ManPy Documentation

Table of Contents

1 Introduction and Scope

ManPy stands for "Manufacturing in Python" and it is a layer of Discrete Event Simulation (DES) objects built in SimPy (http://simpy.sourceforge.net/). The current version of ManPy is based on SimPy2 (http://simpy.sourceforge.net/old/). This happens because at the time ManPy implementation progressed, the newest version was not available. We plan to progress to SimPy3 (http://simpy.readthedocs.org/en/latest/) soon. This is not supposed to affect in a great extend this documentation.

The scope of the project is to provide simulation modellers with a collection of open-source DES objects that can be connected like "black boxes" in order to form a model. This collection is desired to be expandable by giving means to developers for:

- customizing existing objects by overriding certain methods
- adding brand new objects to the list

ManPy is product of a research project funded from the European Union Seventh Framework Programme (FP7-2012-NMP-ICT-FoF) under grant agreement n° 314364. The project name is DREAM and stands for *"Simulation based application Decision support in Real-time for Efficient Agile Manufacturing"*. More information about the scope of DREAM can be found at http://dream-simulation.eu/.

DREAM is a project which kicked off in October of 2012 and finishes in September of 2015. ManPy is an ongoing project and we do not claim that it is complete or bug-free. The platform will be expanded and validated through the industrial pilot cases of DREAM. Nevertheless, it is in a quite mature state to attract the interest of simulation modellers and software developers.

The dream repository contains the following 3 folders:

- **platform**: contains code for a GUI that is being build for ManPy. This is a parallel work and it is not always synchronized to ManPy's latest version
- **simulation**: contains all the simulation ManPy code along with some input files and some files from a commercial simulation package that are used for verification
- **test**: contains unit-tests for the project.

This document regards ONLY the ManPy part of the project. Note that ManPy is independent from the GUI and can be used separately as a library of simulation objects, which can be used to form a model. Users can implement alternative methods to be able to construct models, run them and get results.

The reader of this documentation needs to have basic, yet not deep knowledge of programming in Python (http://www.python.org/) and SimPy2. Also the reader is expected to have a basic knowledge of the Discrete Event Simulation (DES) technique.

2 How to get started

To be able to run the documentation examples just copy the dream folder to your Python folder. Then you can import ManPy objects as it is written in the examples, e.g.:

- *from dream.simulation.Queue import Queue* or
- *from dream.simulation.imports import Machine, Source, Exit*

ManPy uses the following Python libraries which need to be installed in order to run the examples:

- SimPy2
- NumPy
- xlrd
- xlwt

3 Architecture

ManPy objects are written exclusively in Python and they use methods of SimPy. Figure 1 shows the current state of the architecture.

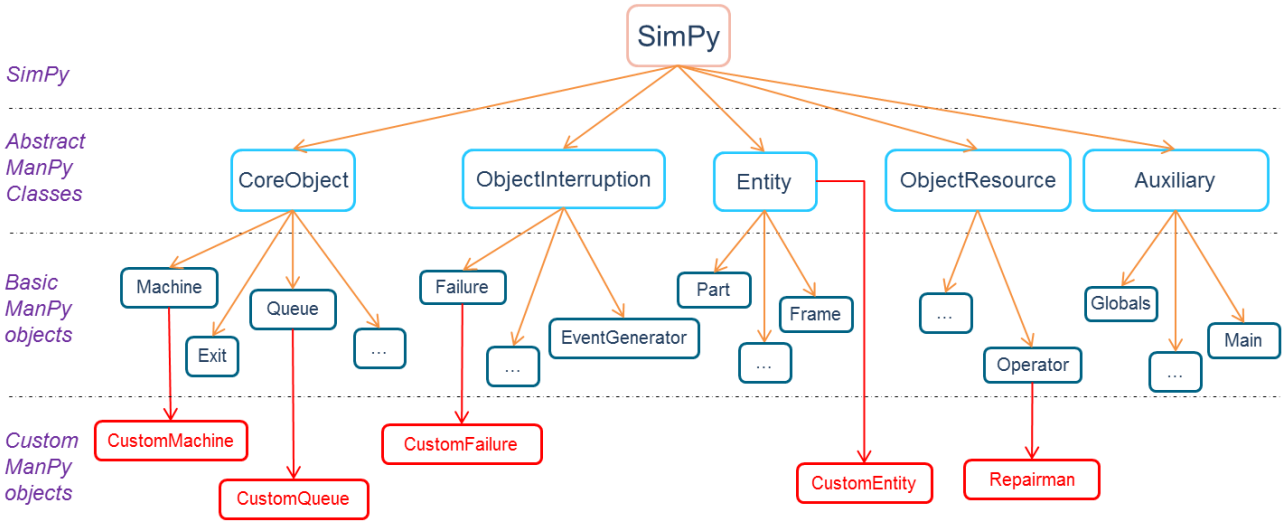

Figure 1: The ManPy class hierarchy

In Figure 1 four different layers are depicted:

