CSC 343 – Operating Systems, Fall 2020, Intro to our UML State Machines

Dr. Dale E. Parson, http://faculty.kutztown.edu/parson

To compile and run this course's state machines, you will have to log onto machine **mcgonagall** from your usual **acad** Unix account by running **ssh mcgonagall**. We will use mcgonagall because it has 32 fast processors; we can run CPU-bound simulations in parallel without swamping machines used by other courses.

Assuming that you are logged into acad, here are the steps need to run the STM (state machine) demo.

ssh mcgonagall
cd $HOME
mkdir OpSys
cd ./OpSys
cp ~parson/OpSys/demo1fall2020.zip demo1fall2020.zip
unzip demo1fall2020.zip
cd ./demo1fall2020
make clean test

Running the above steps on mcgonagall should yield this output:

$ **make clean test**
/bin/rm -f *.o *.class .jar core *.exe *.obj *.pyc __pycache__/*.pyc
/bin/rm -f *.out *.dif *.pyc junk parsetab.py *.vmlf *.png
/bin/rm -f *.dot *.gif *.jpg testmachine.ck junk.* *.tmp *.log
/bin/rm -f Human1STM.py *.crunch *_crunch.py /tmp/parson_STM_*.log parson_STM_*.log *_crunch.csv
COMPILING Human1STM
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. /opt/anaconda3/bin/python3 /home/kutztown.edu/parson/OpSys/state2codeV17/State2CodeParser.py Human1STM.stm Human1STM.dot Human1STM.py CSC343Compile CSC343Compile"
COMPILING COMPLETED
SIMULATING (TESTING) Human1STM
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. time /opt/anaconda3/bin/python3 Human1STM.py 2 4 876000 12345 2"
MSG cmd line: ['Human1STM.py', '2', '4', '876000', **'12345'**, '2'], usage USAGE: python THISFILE.py NUMCONTEXTS NUMFASTIO SIMTIME **SEED**|None LOGLEVEL

Scheduler exiting at time 814681 within time limit 876000, simulation has finished.
0.04user 0.01system 0:00.06elapsed 81%CPU (0avgtext+0avgdata 9200maxresident)k
0inputs+16outputs (0major+4573minor)pagefaults 0swaps
/bin/bash -c "PYTHONPATH=/home/kutztown.edu/parson/OpSys:.:.. /opt/anaconda3/bin/python3 crunchlog.py Human1STM.log"

DIFFing Human1STM_crunch.py Human1STM_crunch.ref
OK: MEAN_infant
OK: MEAN_teen

OK: MEAN_youngAdult
OK: MEAN_midLife
OK: MEAN_gettingOld

TESTING COMPLETED

Note the **SEED** of **12345** in the generated command line that runs the simulation. A fixed **SEED** used to initialize a pseudo-random number generator causes that generator such as the **sample()** function in STM.doc.txt to emit the same sequence of numbers, each time it is run. A **SEED** of **None** on the command line uses no fixed seed, so each generated sequence of numbers will be different. Our testing uses a fixed SEED so we can duplicate testing conditions exactly for catching errors and debugging.

The above steps compile the STM file **Human1STM.stm** into a Python program and run the generated simulation file, performing a statistical comparison between its simulation log file and my handout, reference simulation log file. If the performance of the simulation runs matches mine to within some degree of error (we will go over the details), then the tests pass. Next we will look at the files involved and the constructs for a STM. Running make test produces a raw simulation log file (**Human1STM.log**) and a derived data analysis file (a so-called crunch file **Human1STM_crunch.py**).
Human1STM.stm

https://kuvapcsitrd01.kutztown.edu/~parson/Human1STM.txt

make graphs produces these files

https://kuvapcsitrd01.kutztown.edu/~parson/Human1STM.jpg
https://kuvapcsitrd01.kutztown.edu/~parson/Human1STM_crunch.png

Initially your STM simulation files will consist of two state machines, one called **processor** and the other called **thread**. We may add a third called **io** later in the semester. The primary job of **processor** is to create one or more threads of execution by invoking the **fork()** library procedure. The **processor** also houses some inter-process data structures in later assignments. All of your work for now will go into **thread** state machines.

