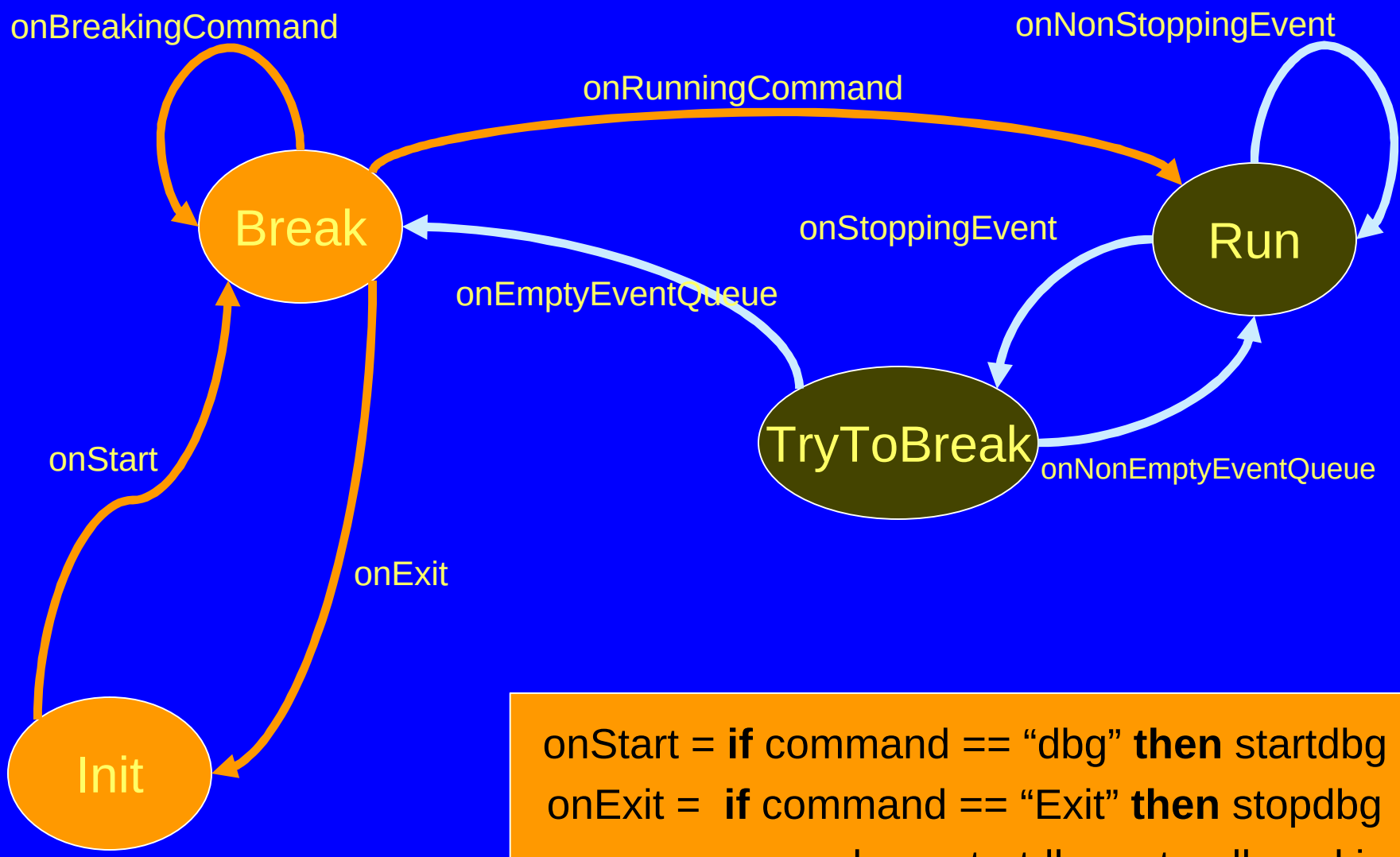
Control model: the dynamic state

- **The current control (at either user, debugger, or runtime)**
  - dbgMode: Init, Break (user), Run (runtime), TryToBreak (debugger)
- **Monitored functions set by the user**
  - command
    - current user command: a) starting/quitting debugger, b) state inspection/updates, c) passing control to runtime
  - dbgEvents
    - indicates whether stopping upon certain events is turned **On** or **Off, e.g.,** dbgEvents("classLoadEvent") = Off
- Monitored functions set by the run time system
  - callback (runtime issues at most one event at a time, only in mode Run, & then waits for an ack from the debugger)
    - current response of the run time system (callback event), e.g. callback = "hit breakpoint". When runtime has the control, debugger waits for callback before proceeding.
  - eventQueueFlag
    - A boolean indicating whether there are events still to be handled before user can get the prompt

Control model: the dynamic state



Control model rules (1st version)



onStart = **if** command == “dbg” **then** startdbg

onExit = **if** command == “Exit” **then** stopdbg

where startdbg= stopdbg=skip

Illustrating the control model spec of user commands

onRunningCommand =

if command == “c” **then** handleRunningCommand(c)

one rule for every running command c

where handleRunningCommand can be specified in a modular fashion, instructionwise, e.g. by

```
handleRunningCommand ( c ) = case ( c ) of
    RunPgm x          -> commandRunPgm x
    Attach x          -> commandAttach x
    Continue          -> command Continue
    Kill x            -> commandKill x
    StepLine x        -> commandStepLine x
    StepInstruction x -> commandStepInstruction x
    StepOut           -> commandStepOut
```

Calls from the ASM debugger (ground) model to debug services are invocations of monitored fcts

