Appendix: Code for Federated Simulation Framework Implementation

Federation

```
from typing import List, Union
from simpy import Environment, Event, events
from simpy.core import SimTime, StopSimulation, EmptySchedule
from simpy.events import URGENT
from fedsim import message
from fedsim.messageEnvironment import MessageEnvironment
class Federation:
simulators: List[Environment]
def __init__(self):
self.simulators = []
def addSimulator(self, simulator: Environment):
self.simulators.append(simulator)
def initSimulator(self, simulator: Environment, until):
if until is not None:
if not isinstance(until, Event):
# Assume that *until* is a number if it is not None and
# not an event. Create a Timeout(until) in this case.
at: SimTime
if isinstance(until, int):
at = until
else:
at = float(until)
if at <= simulator.now:
raise ValueError(
f’until(={at}) must be > the current simulation
time.’
)
# Schedule the event before all regular timeouts.
until = Event(simulator)
until._ok = True
until._value = None
simulator.schedule(until, URGENT, at - simulator.now)
elif until.callbacks is None:
# Until event has already been processed.
return until.value
until.callbacks.append(StopSimulation.callback)
def run(self, until):
for simulator in self.simulators:
self.initSimulator(simulator, until)
try:
while True:
#self.simulators.map { it to it.peek() }.minBy { (_, t) ->
t }
# (nextToProcess, lowestTime) = min(lambda (_, t): t,
map(lambda (e): (e, simulator.peek()), self.simulators)
lowestTime = float(’inf’)
nextToProcess: Union[Environment, None] = None
for simulator in self.simulators:
simTime = simulator.peek()
if simTime < lowestTime:
lowestTime = simTime
nextToProcess = simulator
```

```
try:
eventValue = nextToProcess._queue[0][3].value
# receivedMessages = nextToProcess.stepWithMessages()
nextToProcess.step()
assert nextToProcess.now == lowestTime
# for receivedMessage in receivedMessages:
if isinstance(eventValue, message.Message):
for (dest, name) in eventValue.destinations:
if isinstance(dest, MessageEnvironment):
dest.sync_and_deliver_message(nextToProcess.now,
name, eventValue)
# d.sendMessage(eventValue)
except StopSimulation as exc:
self.simulators.remove(nextToProcess)
if len(self.simulators) == 0:
raise exc
except StopSimulation as exc:
return exc.args[0] # == until.value
except EmptySchedule:
if until is not None:
assert not until.triggered
raise RuntimeError(
f’No scheduled events left but "until" event was not ’
f’triggered: {until}’
)
return None
```

Message Class

```
from typing import List, Union
from simpy import Environment, Event
from simpy.events import NORMAL
class Destination:
env: Environment
name: str
def __init__(self, env: Environment, name: str) -> None:
super().__init__()
self.env = env
self.name = name
def __sizeof__(self) -> int:
return 2
def __getitem__(self, index):
if index == 0:
return self.env
elif index == 1:
return self.name
else:
raise IndexError()
class Message:
destinations: List[Destination]
def __init__(self, destination: Union[Destination,
List[Destination]]):
if isinstance(destination, Destination):
self.destinations = [destination]
else:
self.destinations = destination
def __repr__(self):
return self.__class__.__name__
```

```
class MachineMessage(Message):
machine_name: str
def __init__(self, destination_env: Environment, machine_name: str):
Message.__init__(self, Destination(destination_env,
machine_name))
self.machine_name = machine_name
def __repr__(self):
return f’{self.__class__.__name__}: {self.machine_name}’
class RequestRepair(Message):
machine_name: str
source: Environment
def __init__(self, destination: Environment, source: Environment,
machine_name: str):
Message.__init__(self, Destination(destination, machine_name))
self.machine_name = machine_name
self.source = source
def __repr__(self):
return f’{self.__class__.__name__}: {self.machine_name}’
class RepairFinished(MachineMessage):
def __init__(self, destination: Environment, machine_name: str):
MachineMessage.__init__(self, destination, machine_name)
class RepairStart(MachineMessage):
def __init__(self, destination: Environment, machine_name: str):
MachineMessage.__init__(self, destination, machine_name)
class MessageEvent(Event):
"""A :class:‘~simpy.events.Event‘ that is used for simple event
messages to be processed immediately.
This event is automatically triggered when it is created.
