因循

Agent Based Modeling with Coroutines

Presented at the 18th Annual Tcl/Tk Conference (Tcl'2011)
Manassas, VA

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 22185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Abstract:

Coroutines have been introduced into the Tcl/Tk core with version 8.6. And many developers ask "what on Earth would I do with them?" This paper describes how coroutines are used to model human actors following complex, interdependent procedures. During the paper, we will develop a coroutine based general use architecture for task management. We will also describe some of the common edge cases to look out for.

This paper is based on my experience developing the Integrated Recovery Model for T&E Solutions.

Introduction to Coroutines

What are Coroutines?

I was looking for a definition for coroutines, and I found a Chinese expression, 因循 *[yīn xún]* which translates[1] to:

- to continue the same old routine
- to carry on just as before
- to procrastinate

They are a form of cooperative multitasking. Depending on your application, they could replace threads. (de Maura, 2004)

Coroutines were introduced with TIP #328, and have been available in the Tcl core since Tcl/Tk 8.6a2. (Sofer, 2008)

This paper will focus on the application of coroutines for discrete time simulations. More specifically modeling human agents in naval casualty scenarios within T&E Solutions Integrated Recovery Model (IRM).

A Simple Example

Let's write a very simple task. Imagine we have a toy train. We want it to stop when it reaches a destination. Our environment provides a few functions:

- **close_enough** - Returns true if the agent is close enough to the target to be considered "there".
- **location** - Returns the current position of the agent.
- **motor_direction** - A procedure that calculates which direction is the target, Ahead (+1), Behind (-1), or Stop (0)
- **move_train** - Move the agent for one time step
- **place_train** - Manually set the position of the agent to an absolute location
- **speed** - Applies power to the agent's wheels: Forward (+1), Reverse (-1), or Stop (0)

Our microcontroller runs a Tcl-like interpreter, so the script for our task looks something like this:

```
proc move_to B {
 puts "Starting towards $B"
 set x [location]
 while {![close_enough $x $B]} {
  set x [location]
  puts "I am at $x"
  speed [motor_direction $x $B]
  move_train
 }
 speed 0.0
 puts "Arrived at $B"
}
place_train 0.0
move_to 100.0
puts "(Toot Toot)"
```

Run the script and we'll see:

```
Starting towards 100.0
I am at 0.0
I am at 1.0
I am at 2.0
...
I am at 98.0
I am at 99.0
I am at 100.0
Arrived at 100.0
(Toot Toot)
```

Of course, if this were running in a real microcontroller we wouldn't have a **move_train** routine. The laws of physics would take care of movement, and our task would simply be a monitor. We'll get to that later.

But bear with me, as I'm going to take this same logic and make it into a coroutine:

[1] Translation according to: http://www.websaru.com/因循.html

```
proc move_to B {
 puts “Starting towards $B”
 set x [location]
 while {![close_enough $x $B]} {
  set x [location]
  puts “I am at $x”
  speed [motor_direction $x $B]
  yield 1
 }
 speed 0.0
 puts “Arrived at $B”
 return 0
}
place_train 0.0
coroutine travel_to move_to 100.0
while {[travel_to]} {
 move_train
}
puts “(Toot Toot)”
```

Let’s go ahead and run our example, I’ll explain the notation in a second:

```
Starting towards 100.0
I am at 0.0
I am at 1.0
I am at 2.0
...
I am at 98.0
I am at 99.0
I am at 100.0
Arrived at 100.0
(Toot Toot)
```

Our output is the same, even though the proc **move_to** no longer calls **move_train**.

We use the *coroutine* command to create **travel_to**. **travel_to**, in turn, calls our **move_to** proc. The caller of **travel_to** sees whatever value is yielded or returned by **move_to**. And this arrangement we use to drive the *while* loop, which actually moves the train.

Try **move_to** on it’s own and you’ll see:

```
place_train 0.0
move_to 100.0
ERROR:
yield can only be called in a coroutine
```

The error is pretty self-explanatory. The yield command only makes sense to the Tcl interpreter within the confines of a coroutine.

Note that the “Starting towards” and “Arrived at” strings print only once, even though we call **travel_to** 100 times. That is because our coroutine picks up on the next call where it left off, at the **yield**.

yield can take an argument. That value is returned to the caller, as though it were given in a **return**.

