Integrating XTANGO's Animator into the SR Concurrent Programming Language

Stephen J. Hartley
Math and Computer Science Department
Drexel University
Philadelphia, PA 19104
(215) 895-2678
`shartley@mcs.drexel.edu`

August 23, 1994

Abstract

XTANGO's animation interpreter program, `animator`, has been very useful as a tool for animating concurrent programs written for an operating systems or concurrent programming class. Additional print statements can be added to a program and the output of the program can be read by the animation interpreter. The resulting algorithm animation will appear in a new window on the user's workstation running X-windows. SR is a concurrent programming language that can be used by students in an operating systems or concurrent programming class.

This paper describes integrating XTANGO's animation interpreter into SR so that procedure calls can be made directly to the animation code, rather than generating an intermediate output file or piping the SR program's output to the `animator` program. Two new animation commands, `stepjump` and `stepjumpto`, were added so that more that one object can be moving at a time in the XTANGO window.

Algorithm animation using SR is now even easier. An example SR program is included.

1 Animating SR Programs with XTANGO

In operating systems courses, students study various classical synchronization problems such as the dining philosophers, the readers and writers, and the producers and consumers with bounded buffer [7]. In concurrent and parallel programming courses, students study parallel versions of sorting algorithms, such as quicksort, merge sort, pipeline sort, and compare-exchange sort [5]. Even though the computing systems being studied are getting more and more powerful, these algorithms and classical problems are usually analyzed using blackboard, chalk, pencil, and paper.

John Stasko has written a package called XTANGO [6] that can be used to animate C language programs. The package contains a library of routines that can be called to draw animations in an X-window. Also included is a stand-alone program, `animator`, that can read window drawing commands from a file or UNIX pipe.

The SR concurrent programming language [1] can be used as the programming environment in an operating systems class [2, 3] to give students practical experience with semaphores, monitors, message passing, and the rendezvous. By putting in additional print statements, students can animate their SR programs that simulate the dining philosophers and other classical operating systems synchronization problems [4]. The output of the program can be saved in a file to be read later by the animation interpreter, or a UNIX pipe can be set up as follows.

```
sr -o simulation simulation.sr
simulation | animator
```

Table 1 lists the `animator` commands with a brief description of each.

2 Integrating `animator` into SR

The original `animator` interpreter program from XTANGO is a C language program that parses lines of input containing commands and their arguments. Once a command is parsed, a procedure of the name of the command is called to interface with the XTANGO library routines.

Submitted to the Twenty-Sixth SIGCSE Technical Symposium, March 2-4, 1995, Nashville, Tennessee.

command and arguments	brief description
`bg colorval`	change background color
`coords lx by rx ty`	change the displayed coordinates
`delay steps`	regenerate same animation frame
`line id xpos ypos xsize ysize colorval widthval`	draw a line
`rectangle id xpos ypos xsize ysize colorval fillval`	draw a rectangle
`circle id xpos ypos radius colorval fillval`	draw a circle
`triangle id v1x v1y v2x v2y v3x v3y colorval fillval`	draw a triangle
`text id xpos ypos centered colorval string`	display some text
`bigtext id xpos ypos centered colorval string`	use a larger font
`move id xpos ypos`	smoothly move the object
`moverelative id xdelta ydelta`	smoothly move the object
`moveto id id`	smoothly move the object
`jump id xpos ypos`	move the object in one jump
`jumprelative id xdelta ydelta`	move the object in one jump
`jumpto id id`	move the object in one jump
`color id colorval`	change object color
`delete id`	delete the object
`fill id fillval`	change object fill value
`vis id`	toggle object visibility
`lower id`	push object to lower viewing plane
`raise id`	raise object to closer viewing plane
`exchangepos id id`	smoothly exchange positions
`switchpos id id`	exchange positions in one jump
`swapid id id`	exchange ids

Table 1: XTANGO's `animator` Commands.

To integrate `animator` into SR, the input line parsing code of `animator.c` was taken out. An SR global resource, `SRanimator`, was written that consists of an `in` statement (rendezvous) inside a "do forever" loop. The `in` statement accepts calls to the interface routines, exported by the global. The names of the interface routines are the same as the original `animator` commands, prefixed with "`A_`". The body of each branch of the `in` statement calls the corresponding command in `animator.c`.