Similarly for onBreakingCommand and similarly for run time system events (with callback instead of command)

Illustrating the control model spec of command handling

```
handleBreakingCommand  = case command of
DisplayAllThreads  -> commandDisplayAllThreads
SetThread x          -> commandSetThread          Clear       -> commandClear
RefreshSource x -> commandRefreshSource x       Ignore x    -> commandIgnore x
DisplayPath        -> commandDisplayPath        SetPath x  -> commandSetPath x
DisplayBreakpoints  -> commandDisplayBreakpoints
SetBreakpoint x       -> commandSetBreakpoint x
DelBreakpoint x       -> commandDelBreakpoint x
DelAllBreakpoints     -> commandDelAllBreakpoints
DisplayEvents         -> commandDisplayEvents
CatchEvents x         -> commandCatchEvents x
Down x                -> commandDown x             Up x   -> commandUp x
PrintAllLocalVariables -> commandPrintAllLocalVariables
Print x  -> commandPrint x                    ShowLines x  -> commandShowLines x
ShowInstructions x      -> commandShowInstructions x
ShowStackTrace x      -> commandShowStackTrace x
ShowFullStackTrace   -> commandShowFullStackTrace
CommandsFromFile x -> commandCommandsFromFile x
```

Illustrating the control model spec of events

```
isNonStoppingEvent = case callback of

    CreateProcess   x     ->  True
    CreateAppDomain x     ->  True
    ExitAppDomain   x     ->  True
    LoadAssembly    x     ->  True
    UnloadAssembly  x     ->  True
    ExitThread      x     ->  True
    CreateThread    x     ->  True
    LogMessage      x     ->  True
    LoadModule      x     ->  dbgEvents(CatchModule) == Off
    UnloadModule    x     ->  dbgEvents(CatchModule) == Off
    LoadClass       x     ->  dbgEvents(CatchClass)  == Off
    UnloadClass     x     ->  dbgEvents(CatchClass)  == Off
    ExceptionCB     x     ->  not isStoppingException(x)
    otherEvents           ->  False

isStoppingEvent = not isNonStoppingEvent
```