Each **processor** and **thread** STM is an **active object** in UML parlance, which means that multiple STMs run concurrently in our simulation environment. Every time a **processor fork()s** a new **thread** STM, for example, it creates one **thread** object for each **fork()** call, with each **thread** object running its own instruction stream. Multiple processes and threads will be central concepts to learn in this course. For now, though, there is only one **fork()** call inside **processor**, and hence only one **thread** STM.

Each state machine consists of the following sections, as seen in the code above.

1. A **machine** name, **processor** or **thread** for now.
2. Initialization of **state variables**. Any variable assigned within your code becomes a field in its state machine object. Two distinct thread objects created by **fork()**, for example, have their own distinct state variables. The declarations shown in lines 28 and 29 above show initialization of five state variables. You can initialize a state variable only to a constant value. Expressions come later.
3. Next come **state** declarations. There is always one **start state** in which the machine begins execution; in our programs its name is **init**. Then there are zero or more **regular states**, and one or more **accept states** in which the machine terminates its execution. See lines 31 and 32.
4. The STM compiler supports **macro** definitions, which we will use later in the semester.

5. Next come the **transitions**, which comprise the bulk of our state machines. Each transition leads from one state to another (or back to the same state). Each transition consists of one or more of the following parts.

   FROMSTATE -> TOSTATE EVENT(ARGS) [@OPTIONALGUARD@]/@ACTIONS@

The **FROMSTATE** is the state which execution leaves when the named **EVENT** arrives.

The **TOSTATE** is the state which execution enters when the **EVENT** arrives.

Some **EVENTS** types carry one or more **ARGS** (arguments) documented for these events. **ARGS** are optional, and the parentheses may be empty.

The **OPTIONALGUARD** is a Boolean expression (evaluates to True or False) that tests state variables and/or **ARGS**. It blocks the transition when it is False, i.e., the transition is not taken in that case. An empty [] guard evaluates to True. A non-empty guard uses a pair of @@ symbols to delimit the guard, which is a Python expression. Python uses **and**, **or** and **not** instead of &&, || and ! used in C++ and Java.

The ACTIONS consist of one or more lines of stylized Python code, separated by semicolons, that include **calls to library functions and procedures**, **assignments into state variables**, and **arithmetic expressions**. Python supports a conditional expression similar to this construct from C, C++ and Java:

BOOLEAN ? EXPRESSION1 : EXPRESSION2

STM uses this Python syntax for conditional expressions:

EXPRESSION1 if BOOLEAN else EXPRESSION2

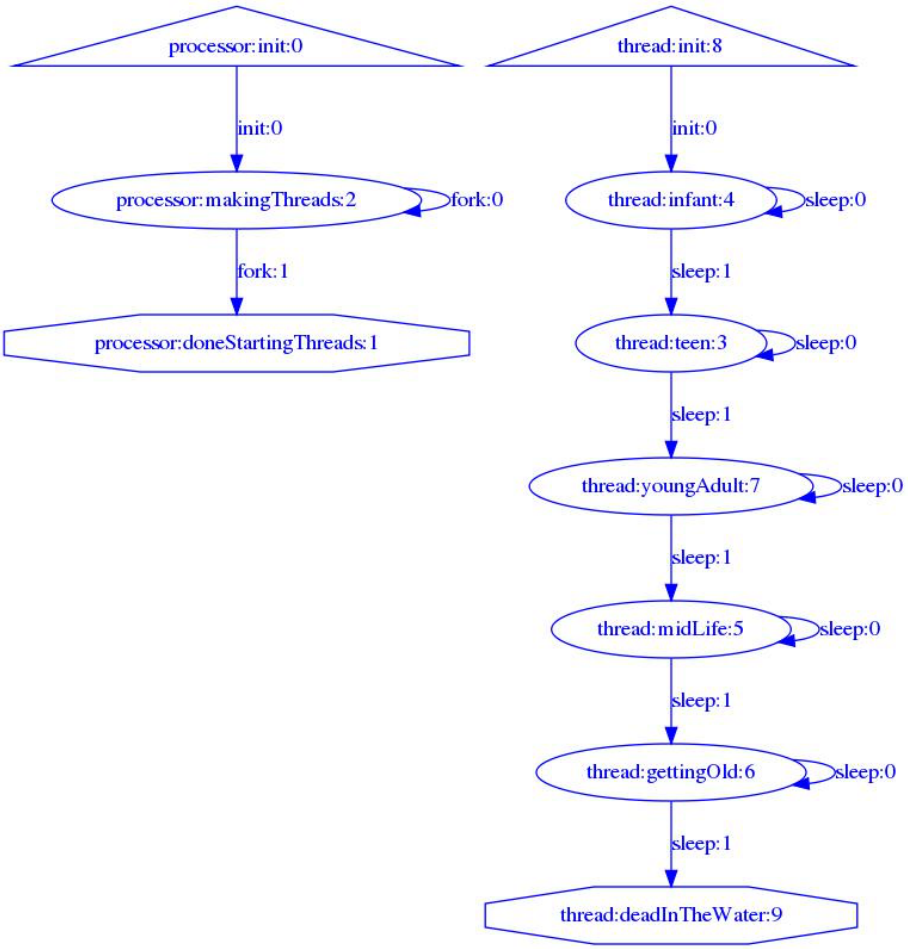
The semantics are the same for both constructs. If the **BOOLEAN** expression evaluates to True, then **EXPRESSION1** is the result of overall evaluation, else **EXPRESSION2** is the result. You can use a conditional expression just like any arithmetic expression (+ or *, for example), and assign its result into a variable. Conditional expressions will allow you to avoid writing multiple transitions with alternative guards in some cases in later projects.