```

```
"""
def __init__(self, env: ’Environment’, message: Message):
Event.__init__(self, env)
self._value = message
self._ok = True
env.schedule(self, NORMAL)
# def _desc(self) -> str:
# """Return a string *Timeout(delay[, value=value])*."""
# return f’{self.__class__.__name__}({self.value})’
```

Message Environment

```
import logging
from typing import Dict, Optional
from simpy import Environment, Process
from simpy.core import SimTime
from fedsim.message import Message
class MessageEnvironment(Environment):
name: str
listeners: Dict[str, Process]
def __init__(self, name: str = None, initial_time: SimTime = 0):
Environment.__init__(self, initial_time)
self.name = name
self.listeners = {}
def register_listener(self, name: Optional[str], process: Process):
self.listeners[name] = process
def sync_and_deliver_message(self, time: SimTime, name:
Optional[str],
message: Message):
def delivery_process():
```

```
if self.now < time:

yield self.timeout(time - self.now)

listener = self.listeners.get(name)

if listener is None:

anonListener: Process

triggered: bool = False

for (_, anonListener) in filter(lambda x: (x[0] is None),

self.listeners.items()):

anonListener.interrupt(message)

triggered = True

if not triggered:

logging.warning(f"Could not deliver message: {message}

due to missing receiver")

else:

listener.interrupt(message)

self.process(delivery_process())

def send_message(self, msg, timeout = 0):

yield self.timeout(timeout, msg)

def __repr__(self):

return f’{self.__class__.__name__}: {self.name}’
```

config

```
import argparse

import __main__

import logging

import sys

from random import Random

LOG_CSV_FILE = f’{__main__.__file__.replace(".py", "")}.csv’

LOG_FILE = f’{__main__.__file__.replace(".py", "")}.log’
```

```
RANDOM_SEED = 42

PT_MEAN = 10.4 # Avg. processing time in minutes

PT_SIGMA = 2.5 # Sigma of processing time

MTTF = 300.0 # Mean time to failure in minutes

#BREAK_MEAN = 1 / MTTF # Param. for expovariate distribution

BREAK_MEAN = 1000 #Avg break machine mins

REPAIR_TIME = 30.0 # Time it takes to repair a machine in minutes

JOB_DURATION = 22.0 # Duration of other jobs in minutes

NUM_MACHINES = 10 # Number of machines in the machine shop

WEEKS = 1 # Simulation time in weeks

SIM_TIME = WEEKS * 7 * 24 * 60 # Simulation time in minutes

WAIT_TIMEOUT = 5000 # Timeout could be infinite, we are expecting a

REPAIRMAN1_SEED: int

REPAIRMAN2_SEED: int

ENV1_SEED: int

ENV2_SEED: int

def initRandom(seed: int):

rnd: Random = Random(RANDOM_SEED)

global REPAIRMAN1_SEED, REPAIRMAN2_SEED, ENV1_SEED, ENV2_SEED

REPAIRMAN1_SEED = rnd.randint(-sys.maxsize, sys.maxsize)

REPAIRMAN2_SEED = rnd.randint(-sys.maxsize, sys.maxsize)

ENV1_SEED = rnd.randint(-sys.maxsize, sys.maxsize)

ENV2_SEED = rnd.randint(-sys.maxsize, sys.maxsize)

initRandom(RANDOM_SEED)

def processArgs():

global LOG_FILE, WEEKS, LOG_CSV_FILE, REPAIRMAN1_SEED,

REPAIRMAN2_SEED, ENV1_SEED, ENV2_SEED

parser = argparse.ArgumentParser(’Simulated federation’)
```

```
parser.add_argument(’--logfile’, type=str, help="The file name to
use to store the log", default=LOG_FILE)
parser.add_argument(’--weeks’, type=int, help="The amount of weeks
to use", default=WEEKS)
parser.add_argument(’--seed’, type=int, help="The amount of weeks
to use", default=RANDOM_SEED)
args = parser.parse_args()
if args.logfile is not None:
LOG_CSV_FILE = args.logfile
LOG_FILE = args.logfile
with open(LOG_FILE, ’w’) as f: # empty out
pass
logging.basicConfig(format=’%(message)s’, filename=LOG_FILE,