Illustrating the control model spec of event handling (1)

handleNonStpEvent =

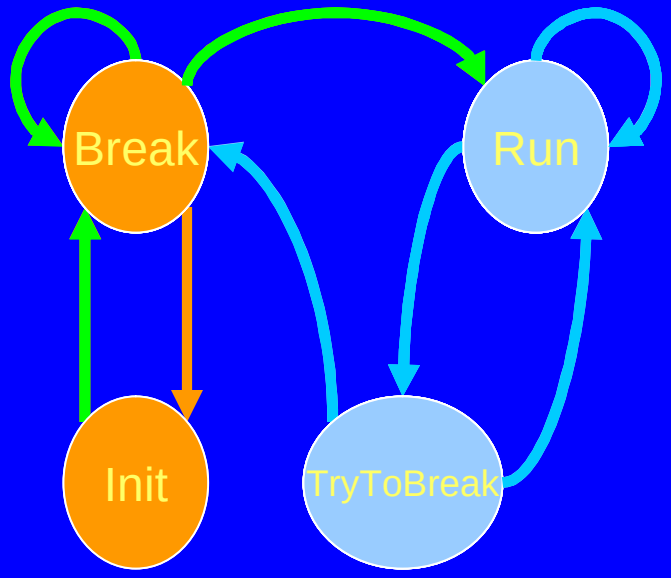
```
case callback of
        CreateProcess   x  -> callbackCreateProcess   x
        CreateAppDomain x  -> callbackCreateAppDomain x
        ExitAppDomain   x  -> callbackExitAppDomain   x
        LoadAssembly    x  -> callbackLoadAssembly    x
        UnloadAssembly  x  -> callbackUnloadAssembly  x
        ExitThread      x  -> callbackExitThread      x
        CreateThread    x  -> callbackCreateThread    x
        LogMessage      x  -> callbackLogMessage      x
        LoadModule      x  -> callbackLoadModule   x False
        UnloadModule    x  -> callbackUnloadModule x False
        LoadClass       x  -> callbackLoadClass    x False
        UnloadClass     x  -> callbackUnloadClass  x False
        ExceptionCB     x  -> callbackException    x False
```

Illustrating the control model spec of event handling (2)

```
handleStpEvent = case callback of
   ExitProcess     x    -> callbackExitProcess   x
   StepComplete    x    -> callbackStepComplete  x
   Breakpoint      x    -> callbackBreakpoint    x
   BreakCB         x    -> callbackBreak         x
   EvalComplete    x    -> callbackEvalComplete  x
   EvalException   x    -> callbackEvalException x
   DebuggerError   x    -> callbackDebuggerError x
   LoadModule x         -> callbackLoadModule x True
   UnloadModule x       -> callbackUnloadModule x True
   LoadClass x          -> callbackLoadClass x True
   UnloadClass x        -> callbackUnloadClass x True
   ExceptionCB x        -> callbackException x True
```