Once a coroutine calls **return** it dies. If we to call **travel_to** after our *while* loop terminates we would would see:

```
travel_to
ERROR:
invalid command name "travel_to"
```

Let’s tweak our example. Say we would like our train to return to the place it left from.

```
proc travel_circuit {A B} {
  move_to $B
  puts "(Toot Toot)"
  move_to $A
  puts "(Toot Toot)"
  return 0
}
place_train 0.0
coroutine travel travel_circuit 0 100
while {[travel]} {
 move_train
}
puts "(Done)"
```

Our coroutine now calls a proc **travel_circuit** which calls our earlier proc

move_to. But it calls it twice with two different destinations.

```
Starting towards 100.0
I am at 0.0
I am at 0.0
I am at 1.0
I am at 2.0
...
I am at 98.0
I am at 99.0
I am at 100.0
Arrived at 100.0
(Toot Toot)
Starting towards 0.0
I am at 100.0
...
I am at 1.0
I am at 0.0
Arrived at 0.0
(Toot Toot)
```

The bot moves from A to B, reverses direction, and moves from B to A. The coroutine picks up wherever the **yield** left it. Even if the **yield** is inside of another procedure!

```
while ->
  travel ->
    travel_circuit ->
      move_to ->
        while ->
          yield
```

Coroutines and TclOO

Now, the next question you surely have. Can I use coroutines with TclOO? Yes!

Let's rebuild our example in object oriented code. The rest of the class is defined elsewhere. There's only one method that is interesting at the moment:

```
oo::define train {
 method move_to {B} {
  set x [my location]
  puts "[self] Starting towards $B"
  while {![close_enough $x $B]} {
   set x [my location]
   puts "[self] I am at $x"
   my speed [motor_direction $x $B]
   yield 1
  }
  puts "[self] Arrived at $B"
  my speed 0.0
  return 0
 }
}
proc travel_circuit {train A B} {
  $train move_to $B
  puts "(Toot Toot)"
  $train move_to $A
  puts "(Toot Toot)"
  return 0
}
train create zephyr
zephyr place_train 0.0
coroutine travel \
  travel_circuit zephyr 0.0 100.0
while {[travel]} {
 zephyr move_train
}
puts "(Done)"
```

Instead of running as a procedure, **move_to** is now a method in a TclOO object *zephyr*, of class *train*. **travel_circuit** is still a procedure, but we pass it the name of the object, and it calls the object's methods.

And we find that despite all of these changes, our example still works:

```
::zephyr Starting towards 100.0
::zephyr I am at 0.0
::zephyr I am at 0.0
::zephyr I am at 1.0
...
::zephyr I am at 98.0
::zephyr I am at 99.0
::zephyr I am at 100.0
::zephyr Arrived at 100.0
(Toot Toot)
::zephyr Starting towards 0.0
::zephyr I am at 100.0
::zephyr I am at 99.0
::zephyr I am at 98.0
...
::zephyr I am at 1.0
::zephyr I am at 0.0
::zephyr Arrived at 0.0
(Toot Toot)
(Done)
```

The coroutine has no problems descending into an object and exercising its methods. In fact, we could call out to multiple objects within a coroutine, and the coroutine would properly react as the specific object. Conversely, multiple coroutines could also call this same method.

Just to show this is an ordinary object, if we call that method outside of a coroutine, I still get the same error as our earlier **move_to** procedure:

```
zephyr move_to 100.0
ERROR:
yield can only be called in a coroutine
```

Coroutines as Objects

A useful property of coroutines is that they maintain their own internal state. If I define a variable, the value of that variable is preserved in between calls.

Let's suppose we are a lazy high schooler, and we want to solve the classic Two Trains Problem[2].

> Train A, traveling 70 miles per hour (mph), leaves Westford heading toward Eastford, 260 miles away. At the same time Train B, traveling 60 mph, leaves Eastford heading toward Westford. When do the two trains meet? How far from each city do they meet?

Instead of using algebra, we will brute force the solution with Tcl code. We begin by modeling each train with a coroutine. That coroutine calculates an updated position for the train every time step, and yields the current position:

```
proc advance {start end speed} {
  set x $start
  if { $start < $end } {
    set dX [expr $speed*$::dt]
  } else {
    set dX [expr -1.0 * $speed \
       * $::dt]
  }
  while 1 {
    set x [expr {$x + $dX}]
    yield $x
  }
  return $x
}
```

Our simulator is no longer looking for when the train reaches the destination. Instead, we are interested in when the position of **train_a** crosses **train_b**. Since the position of A is counting up, and B is counting down, we'll be at our solution point the iteration where A surpasses B in value:

[2] Text of the problem copied from:http://mathforum.org/dr.math/faq/faq.two.trains.html

```
set ::dt [expr {1/60.0}]
coroutine move_a advance 0 260 70
coroutine move_b advance 260 0 60
while {1} {
  set a [move_a]
  set b [move_b]
  if {$a > $b} break
}
puts "They Met at..."
puts "$a From Westford"
puts "[expr 260-$b] From Eastford"
puts "(Done)"
```

Run our simulation to get our answer:

```
They Met at...
140.0000000000001 From Westford
120.0 From Eastford
(Done)
```

Notice that we are running two copies of the same procedure at the same time. The fact they ran inside of two different coroutines meant that each had a different set of parameters, and each maintained a different recollection of X for every time step.