When compiling and linking an SR program, the user must also compile the global resource containing the interface routines and link in the modified animation code as follows. Note that several X-windows libraries must be specified during linking.

```
cc -c animator.c
sr -c SRanimator.sr
sr -c simulation.sr
srl -o simulation SRanimator \
    other resources animator.o xtango.o \
    -lXaw -lXmu -lXext -lXt -lX11 -lm
```

3 Two New Animation Commands

The `move`, `moverelative`, and `moveto` commands move an object from one location on the screen to another in a sequence of smaller steps so the motion is smooth in the window. The corresponding `jump` commands move the object in one instantaneous step. Since the XTANGO library routines were designed for sequential programs, they can handle only one operation at a time, and therefore only one object at a time can be moving in the window. This is a limitation for the animation of concurrent programs which need two or more objects moving at the same time. To deal with this, two new commands were added to the animation code integrated into SR: `stepjump` and `stepjumpto`. Each of these takes two additional arguments, the number of steps and the number of milliseconds between each step. These commands are implemented inside `SRanimator` by breaking a jump up into the number of steps specified and sleeping (SR `nap()` function) the number of milliseconds specified between each smaller jump. While this is being done, other animation commands can be processed.

4 An Example

The Appendix shows the SR code for a simulation of the dining philosophers without a central server process (section 13.3 of [1]). Figure 1 shows a snapshot of the animation window during a simulation. Two forks, the ones philosopher 4 has just finished using, can be seen to be in motion at the same time. The code uses a call to `A_stepjumpto` to perform a fork move in five steps.

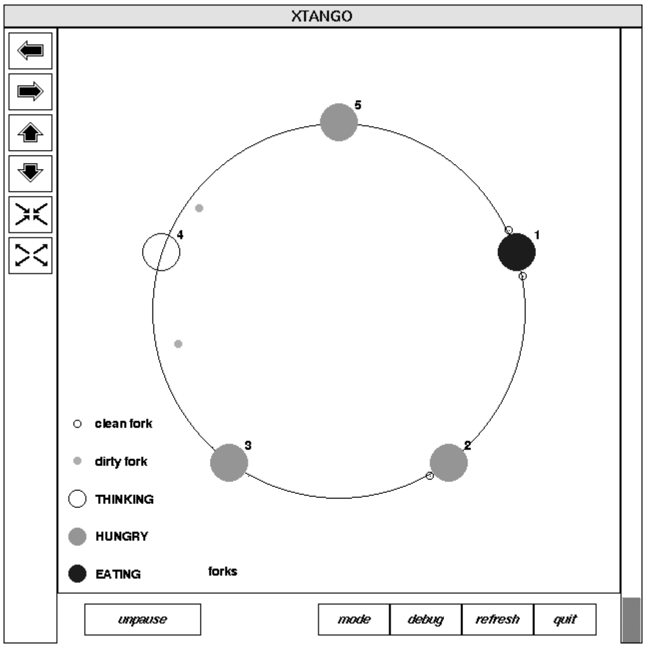

Figure 1: Animation Snapshot of the Distributed Dining Philosophers.

5 Comparison with SRWin

SR has a built-in interface to X-windows called SRWin [8]. SRWin is at a lower level and "closer" to the X-windows system. In my opinion, it is harder for students to use SRWin for algorithm animation than XTANGO's `animator` command set.

6 Conclusions

Integrating XTANGO's `animator` into SR has made animation of concurrent SR programs even easier for students and instructors than described in [4]. The animation can be accomplished with less overhead since commands do not have to be formatted with a print statement, written to a file or pipe, and then parsed by the `animator` program; the binary values of the command arguments are sent to `SRanimator` using SR's message passing. The SR development team at the University of Arizona will include `SRanimator` into a future release (perhaps late 1994).

References

[1] Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings Publishing, 1993 (the SR language is available by anonymous `ftp` from machine `cs.arizona.edu` in file `/sr/sr.tar.Z`).

[2] Stephen J. Hartley, "Experience with the Language SR in an Undergraduate Operating Systems Course," *ACM SIGCSE Bulletin*, Vol. 24, No. 1, March 1992.

[3] Stephen J. Hartley, "An Operating Systems Laboratory Based on the SR (Synchronizing Resources) Programming Language," *Computer Science Education*, Vol. 3, No. 3, 1992.

[4] Stephen J. Hartley, "Animating Operating Systems Algorithms with XTANGO," *ACM SIGCSE Bulletin*, Vol. 26, No. 1, March 1994.

[5] Michael J. Quinn, *Parallel Computing: Theory and Practice*, second edition, McGraw-Hill, 1994.

[6] John T. Stasko, "XTANGO Algorithm Animation Designer's Package," available by anonymous `ftp` from machine `par.cc.gatech.edu` (from directory `pub`, retrieve file `xtango.tar.Z`, then uncompress and extract file `xtangodoc.ps` from directory `./xtango/doc` in the archive file `xtango.tar`).

[7] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

[8] Qiang A. Zhao, *SRWin, a Graphics Library for SR*, Department of Computer Science Technical Report 93-14, University of Arizona, May 1993 (available by anonymous `ftp` from machine `cs.arizona.edu` in file `/reports/1993/TR93-14.ps`).