Experimenting user scenarios with control model

Row t exhibits state after t-th step of control model

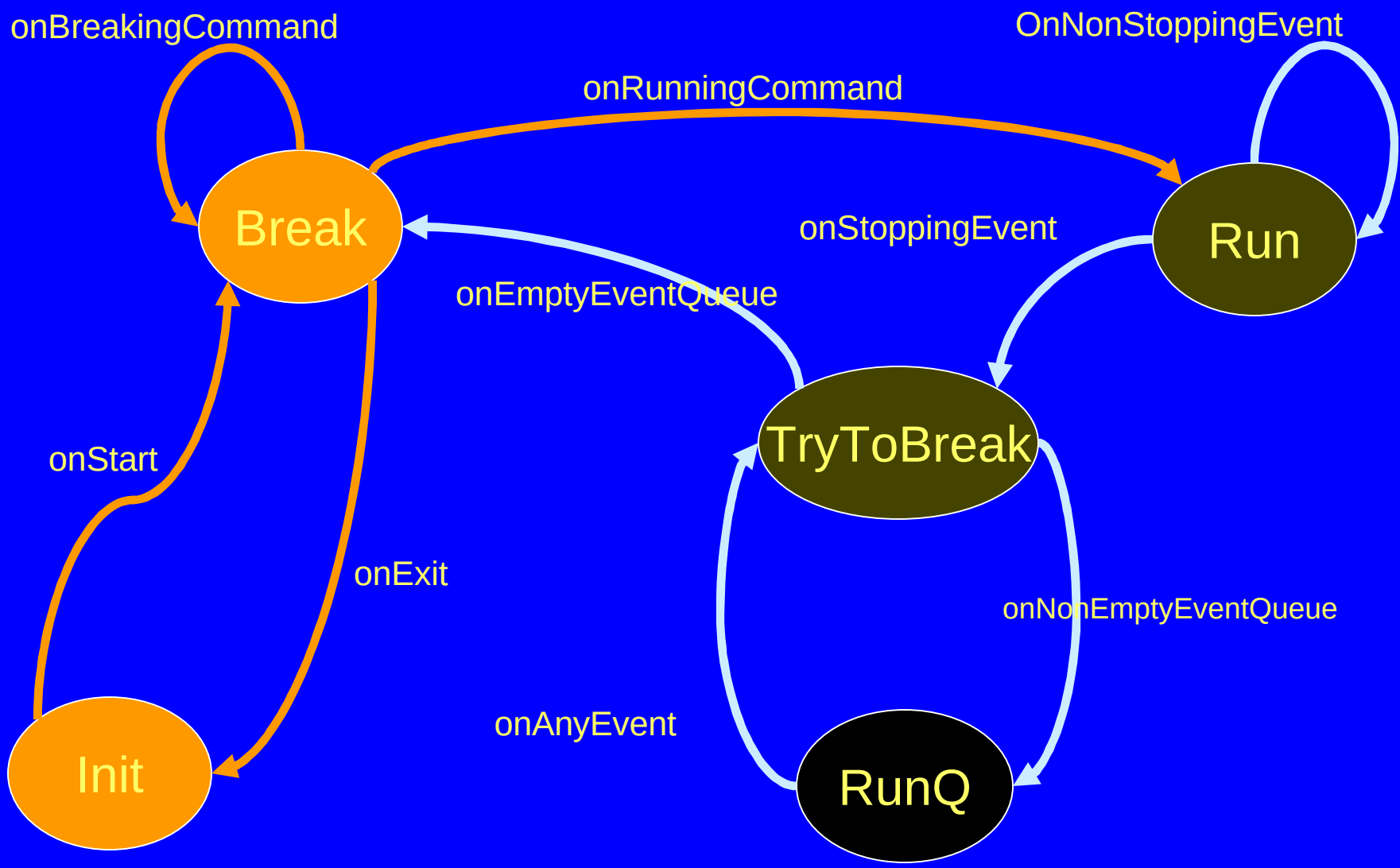


User	Environment	Events in queue
start		
b hello.cpp:13		
run hello.exe		
	Created process	
	.	
	.	
	Loaded module	
	Created 1st thread	
		False
continue		
	Hit breakpoint	
		True
	Loaded Class	
		False