level=logging.INFO)
if args.weeks is not None:
WEEKS = args.weeks
if args.seed is not None:
initRandom(args.seed)
return args
```

Machine Class

```
import logging
import random
import simpy
from fedsim import message
from fedsim.messageEnvironment import MessageEnvironment
from machineSim.config import WAIT_TIMEOUT
from machineSim.repairman import FederatedRepairMan, RepairMan,
NonFedRepairMan
```

```
DEFAULT_PT_MEAN = 10.4 # Avg. processing time in minutes
DEFAULT_PT_SIGMA = 2.5 # Sigma of processing time
#DEFAULT_BREAK_MEAN = 1 / MTTF # Param. for expovariate distribution
DEFAULT_BREAK_MEAN = 12345 #Avg break machine mins
DEFAULT_REPAIR_TIME = 30.0 # Time it takes to repair a machine in minutes
DEFAULT_JOB_DURATION = 22.0 # Duration of other jobs in minutes
class Machine(object):
env: MessageEnvironment
name: str
repairman: FederatedRepairMan
pt_mean: float
pt_sigma: float
break_mean: float
repair_time: float
job_duration: float
random: random.Random
#class Machine(object):
def __init__(self,
env: simpy.Environment,
name: str,
repairman: RepairMan,
randomSeed: int = random.randint(0, 1000),
pt_mean: float = DEFAULT_PT_MEAN,
pt_sigma: float = DEFAULT_PT_SIGMA,
break_mean: float = DEFAULT_BREAK_MEAN,
repair_time: float = DEFAULT_REPAIR_TIME,
job_duration: float = DEFAULT_JOB_DURATION,
logger: logging.Logger = None,
break_sigma: float = None,
):
self.env = env
self.name = name
self.repairman = repairman
self.parts_made = 0
self.numbroke = 0
self.days = 0
self.prior1 =0
self.pt_mean = pt_mean
self.pt_sigma = pt_sigma
self.break_mean = break_mean
if (break_sigma is None):
self.break_sigma = break_mean*0.25
else:
self.break_sigma = break_sigma
self.repair_time = repair_time
self.job_duration = job_duration
self.random = random.Random()
self.random.seed(randomSeed)
self.logger = logger
self.broken = False
# Start "working" and "break_machine" processes for this machine.
working = self.working()
self.manufacturingprocess = self.env.process(generator=working)
if isinstance(self.env, MessageEnvironment):
self.env.register_listener(self.name,
self.manufacturingprocess)
self.env.process(self.break_machine())
def log(self, message:str):
if (self.logger is None):
logging.info(message.replace('.0:',':'))
else:
self.logger.info(message.replace('.0:',':'))
def time_per_part(self):
"""Return actual processing time for a concrete part."""
return max(0, self.random.normalvariate(self.pt_mean,
self.pt_sigma))
def time_to_failure(self):
"""Return time until next failure for a machine."""
return max(0, self.random.normalvariate(self.break_mean,
self.break_sigma))
def working(self):
"""Produce parts as long as the simulation runs.
While making a part, the machine may break multiple times.
Request a repairman when this happens.
"""
while True:
# Start making a new part
while True:
```

```
try:
if self.broken:
yield self.env.timeout(WAIT_TIMEOUT, "still broken")
```

```
if self.broken:

self.log(f’M:{self.env.now}: still broken after

waiting for fix’)

else:

done_in = self.time_per_part()

# Working on the part

start = self.env.now

t, prio, eid, event = self.env._queue[0]

self.log(f’M:{self.env.now}: {self.name} - start

making part {self.parts_made+1}’)

yield self.env.timeout(done_in, "Make part")

self.parts_made += 1

self.log(f’M:{self.env.now}: {self.name} - finish

making part {self.parts_made}’)

done_in = 0 # Set to 0 to exit while loop.

except simpy.Interrupt as i:

done_in -= self.env.now - start # How much time left?