Discrete Time Agents

The simulations I work with play very much like board game. The scenario is broken into "steps". The steps are broken into phases, so that each actor gets a chance to affect the simulation equally.

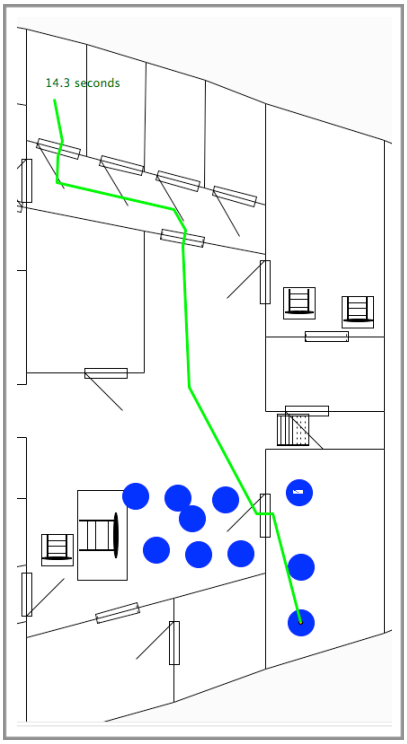
However, some physical phenomena don't tend to happen in neat 1 second intervals. Up until now, we have taken for granted that our agents move at a constant speed. Most simulations must account for momentum.

Before I'm accused of having a one track mind, let us transition away from examples with trains, and into problems I deal with in the real world. Well, real, virtual world.

Crew Movement

The major application thus far for coroutines within the IRM is modeling crew behavior.

Now you may be wondering, why did I start with so many examples of moving in one dimension? Crew can move in 2 dimensions, with a limited ability to move in the third dimension via stairways and ladders.



Well, it turns out that once the crew member has selected a route, he or she breaks the path into segments. Each of those segments is a line or spline, and we can consider the movement along it to be the very same one dimensional "Am I there yet?" problem that I opened this paper with.

Exception Handling

However, we have a few other rules that come into play.

Because we are calculating a route in a ship that can include spaces that are on fire, flooded, or both, it's a very real pos-

sibility that no route exists between two points. In that case we must fail our task.

An agent may find him or herself in a hazardous situation, or discover that a compartment he/she was intending to route through is inaccessible. If that is the case, he/she should withdraw to a safe location and compute a new route.

We also have to account for the fact that this task may be interrupted. And when we get control back, the agent may be in a different location than where we had intended to be.

```
method movement location {
  set here [my location]
  if {[my isNearby $destination]} {
    return 0
  }
  set route [crewroute find $here \
     $destination]
  if {[llength $route]==0} {return -1}
  my route $route
  while 1 {
    if {[my goal] != $destination} {
      return 2
    }
    if {[my hazard_detect]} {
      my withdraw
      return 2
    }
    if {[my isNearby $destination]} {
     return 0
    }
    yield 1
  }
}
```

In agent based modeling there are a different grades of exceptions. I imagine there are canonical terms for them, but I classify them as blocks, conflicts, and punts.

A **block** exception is when something external temporarily impedes the progress of our agent. The task simply bides it's time until the blockage has cleared.

A **conflict** exception is when two tasks require the same resource for mutually exclusive goals. A higher power sorts out which task gets priority. But the loser of that battle will have to restart from square one the next time it's called.

A **punt** exception is one which terminates the task because the conditions that justify the task's existence are no longer valid.

Standardize Yield and Return Codes

> *Fighting with a large army under your command is nowise different from fighting with a small one: it is merely a question of instituting signs and signals.*
> *--Sun Tsu, The Art of War, Chapter V*

One trouble with coroutines is that once they return a value, they cease to exist. Calling a completed coroutine will cause an error.

In my systems, I use the code returned to tell us the fate of the coroutine. An active coroutine yields a 1. Any other value indicates that the coroutine terminated, and will need to be restarted.

Code	Meaning
-1	Exception
0	Normal Exit
1	Running
2	Waiting
3	Blocked

The caller can interpret these codes, and react accordingly.

Task Nesting

It's very useful to break large goals into smaller goals that can be reused. We often have a crew member go out to a device, operate it, and come home.

But our toplevel task may want to respond to exceptions in it's own way.