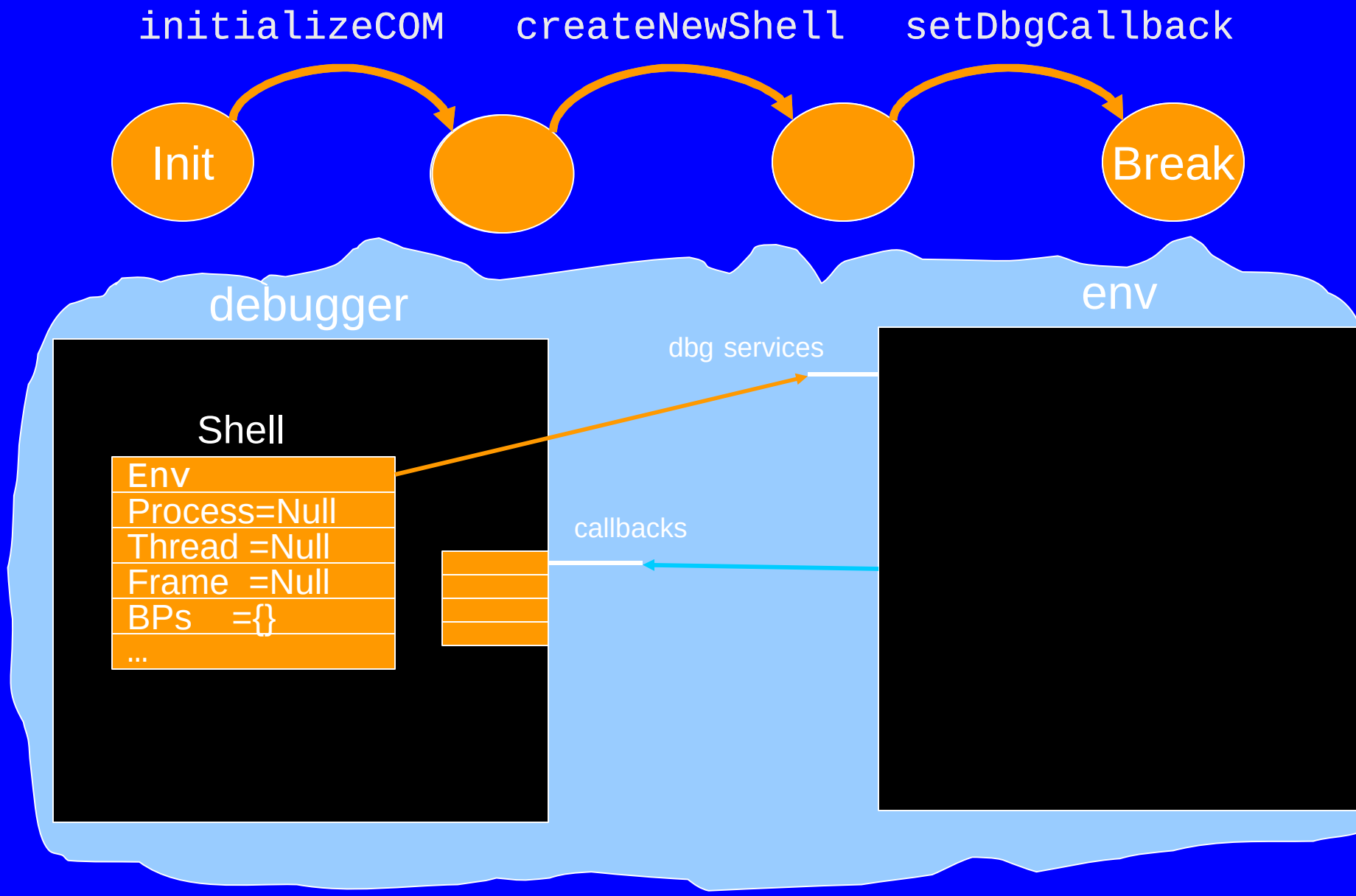
Test of the Control Model showing an undesired behavior (which then was found to have been fixed at the same time also in the C++ code)

After non-stopping class loading event in Run mode, Break mode is unreachable although breakpoint was reached

Control model rules (revised)