if (isinstance(i.cause, message.MachineMessage) and

i.cause.machine_name == self.name):

if isinstance(i.cause, message.RepairFinished):

self.broken = False

self.prior1 = self.prior1 + 1

self.log(f’M:{self.env.now}: Finished repairing

{self.name}’)

elif isinstance(i.cause, message.RepairStart):

self.log(f’M:{self.env.now}: Start repairing

{self.name}’)

elif i.cause == "Break machine" and not self.broken: #
```

```
triggered by failure
self.broken = True
self.log(f’M:{self.env.now}: Request repair of
{self.name}’)
# request =
message.RequestRepair(self.repairman.env,
```

```
self.env, self.name)
yield from self.repairman.request_repair(self)
# Part is done.
#df1.to_csv(’logtocsv.csv’, mode=’a’,float_format=’%.2f’)
def break_machine(self):
"""Break the machine every now and then."""
while True:
ttf = self.time_to_failure()
# self.log(f’M:{self.env.now}: expect {self.name} breakdown
after {ttf}’)
yield self.env.timeout(ttf, "Break machine if not broken")
# yield self.env.timeout(self.time_to_failure())
if not self.broken:
# self.log(f’M:{self.env.now}: Trigger machine being
broken #{self.numbroke}’)
# Only break the machine if it is currently working.
self.manufacturingprocess.interrupt("Break machine")
self.numbroke = self.numbroke+1
def other_jobs(self, repairman):
"""The repairman’s other (unimportant) job."""
```

```
while True:
# Start a new job
done_in = self.job_duration
while done_in:
# Retry the job until it is done.
# It’s priority is lower than that of machine repairs.
with repairman.request(priority=2) as req:
yield req
logging.info("Repairman is doing another job for %d
minutes" % (self.job_duration))
try:
start = self.env.now
yield self.env.timeout(done_in, "Done in")
done_in = 0
except simpy.Interrupt:
done_in -= self.env.now - start
```

Repairman Class

```
import logging
import simpy
import abc
import fedsim.messageEnvironment
import machineSim.machine
from machineSim import config
from fedsim import message
import random
from fedsim.messageEnvironment import MessageEnvironment
class RepairMan(metaclass=abc.ABCMeta):
name: str
```

```
resource: simpy.PreemptiveResource
random: random.Random
def __init__(self, name: str, resource: simpy.PreemptiveResource,
randomSeed: int = random.randint(0, 1000)):
self.name = name
self.resource = resource
self.random = random.Random()
self.random.seed(randomSeed)
def repair_time(self) -> int:
return 30
@abc.abstractmethod
def inform_repair_start(self, machineEnv: simpy.Environment,
machine: str):
"""This method is used to inform other elements (machines) that
repair of a machine has started.
@type machine: machineSim.machine.Machine
"""
pass
@abc.abstractmethod
def inform_repair_finished(self, machineEnv: simpy.Environment,
machine: str):
"""This method is used to inform other elements (machines) that
repair of a machine has finished.
"""
pass
@abc.abstractmethod
def request_repair(self, machine):
pass
```

```
def do_repair(self, request: message.RequestRepair):
machineEnv = request.source
machine = request.machine_name
# Use with to release the resource afterwards
with self.resource.request(priority=1) as r:
before_wait = self.env.now
yield r # Mutually exclusive
after_wait = self.env.now
if (before_wait!=after_wait):
self.do_log(f’R:{self.env.now}: {self.name} Waited for
repairman: {after_wait-before_wait}’)
yield from self.inform_repair_start(machineEnv, machine)
# yield self.env.timeout(0, message.RepairStart(machineEnv,
machine))
# logging.info(f’R:{self.env.now}: {self.name} Started
repairing {machine}’)
# yield self.env.timeout(self.repair_time(), "Repairing")
yield from self.inform_repair_finished(machineEnv, machine)
# yield self.env.timeout(self.repair_time(),
message.RepairFinished(machineEnv, machine))