I've found it useful to employ a bit of syntactic sugar in the form of the **subtask** command.

```
method attend {objective} {
  set location [objective location \
    $objective]
  # Go to the device
  while 1 [subtask movement $location]
  # Operate the device
  while 1 [subtask mitl $objective]
  # Return home
  set home [my home]
  while 1 [subtask movement $home]
  return 0
}
```

With **subtask**, we assume that a positive value (even if non-one) will not allow the program to continue. A zero indicates success, and allows the program to continue. A negative value represents an exception that should be punted.

Without **subtask**, the method above would look like:

```
method attend {objective} {
  set location [objective \
    location $objective]
  # Go to the device
  while 1 {
   set result [movement \
     $location]
   if { $result < 0 } {
     return -1
   } elseif { $result > 0 } {
     yield 1
   } else {
     break
   }
  # Operate the device
  while 1 {
    ....
```

(And continue on to fill the entire column on the right.)

The implementation for **subtask** is as follows:

```
proc subtask {cmd args} {
 set positive {yield 1}
 set negative {return $result}
 set zero {return 0}
 foreach {f v} $args {set $f $v}
 foreach f {
   positive negative zero cmd
 } {
  lappend replace %${f}% [set $f]
 }
 return [string map $replace {
  set result [{*}%cmd%]
  if { $result < 0 } {%negative%} \
  elseif { $result > 0 } {%positive%} \
  else {%zero%}
 }
}
```

Note, **subtask** doesn't run code, it builds code. That block of code becomes the body of the while loop.

subtask can take options (positive, negative, and zero) which allow the developer to control the agent's reactions to the sub-task's return code.

High Level Tasks

Agents often have to deal with competing goals. Because we've gone through the trouble of standardizing our return and yield codes, it's easy to detect when one goal is running, and could potentially block another task from running.

Let's refactor our methods so that we have three top level goals. One is to "attend". If the agent is assigned a device, he/she will walk to and operate the device. How the agent receives the assignment can vary. It is quite possible that after completing the first assignment the agent could have received a communication to do a second or a third. So it

wouldn't be very efficient to walk home after each time.

The next goal is to return home, but only if we have nothing to do.

Preempting either goals is the **safety_check**. **safety_check** is a reflex that will cause the agent to flee a space if he or she detects danger.

We also include a method "task" which will kick off a coroutine if it isn't operating yet, or evaluate one iteration of a coroutine that does exist.

You can see all of this put together in an example on the right.

Multitasking

All of this is work as built up to a system for multitasking that, while powerful, turns out to be simple and relatively uninteresting. Because coroutines are engaged in cooperative multitasking the loop for running an entire simulation with a few hundred agents can be as simple as:

```
proc simulation_step {} {
  physics_step
  foreach agent [agent::list] {
    $agent behavior
  }
}
```

In the IRM I have a routine no more complex than this that runs 40 odd crew members, 30 automated devices (which also behave as agents), and still operates in real time[3].

```
method attend {} {
 set objective [my get_assignment]
 if { $objective eq {} } {return 0}
 set location [objective \
    location $objective]
  # Go to the device
  while 1 [subtask \
    {movement $location} negative {
      record_failure $objective
      cancel_assignment $objective
      return 0
    }]
  # Operate the device
  while 1 [subtask mitl $objective]
  cancel_assignment $objective
  return 0
}

method go_home {} {
  set home [my home]
  while 1 [subtask movement $home]
  return 0
}

method safety_check {} {
  if {![my hazard_check]} {return 0}
  set dest [my escape_route]
  my route [route $dest]
  while 1 {
    if {![my hazard_check]} {return 0}
    yield 1
  }
  return 1
}

method task name {
 set coro [self]/coro_$name
 if {[info command $coro] == {} } {
  return [coroutine $coro [self] $name]
 } else {
  return [$coro]
 }
}

method behavior {} {
  my variable task_status
  set task_status {}
  foreach task {
    safety_check
    attend
    go_home
  } {
    set status [my task $task]
    dict set task_status $status
    if {$status > 1} break
  }
  return $task
}
```

3 Granted with a lot of the heavy calculations optimized in C.

Conclusions

Coroutines, while not new as a concept, are new to Tcl. In this paper I have have demonstrated that coroutines can be used to run complex discrete time simulations. And not just run, but run simply.

Coroutines are particularly well suited for simulations:

- That require multitasking across multiple agents
- Operate in discrete time
- Are amenable to cooperative multitasking.

Bibiliography

de Moura, Ana Lu ´cia and Ierusalimschy, Roberto, 2004, Revisiting Coroutines, (PUC-RioInf.MCC15/04 June, 2004), http://www.inf.puc-rio.br/~roberto/docs/MCC15-04.pdf, (October, 8 2011)

Sofer, Miguel and Madden, Neil, Coroutines, (Tip #328, Revision: 1.6), http://www.tcl.tk/cgi-bin/tct/tip/328.html, (October 9, 2011)