The state machine above simulates the life of a rather boring human who does nothing but sleep for specific periods of time. We will go over the execution of this STM in class, and you will add some complexity into its life in Assignment 1.

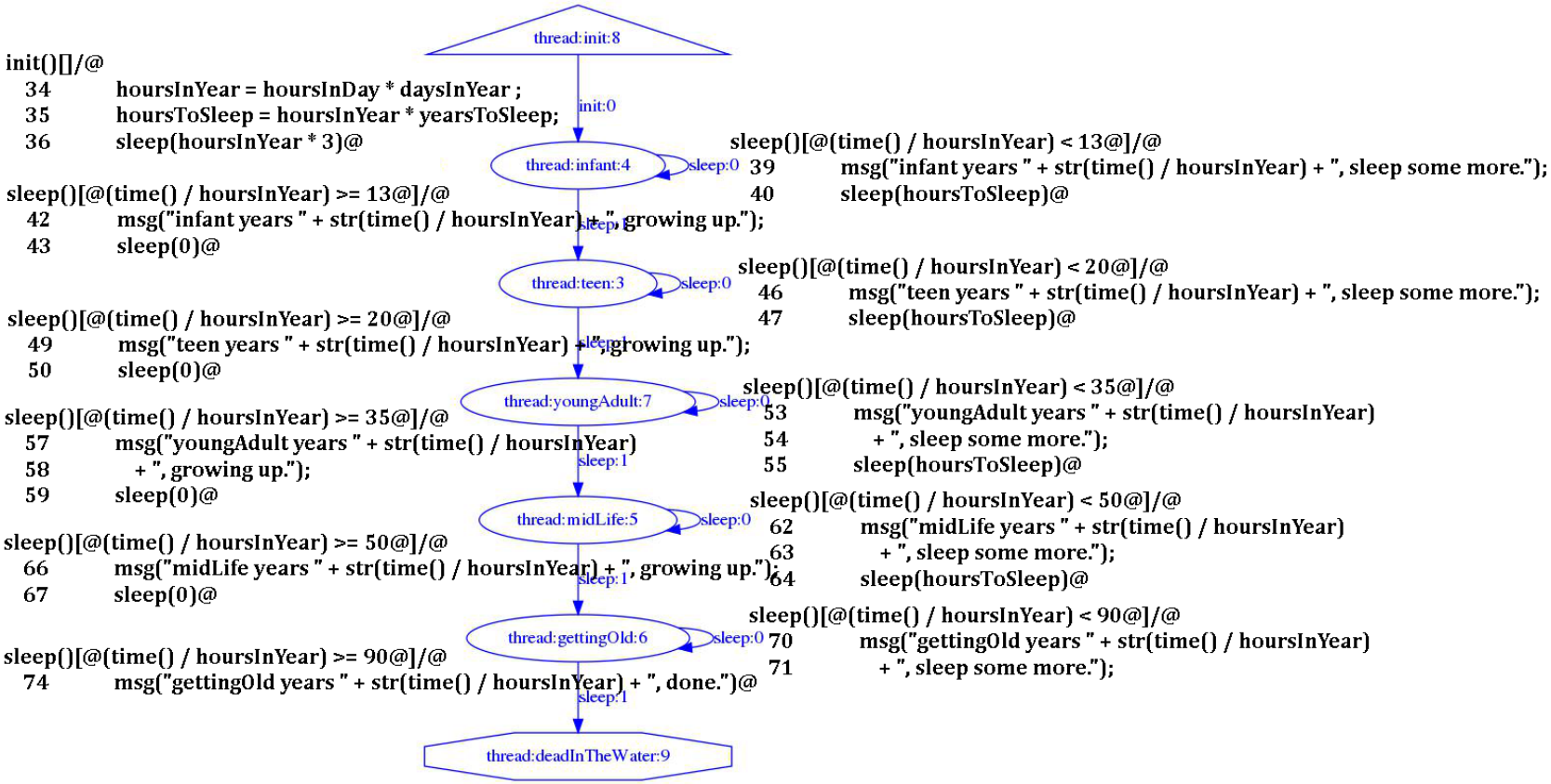
After compiling the above state machine on **mcgonagall**, you can run **make graphs** from within that directory on **acad**, which runs as follows.