# logging.info(f’R:{self.env.now}: {self.name} Finished
repairing {machine}’)
def do_log(self, message:str):
logging.info(message.replace(’.0:’,’:’))
class FederatedRepairMan(RepairMan):
env: MessageEnvironment
def __init__(self, name: str, env: MessageEnvironment, resource:
simpy.PreemptiveResource, randomSeed: int = random.randint(0,
1000)):
super().__init__(name, resource, randomSeed)
self.env = env
waitProcess = env.process(self.working())
self.env.register_listener(None, waitProcess)
def working(self):
while True:
try:
yield self.env.timeout(config.WAIT_TIMEOUT, "Waiting for
repair request...") # just repeated timeouts, we only
really care about interrupts
except simpy.Interrupt as i:
cause = i.cause
if isinstance(cause, message.RequestRepair):
self.do_log(f’R:{self.env.now}: {self.name} Received
repair request for {cause.machine_name}’)
self.env.process(self.do_repair(cause))
def request_repair(self, machine):
request = message.RequestRepair(self.env, machine.env,
machine.name)
yield machine.env.timeout(0, request)
def inform_repair_start(self, machineEnv:
fedsim.messageEnvironment.MessageEnvironment, machine: str):
yield from self.env.send_message(msg =
message.RepairStart(machineEnv, machine) )
self.do_log(f’R:{self.env.now}: {self.name} Started repairing
{machine}’.replace(’.0:’,’’))
def inform_repair_finished(self, machineEnv:
```

```
fedsim.messageEnvironment.MessageEnvironment, machine: str):
yield from self.env.send_message(msg =
message.RepairFinished(machineEnv, machine),
timeout=self.repair_time())
self.do_log(f’R:{self.env.now}: {self.name} Finished repairing
{machine}’)
pass
class NonFedRepairMan(RepairMan):
env: simpy.Environment
def __init__(self, name: str, env: simpy.Environment, resource:
simpy.PreemptiveResource, randomSeed: int = random.randint(0,
1000)):
super().__init__(name, resource, randomSeed)
self.env = env
def request_repair(self, machine):
request = message.RequestRepair(self.env, machine.env,
machine.name)
self.do_log(f’R:{self.env.now}: {self.name} Received repair
request for {machine.name}’)
yield from self.do_repair(request)
machine.broken = False
def inform_repair_start(self, machineEnv: simpy.Environment,
machine: str):
self.do_log(f’R:{self.env.now}: {self.name} Started repairing
{machine}’)
self.do_log(f’M:{self.env.now}: Start repairing {machine}’)
yield from list()
def inform_repair_finished(self, machineEnv: simpy.Environment,
machine: str):
yield self.env.timeout(self.repair_time())
self.do_log(f’R:{self.env.now}: {self.name} Finished repairing
{machine}’)
self.do_log(f’M:{machineEnv.now}: Finished repairing {machine}’)
```

Simple Federation

```
import logging
from fedsim.federation import Federation
from machineSim import createFederatedRepairman, createSimulator
from machineSim.config import REPAIRMAN1_SEED, REPAIRMAN2_SEED,
ENV1_SEED, ENV2_SEED, LOG_FILE, SIM_TIME, processArgs
args = processArgs()
repairman1 = createFederatedRepairman(REPAIRMAN1_SEED, "Repairman 1")
#repairman2 = createRepairman(RANDOM_SEED + 2, "Repairman 2")
(env1, _) = createSimulator(ENV1_SEED, repairman1, "env1")
# env2 = createSimulator(RANDOM_SEED+1, repairman1, "env2")
# singleEnv = createSimulator(RANDOM_SEED)
# singleRepairman = RepairMan("Johny fixit", singleEnv,
simpy.PreemptiveResource(singleEnv, capacity=1))
# singleEnv.process(singleRepairman.working())
#env1.run(until=SIM_TIME)
# singleEnv.run(until=SIM_TIME)
fed = Federation()
fed.addSimulator(env1)
# fed.addSimulator(env2)
fed.addSimulator(repairman1.env)
#fed.addSimulator(repairman2.env)
fed.run(until=SIM_TIME)
```

\paragraph{doubleFed.py}

```
import logging

import random

import sys

from fedsim.federation import Federation

from fedsim.messageEnvironment import MessageEnvironment