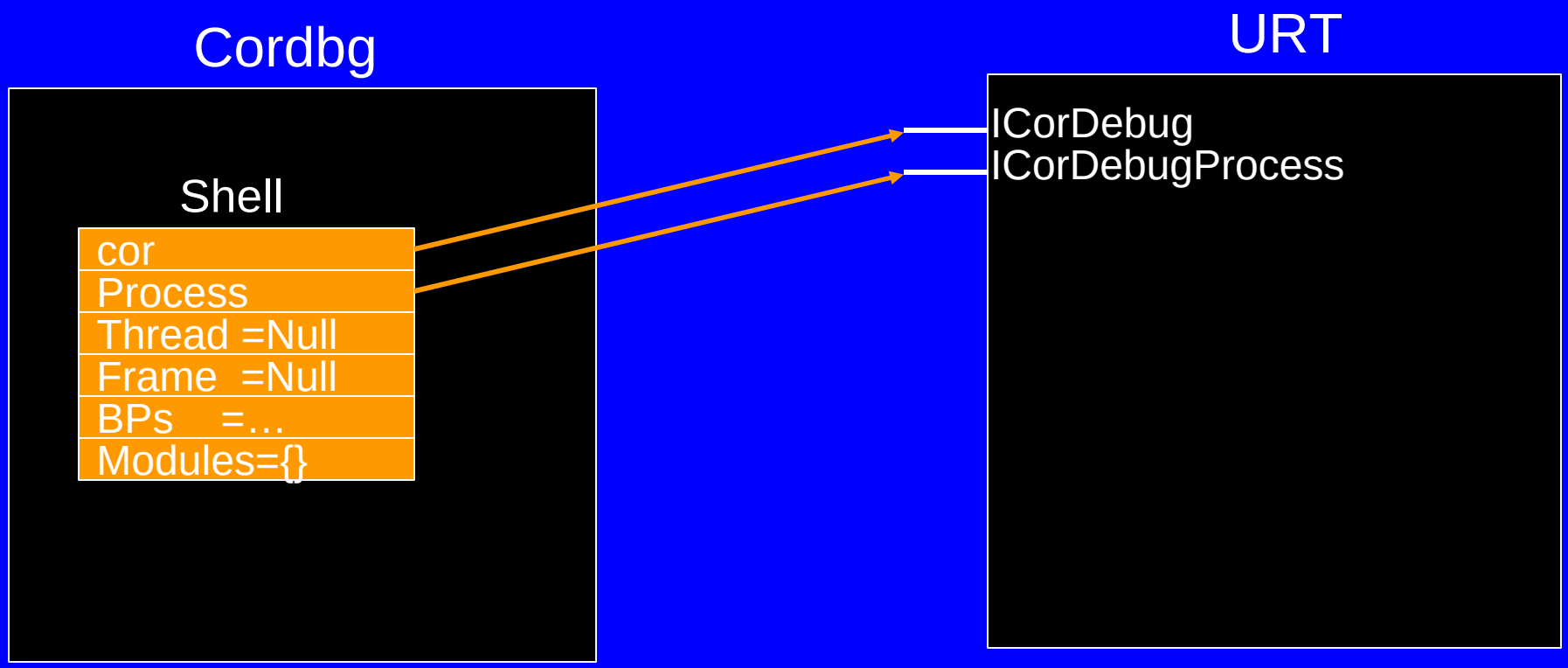


Object Model Refinement of machine onStart into a sequence of three submachines



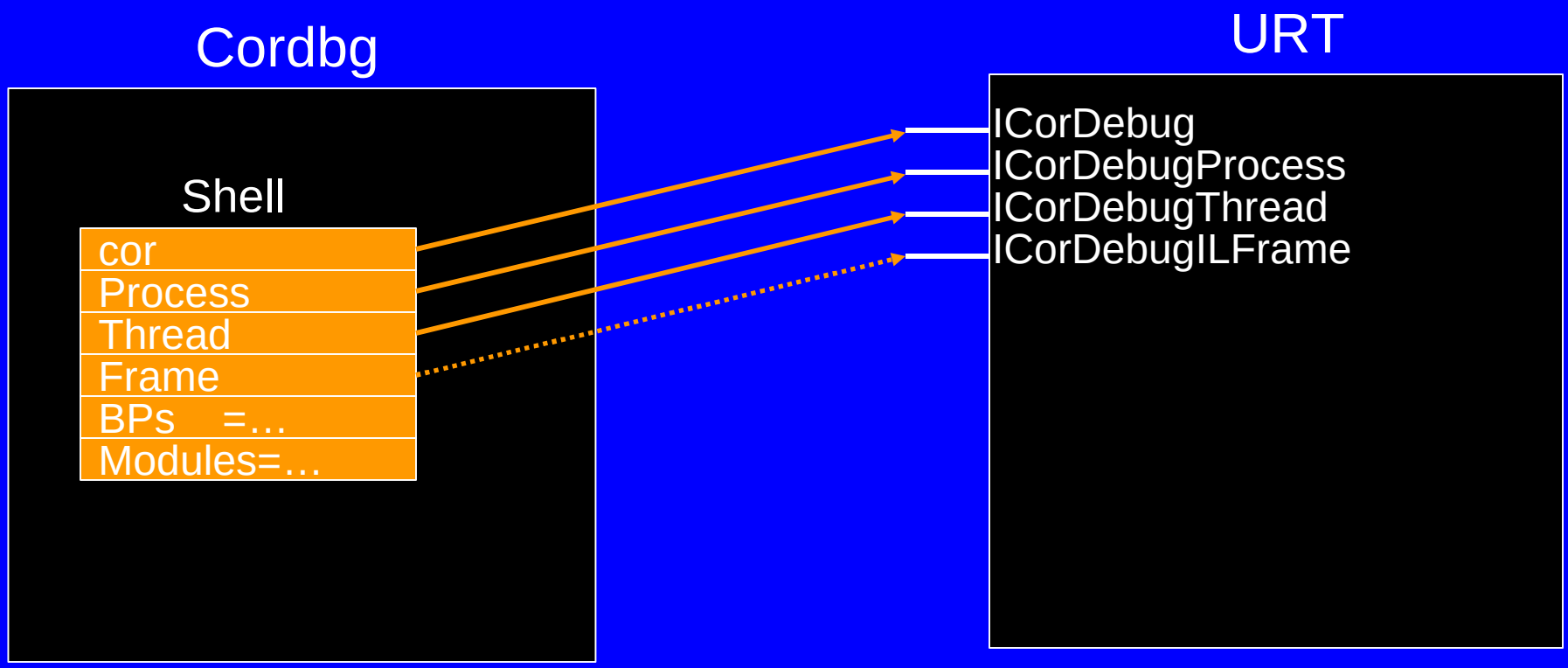
Object Model Refinement of commandRunPgm by Process creation