$ **make graphs**
/bin/bash -c "/usr/bin/dot -Tjpeg Human1STM.dot > Human1STM.jpg"
/bin/bash -c "python graphcrunch.py f1 Human1STM_crunch MEAN_infant MEAN_teen MEAN_youngAdult MEAN_midLife MEAN_gettingOld"
mkdir $HOME/public_html
mkdir: cannot create directory ‘/home/kutztown.edu/parson/public_html’: File exists **# IGNORE THIS**
make: [graphs] Error 1 (ignored)
cp -p Human1STM.jpg *.png $HOME/public_html
chmod -R o+r+X $HOME/public_html
ls -l *png Human1STM.jpg

-rw-r--r--. 1 parson domain users 27852 Aug 21 15:10 Human1STM_crunch.png
-rw-r--r--. 1 parson domain users 62601 Aug 21 15:10 Human1STM.jpg
You can browse JPEGS and PNGS in http://acad.kutztown.edu/~parson
Directory http://acad.kutztown.edu/~YOURLOGINID then holds a JPEG file that graphs the state machines in the STM file. Here is http://acad.kutztown.edu/~parson/Human1STM.jpg:



Here is the **thread** STM annotated with the guards and actions of the original source code:


thread:init:8
init()[]/@
34 hoursInYear = hoursInDay * daysInYear ;
35 hoursToSleep = hoursInYear * yearsToSleep;
36 sleep(hoursInYear * 3)@
init:0
thread:infant:4
sleep:0
sleep()[@(time() / hoursInYear) >= 13@]/@
42 msg("infant years " + str(time() / hoursInYear) + ", growing up.");
43 sleep(0)@
sleep()[@(time() / hoursInYear) < 13@]/@
39 msg("infant years " + str(time() / hoursInYear) + ", sleep some more.");
40 sleep(hoursToSleep)@
sleep:1
thread:teen:3
sleep:0
sleep()[@(time() / hoursInYear) >= 20@]/@
49 msg("teen years " + str(time() / hoursInYear) + ", growing up.");
50 sleep(0)@
sleep()[@(time() / hoursInYear) < 20@]/@
46 msg("teen years " + str(time() / hoursInYear) + ", sleep some more.");
47 sleep(hoursToSleep)@
sleep:1
thread:youngAdult:7
sleep:0
sleep()[@(time() / hoursInYear) >= 35@]/@
57 msg("youngAdult years " + str(time() / hoursInYear)
58 + ", growing up.");
59 sleep(0)@
sleep()[@(time() / hoursInYear) < 35@]/@
53 msg("youngAdult years " + str(time() / hoursInYear)
54 + ", sleep some more.");
55 sleep(hoursToSleep)@
sleep:1
thread:midLife:5
sleep:0
sleep()[@(time() / hoursInYear) >= 50@]/@
66 msg("midLife years " + str(time() / hoursInYear) + ", growing up.");
67 sleep(0)@
sleep()[@(time() / hoursInYear) < 50@]/@
62 msg("midLife years " + str(time() / hoursInYear)
63 + ", sleep some more.");
64 sleep(hoursToSleep)@
sleep:1
thread:gettingOld:6
sleep:0
sleep()[@(time() / hoursInYear) >= 90@]/@
74 msg("gettingOld years " + str(time() / hoursInYear) + ", done.")@
sleep()[@(time() / hoursInYear) < 90@]/@
70 msg("gettingOld years " + str(time() / hoursInYear)
71 + ", sleep some more.");
sleep:1
thread:deadInTheWater:9

Here are a few important points about these STMS:

1. The actions take place as the machine enters the destination of a transition. By the time the destination state is entered, the actions have complete.
2. Typically, the final action step of a transition is a call to a library procedure that generates the event that, some time later, causes transition out of the destination state. In all of the above transitions within STM thread, the action call to library procedure sleep(TICKS) generates a sleep() event after TICKS simulated time steps.
3. In later projects one STM object will signal and event to a different STM object. In such cases it is a different STM object that generates an event leading out of a state.
4. STM.doc.txt in each handout directory is the documentation for the STM library.
5. If you get a run-time error while running a STM model the reports only a __codeTable__[INDEX], run decode.py in that project directory like this:

$ **./decode.py Human1STM.py 29**

__codeTable__[29] = compile('sleep(hoursToSleep)','nofile','exec'),

Running **make test** for this demo creates simulation out file **Human1STM.log** with these lines:

https://kuvapcsitrd01.kutztown.edu/~parson/Human1STM.log.txt

The first field is the simulation time in abstract **ticks**. The second field is MSG for msg() calls from the state machine (useful for debugging), and is LOG for automatic logging of state transitions. Next comes the unique ID of the logging STM object, and a record of event arrivals and subsequent state transitions.

If one of your STM simulations blows up, check the contents of its .log file for error messages. I recommend changing this line in the **makefile**:
SIMLOGLEVEL = 2
to this when you are debugging:
SIMLOGLEVEL = 3

Increasing the logging level slows the simulation down because it flushes every output call to the log file when your models sends output – we will discuss this slowing during the semester – but the reason to do it for debugging is because if the program crashes, unflushed output will not appear in the log file. Also, remember when you get a cryptic error message involving __codeTable__[INDEX], run decode.py in that project directory like this:

$ **./decode.py Human1STM.py INDEX**

My script file **crunchlog.py** analyzes **Human1STM.log** to create file **Human1STM.crunch** and compares it to reference file **Human1STM_crunch.ref**, which is the expected output for this simulation. If the measures of interest are within project tolerance defined in file **diffset.py**, then the tests pass. Otherwise, make test retorts an error and identifies the offending values. Here is **Human1STM.crunch.ref** for this demo.

https://kuvapcsitrd01.kutztown.edu/~parson/Human1STM_crunch.ref.txt

Below is the documentation for the library functions, procedures and methods available to your guards and actions in your STMs. This information appears in file **STM.doc.txt** in the project directory. We will go over these in class.

https://kuvapcsitrd01.kutztown.edu/~parson/STM.doc.txt