from machineSim import createFederatedRepairman, NUM_MACHINES,

Machine, BREAK_MEAN

from machineSim.config import REPAIRMAN1_SEED,

REPAIRMAN2_SEED,ENV1_SEED, SIM_TIME, processArgs

args = processArgs()

repairman1 = createFederatedRepairman(REPAIRMAN1_SEED, "Repairman 1")

#repairman2 = createFederatedRepairman(REPAIRMAN2_SEED, "Repairman 2")

env1 = MessageEnvironment("env1")

env2 = MessageEnvironment("env2")

rnd = random.Random(ENV1_SEED)

for i in range(NUM_MACHINES):

machineSeed = rnd.randint(-sys.maxsize, sys.maxsize)

logging.info(f’Creating Machine [{"env1"}] {0 + i} with seed

{machineSeed}’)

if i%2 == 0:

env = env2

else:

env = env1

Machine(env, f’Machine [{env.name}] {0 + i}’, repairman1,

break_mean=BREAK_MEAN, randomSeed=machineSeed)

# env2 = createSimulator(RANDOM_SEED+1, repairman1, "env2")

# singleEnv = createSimulator(RANDOM_SEED)
# singleRepairman = RepairMan("Johny fixit", singleEnv,
simpy.PreemptiveResource(singleEnv, capacity=1))
# singleEnv.process(singleRepairman.working())
#env1.run(until=SIM_TIME)
# singleEnv.run(until=SIM_TIME)
fed = Federation()
fed.addSimulator(env1)
fed.addSimulator(env2)
fed.addSimulator(repairman1.env)
#fed.addSimulator(repairman2.env)
fed.run(until=SIM_TIME)
```

Non Federation

```
import logging
import random
import csv
import os
from machineSim.config import PT_MEAN, PT_SIGMA, BREAK_MEAN,
RANDOM_SEED, LOG_CSV_FILE, SIM_TIME, NUM_MACHINES
import simpy
from fedsim import message
from machineSim.config import LOG_CSV_FILE, WAIT_TIMEOUT
from fedsim.messageEnvironment import MessageEnvironment
from machineSim.repairman import FederatedRepairMan
DEFAULT_PT_MEAN = 10.4 # Avg. processing time in minutes
DEFAULT_PT_SIGMA = 2.5 # Sigma of processing time
#DEFAULT_BREAK_MEAN = 1 / MTTF # Param. for expovariate distribution
DEFAULT_BREAK_MEAN = 1234 #Avg break machine mins
DEFAULT_REPAIR_TIME = 30.0 # Time it takes to repair a machine in
```

```
minutes
DEFAULT_JOB_DURATION = 22.0 # Duration of other jobs in minutes
class Machine(object):
env: MessageEnvironment
name: str
repairman: FederatedRepairMan
pt_mean: float
pt_sigma: float
break_mean: float
repair_time: float
job_duration: float
random: random.Random
#class Machine(object):
def __init__(self,
env: simpy.Environment,
name: str,
repairman: FederatedRepairMan,
randomSeed: int = random.randint(0, 1000),
pt_mean: float = DEFAULT_PT_MEAN,
pt_sigma: float = DEFAULT_PT_SIGMA,
break_mean: float = DEFAULT_BREAK_MEAN,
repair_time: float = DEFAULT_REPAIR_TIME,
job_duration: float = DEFAULT_JOB_DURATION,
logger: logging.Logger = None):
self.env = env
self.name = name
self.repairman = simpy.PreemptiveResource(self.env,capacity=1)
self.parts_made = 0
self.numbroke = 0
self.days = 0
self.prior1 =0
self.pt_mean = pt_mean
self.pt_sigma = pt_sigma
self.break_mean = break_mean
self.repair_time = repair_time
self.job_duration = job_duration
self.random = random.Random()
self.random.seed(randomSeed)
self.logger = logger
self.broken = False
# Start "working" and "break_machine" processes for this machine.
self.manufacturingprocess =
self.env.process(self.working(repairman))
#self.env.register_listener(self.name, self.manufacturingprocess)
self.env.process(self.break_machine())
def log(self, message:str):
if (self.logger is None):
logging.info(message)
else:
self.logger.info(message)
def time_per_part(self):
"""Return actual processing time for a concrete part."""
return max(0, self.random.normalvariate(self.pt_mean,
self.pt_sigma))
def time_to_failure(self):
"""Return time until next failure for a machine."""
```

```
return self.break_mean
def working(self, repairman):
"""Produce parts as long as the simulation runs.
While making a part, the machine may break multiple times.
Request a repairman when this happens.