A Appendix: Distributed Dining Philosophers

```
# Distributed Dining Philosophers.  Based
# on the example in Section 13.3 of "The
# SR Programming Language: Concurrency in
# Practice", by Greg Andrews and Ron Olsson,
# Benjamin/Cummings, 1993.  The algorithm
# is from Chandy and Misra, "Drinking
# Philosophers Problem", ACM TOPLAS v 6,
# n 4, Oct 1984, pp 632-646.
#
# Usage:  a.out [n secs t_secs e_secs
#                t_secs e_secs ...]
# (for n philosophers, secs seconds running
# time, t_secs max thinking seconds, e_secs
# max eating seconds for each philosopher)
#
# Modified for xtango's animator interpreter
# imported global resource SRanimator.  All
# animator calls start with "A_".

resource Philosopher
 import Servant, SRanimator
body Philosopher(myservant : cap Servant;
```

```
  id, thinking, eating: int)
 process phil
  do true ->
   nap(int(random(thinking)))
   write(age(), "philosopher", id,
    "is hungry")

# Change a hungry philosopher's symbol
# to be solid green.
   A_color(id, "green")
   A_fill(id, "solid")
# Above for animator.

   myservant.getforks()

# Change an eating philosopher's symbol
# to be solid blue.
   A_color(id, "blue")
# Above for animator.

   nap(int(random(eating)))

# Change a thinking philosopher's symbol
# to an outline black circle.
   A_fill(id, "outline")
   A_color(id, "black")
# Above for animator.

   myservant.relforks()
  od
 end phil
end Philosopher

resource Servant
 import SRanimator
 op getforks(), relforks()
 op needR(), needL(), passR(), passL()
 op links(r, l : cap Servant)
 op forks(haveR, dirtyR,
  haveL, dirtyL : bool)

# This op and the variables below are used
# to hold the symbol id numbers of the
# left and right fork symbols and where
# the left and right forks are placed next
# to the philosopher when it possesses them.
 op fork_ids(forkR, forkL,
  holderR, holderL : int)
# Above for animator.

body Servant(id : int)
 op hungry(), eat()
 var r, l : cap Servant
 var haveR, dirtyR, haveL, dirtyL : bool

# Used with the fork_ids op above.
 var forkR, forkL, holderR, holderL : int
# Above for animator.

 proc getforks()
  send hungry()
  receive eat()
 end

 process server
  receive links(r, l)
  receive forks(haveR, dirtyR,
   haveL, dirtyL)

# Move a left or right fork initially
# given to this philosopher to be next
# to the philosopher.
  receive fork_ids(forkR, forkL,
   holderR, holderL)
  if haveR ->
   A_jumpto(forkR, holderR)
  fi
  if haveL ->
   A_jumpto(forkL, holderL)
  fi
  if dirtyR ->
   A_color(forkR, "orange")
   A_fill(forkR, "solid")
  fi
  if dirtyL ->
   A_color(forkL, "orange")
   A_fill(forkL, "solid")
  fi
# Above for animator.

  do true->
   in hungry() ->
    if ~haveR ->
     send r.needL()
    [] else ->
     write(age(), "philosopher", id,
      "has right fork")
    fi
    if ~haveL ->
     send l.needR()
    [] else ->
     write(age(), "philosopher", id,
      "has left fork")
    fi
    do ~(haveR & haveL) ->
     in passR() ->
      haveR := true; dirtyR := false
      write(age(), "philosopher", id,
      "got right fork")

# Move the right fork from where it was
# to be next to this philosopher and then
# change its symbol to be a black outline
# circle i.e. not dirty.  Also raise the
# fork's symbol to the closest viewing
# plane to make it more visible.
      A_stepjumpto(forkR, holderR, 5, 100)
      A_fill(forkR,  "outline")
      A_color(forkR, "black")
      A_raise(forkR)
```

```
# Above for animator.

     [] passL() ->
      haveL := true; dirtyL := false
      write(age(), "philosopher", id,
       "got left fork")

# Ditto for the left fork.
      A_stepjumpto(forkL, holderL, 5, 100)
      A_fill(forkL,  "outline")
      A_color(forkL, "black")
      A_raise(forkL)
# Above for animator.

     [] needR() st dirtyR ->
      haveR := false; dirtyR := false
      send r.passL(); send r.needL()

      write(age(), "philosopher", id,