- commandRunPgm (name,args) =
  if shell.Process == Null
  then
  let proc = shell.cor.createProcess(name,args,…) **in**
  shell.Process := proc
  proc.continue2(0)



Object Model Refinement of callbackCreateThread

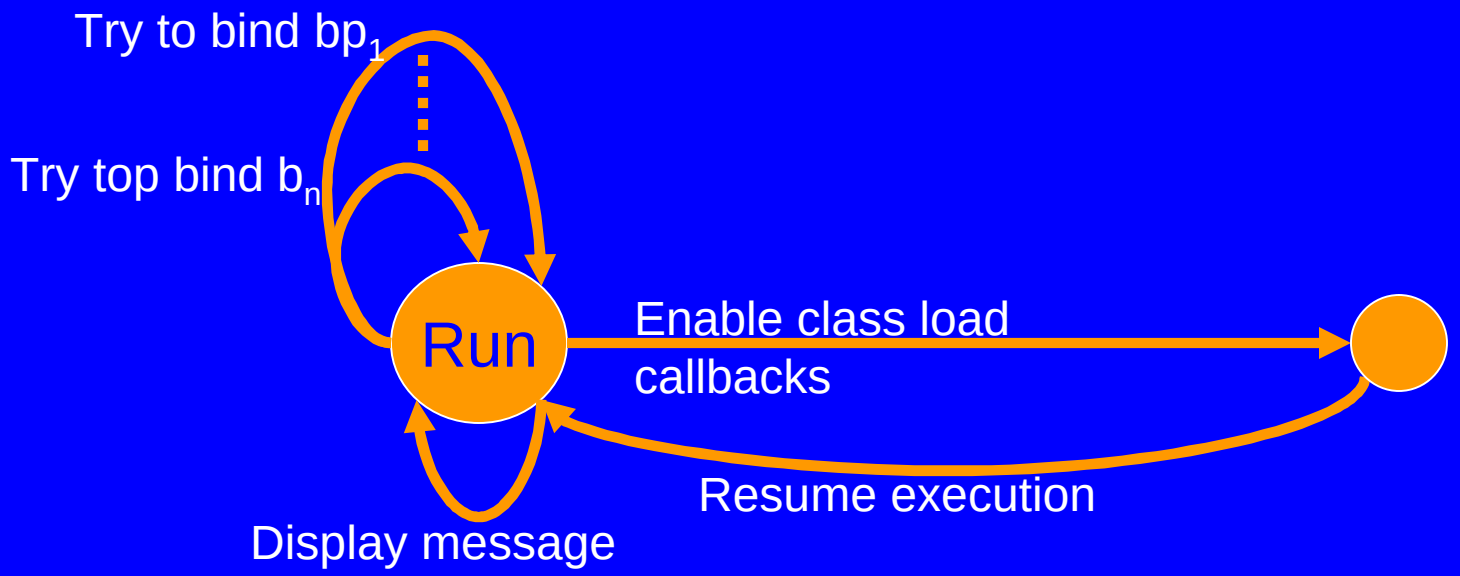
```
callbackCreateThread(appDom,thd) = let proc  = appDom.getProcess() in
      if dbgEvents(CatchThread) == On or not(gotFirstThread)
        then dbgMode       := TryToBreak
             shell.Process := proc
             shell.Thread  := thd
             if defined(thd) then shell.setDefaultFrame(thd)
        else proc.continue2(0)
      gotFirstThread := True
```