"""
logging.basicConfig(format=’%(message)s’,
filename="nonfed1.log", level=logging.INFO)
while True:
# Start making a new part
done_in = self.time_per_part()
while done_in:
try:
# Working on the part
start = self.env.now
t, prio, eid, event = self.env._queue[0]
self.log(f’M:{self.env.now}: {self.name} - start making
part {self.parts_made + 1}’)
yield self.env.timeout(done_in, "Make part")
self.parts_made += 1
self.log(f’M:{self.env.now}: {self.name} - finish
making part {self.parts_made}’)
#yield self.env.timeout(done_in)
done_in = 0 # Set to 0 to exit while loop.
except simpy.Interrupt:
self.broken = True
done_in -= self.env.now - start # How much time left?
# Request a repairman. This will preempt its
```

```
"other_job".
with repairman.request(priority=1) as req:
yield req
self.log(f’M:{self.env.now}: Start repairing
{self.name}’)
yield self.env.timeout(DEFAULT_REPAIR_TIME)
self.prior1 = self.prior1 + 1
self.log(f’M:{self.env.now}: Finished repairing
{self.name}’)
self.broken = False
t, prio, eid, event = self.env._queue[0]
# write the header
# writer.writerow(df1)
# write the data
# df1 = df1.append({self.name, start, env.peek(),
env._queue[0], env.REPAIR_TIME}, ignore_index=True)
# Part is done.
self.parts_made += 1
def break_machine(self):
"""Break the machine every now and then."""
while True:
yield self.env.timeout(self.time_to_failure(), "Break machine
if not broken")
# yield self.env.timeout(self.time_to_failure())
if not self.broken:
# Only break the machine if it is currently working.
self.manufacturingprocess.interrupt("Break machine")
self.numbroke = self.numbroke+1
```

```
def other_jobs(self, repairman):
"""The repairman’s other (unimportant) job."""
while True:
# Start a new job
done_in = self.job_duration
while done_in:
# Retry the job until it is done.
# It’s priority is lower than that of machine repairs.
with repairman.request(priority=2) as req:
yield req
logging.info("Repairman is doing another job for %d
minutes" % (self.job_duration))
try:
start = self.env.now
yield self.env.timeout(done_in, "Done in")
done_in = 0
except simpy.Interrupt:
done_in -= self.env.now - start
env= simpy.Environment()
repairman = simpy.PreemptiveResource(env, capacity=1)
machines = [Machine(env, ’Machine %d’ % i, repairman)
for i in range(NUM_MACHINES)]
env.run(until=SIM_TIME)
```

Double Federation Example

```
import logging
import random
import sys
from fedsim.federation import Federation
from fedsim.messageEnvironment import MessageEnvironment
from machineSim import createFederatedRepairman, NUM_MACHINES,
Machine, BREAK_MEAN
from machineSim.config import REPAIRMAN1_SEED,
REPAIRMAN2_SEED,ENV1_SEED, SIM_TIME, processArgs
args = processArgs()
repairman1 = createFederatedRepairman(REPAIRMAN1_SEED, "Repairman 1")
#repairman2 = createFederatedRepairman(REPAIRMAN2_SEED, "Repairman 2")
env1 = MessageEnvironment("env1")
env2 = MessageEnvironment("env2")
rnd = random.Random(ENV1_SEED)
for i in range(NUM_MACHINES):
machineSeed = rnd.randint(-sys.maxsize, sys.maxsize)
logging.info(f’Creating Machine [{"env1"}] {0 + i} with seed
{machineSeed}’)
if i%2 == 0:
env = env2
else:
env = env1
Machine(env, f’Machine [{env.name}] {0 + i}’, repairman1,
break_mean=BREAK_MEAN, randomSeed=machineSeed)
# env2 = createSimulator(RANDOM_SEED+1, repairman1, "env2")
# singleEnv = createSimulator(RANDOM_SEED)
# singleRepairman = RepairMan("Johny fixit", singleEnv,
simpy.PreemptiveResource(singleEnv, capacity=1))
# singleEnv.process(singleRepairman.working())
#env1.run(until=SIM_TIME)
# singleEnv.run(until=SIM_TIME)
```

```
fed = Federation()
fed.addSimulator(env1)
fed.addSimulator(env2)
fed.addSimulator(repairman1.env)
#fed.addSimulator(repairman2.env)
fed.run(until=SIM_TIME)
```

\paragraph{requirements.txt}

```
numpy==1.21.4
pandas==1.3.4
python-dateutil==2.8.2
pytz==2021.3
simpy==4.0.1
six==1.16.0
progressbar~=2.5
tqdm~=4.62.3
```