       "sends dirty right fork")
     [] needL() st dirtyL ->
      haveL := false; dirtyL := false
      send l.passR(); send l.needR()
      write(age(), "philosopher", id,
       "sends dirty left fork")
     ni
    od
    write(age(), "philosopher", id,
     "has both forks")
    send eat()
    dirtyR := true; dirtyL := true
    receive relforks()
    write(age(), "philosopher", id,
     "is finished eating")

# Now that the philosopher has finished
# eating, its forks are dirty so change
# their symbols to be solid orange circles.
    A_color(forkL, "orange")
    A_fill(forkL, "solid")
    A_color(forkR, "orange")
    A_fill(forkR, "solid")
# Above for animator.

   [] needR() ->
    haveR := false; dirtyR := false
    send r.passL()
    write(age(), "philosopher", id,
     "sends right fork")
   [] needL() ->
    haveL := false; dirtyL := false
    send l.passR()
    write(age(), "philosopher", id,
     "sends left fork")
   ni
  od
 end server
end Servant

resource Main()
 import Philosopher, Servant, SRanimator
 var n := 5; getarg(1, n)
 var runtime := 60; getarg(2, runtime)
 var s[1:n] : cap Servant
 var p[1:n] : cap Philosopher
 var think[1:n] : int := ([n] 5)
 var eat[1:n] : int := ([n] 2)
 fa i := 1 to n ->
  getarg(2*i+1, think[i])
  getarg(2*i+2, eat[i])
 af

 write(n, "philosophers;",
  "think, eat times in seconds:")
 fa i := 1 to n ->
  writes(" ", think[i])
 af
 write()
 fa i := 1 to n ->
  writes(" ", eat[i])
 af
 write()

# Change coordinates so 0,0 is the center,
# then create a big black outline circle
# to be the table.
 A_coords(-1.5, -1.5, 1.5, 1.5)
 A_circle(1000, 0.0, 0.0, 1.0, "black", "outline")
# Put some annotated symbols on the screen.
 A_circle(1001, -1.4, -0.6, 0.02, "black", "outline")
 A_text(1002, -1.3, -0.625, 0, "black", "clean fork")
 A_circle(1003, -1.4, -0.8, 0.02, "orange", "solid")
 A_text(1004, -1.3, -0.825, 0, "black", "dirty fork")
 A_circle(1005, -1.4, -1.0, 0.05, "black", "outline")
 A_text(1006, -1.3, -1.025, 0, "black", "THINKING")
 A_circle(1007, -1.4, -1.2, 0.05, "green", "solid")
 A_text(1008, -1.3, -1.225, 0, "black", "HUNGRY")
 A_circle(1009, -1.4, -1.4, 0.05, "blue",  "solid")
 A_text(1010, -1.3, -1.425, 0, "black", "EATING")
# Put a clean set of forks, small black
# outline circles, near the table.
 A_text(1011, -1.0+0.05*(n+1), -1.41, 0,
  "black", "forks")
 fa i := 1 to n ->
  A_circle(3000+i-1, -1.0+0.05*i, -1.4, 0.02,
   "black", "outline")
 af
 const TWO_PI := 2.0*acos(-1.0)
 fa i := 1 to n ->
# Put the philosophers, black outline
# circles, around the table.
  A_circle(i, sin(i*(TWO_PI/n)), cos(i*(TWO_PI/n)),
   0.1, "black", "outline")
# Number the philosophers.
  A_text(2000+i, sin(i*(TWO_PI/n))+0.1,
   cos(i*(TWO_PI/n))+0.1, 1, "black", string(i))
# Put nearly invisible circles (points)
# on the left and right side of each
# philosopher to be places the forks can
# be moved to when the philosopher gets
```

```
# possession of a fork.
  A_circle(4000+2*i, sin(i*(TWO_PI/n)-0.12),
   cos(i*(TWO_PI/n)-0.13), 0.001, "black", "outline")
  A_circle(4000+2*i+1, sin(i*(TWO_PI/n)+0.12),
   cos(i*(TWO_PI/n)+0.13), 0.001, "black", "outline")
 af
# Above for animator.

 fa i := 1 to n ->
  s[i] := create Servant(i)
  create Philosopher(s[i], i, 1000*think[i],
   1000*eat[i])
 af
 fa i := 1 to n ->
  send s[i].links(s[((i-2) mod n)+1],
   s[(i mod n)+1])
 af
 send s[1].forks(true, true, true, true)
 fa i := 2 to n-1 ->
  send s[i].forks(false, false, true, false)
 af
 send s[n].forks(false, false, false, false)

# Send to each philosopher the xtango
# animator symbol id's of the two forks
# the philosopher needs to eat and the
# places where possessed forks are to
# be moved next to the philosopher.
 fa i := 1 to n ->
  send s[i].fork_ids(3000+((i-1) mod n),
   3000+(i mod n), 4000+2*i, 4000+2*i+1)
 af
# Above for animator.

 nap(1000*runtime)
 write("must stop now")
 A_end()
 stop
end Main
```