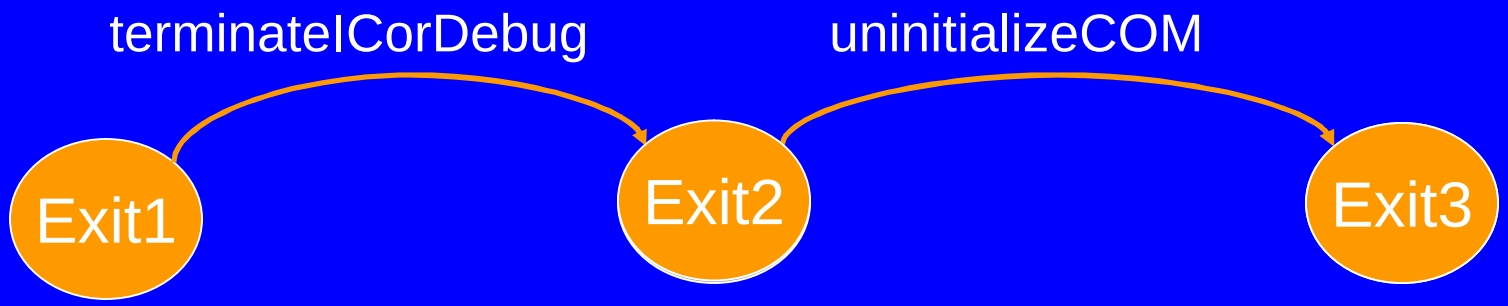
Refining Callback “LoadModule” in the Object Model

```
callback(LoadModule (proc,mod)) =
    displayMessage("Loaded module: " ++ mod.name())      record mod in shell
    forall bp in shell.BPs                  bind all breakpoints to the mod (in any order)
      bp.bind(mod)
    seq
        mod.enableClassLoadCallbacks()
        proc.resume()                                    continue via external call
```

Analogously for UnloadModule



Object Model Refinement of machine onExit into a sequence of two submachines (Break=Exit1, Exit3=Init)



Illustrating Further Object Model Refinements

```
continueCurrentProcess = if not shell.Process == Null
                         then shell.Process.continue2(0)

commandSetBreakpoint(location) =

let bp = newDbgBP(location,True) in
    shell.BPs := shell.BPs ++ [bp]
    forall mod in shell.Modules do bp.bind(mod)

where bp.bind(mod) = if not(bp.isBoundToModule(mod)) then
                       case bp.location of
                         FunctionName(loc)  -> bp.bindFn(mod,loc)
                         SrcFileName(loc)   -> bp.bindSF(mod,loc)

commandContinue = if not (shell.Process == Null)
                  then shell.Thread := Null
                       shell.Frame  := Null
                       shell.Process.continue2(0)
```

Reference

- **Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, Margus Veanes: Using Abstract State Machines at Microsoft: A Case Study**
  - **In:** Proc. ASM'2000 Workshop, Ascona (CH), March 20-24, 2000, Springer LNCS 1912, pp.367-379
- **E. Börger, R. Stärk:** Abstract State Machines. A Method for High-Level System Design and Analysis Springer-Verlag 2003, see http://www.di.unipi.it/AsmBook