- On the top we have SimPy classes
- The top layer of ManPy is a set of generic, abstract classes. There are not supposed to have instances, nevertheless they are important because:
    - they help in the grouping of objects
    - generic methods are defined for all those classes which the simulation objects inherit and override
- Below the generic objects lies the basic core of ManPy objects. This is currently being populated and expanded.
- On the bottom we have custom objects of ManPy. These inherit from one object of the basic core and customize it according to the needs of the modeller

In the remaining of this chapter the generic classes of ManPy will be described.

3.1 ManPy Generic (Abstract Classes)

The layer of abstract classes is the “heart” of ManPy. These give the basic guidelines of how the platform is structured. Note that since this is an ongoing work, the names of the classes may change, since we currently think towards the best abstraction. Also new generic classes might be added in future versions, even though the number should be kept reasonably short. The abstract classes include:

- **CoreObject**: all the stations in a model that are permanent for the model. These can be servers or buffers of any type.
- **ObjectInterruption**: all the objects that can affect the availability of another object. For example failures, scheduled breaks, shifts etc.
- **Entity**: all objects that get processed by or wait in CoreObjects and they are not permanent in a model. For example parts in a production line, customers in a shop, calls in a call centre etc.

- **ObjectResource**: all the resources that might be necessary for certain operation of a CoreObject. For example repairman, operator, electric power etc. An ObjectResource is necessary in modelling when two or more CoreObjects compete for the same resource (e.g. two machines competing for the same operator).
- **Auxiliary**: These are auxiliary classes that are needed for different simulation functionalities. Unlike the other categories described here, auxiliary classes do not inherit from one parent class, even though it is depicted in such a way in Figure 1 for reasons of coherence.

In the following subsection each category of generic classes will be described in more depth.

3.1.1 CoreObject

As CoreObjects are categorized all the stations in a model that are permanent for the model. These can be servers or buffers of any type. It is in the philosophy of ManPy that the CoreObjects will handle most of the simulation logic, so that a more generic process oriented approach is achieved.

CoreObjects should be able to communicate no matter what their type is. For example, a Machine should be able to retrieve an Entity from another CoreObject, using the same code, no matter if this CoreObject is a Queue or also a Machine. For this reason all the CoreObjects implement a set of methods which have the same name, but different implementation for every object. This set of methods includes:

i. **Definition methods**: are used for the instantiation of the object
ii. **Signalling methods:** Handle the communication between CoreObjects
iii. **Transaction methods**: are used to define how the objects exchange entities
iv. **State Control methods**: are used to retrieve the state of an object
v. **Flow Control methods:** used to control the flow of information and more specifically route the entities through the network of objects
vi. **Supplementary methods**: are used to define certain objects in Transactions or Control methods
vii. **Output and calculation methods**: are used either to output results or trace in different formats or to make certain calculations
viii. **Main simulation method**: one or more methods that are used to control the progress of the object in the simulated time. In Python terms this is a generator method. In SimPy2 there could be only one generator method where the *yield* commands of SimPy can be invoked while in SimPy3, there can be more than one generator method for each instance of a class.

Also, CoreObjects share some conventions for certain variables listed in the following subsection.

3.1.1.1 CoreObject Attributes

The most important attributes that all the objects share are given bellow:

- *Res*: this is an instance SimPy.resource type. It keeps the Entities that the CoreObject holds in its users (SimPy naming) list, to which we also refer to as the “internal queue” of the core object in this documentation (not to be confused with the Queue object).
- *next*: a list that holds all the successors of the CoreObject, i.e. the CoreObjects to which the object can give an Entity.
- *previous*: a list that holds all the predecessors of the CoreObject, i.e. the CoreObjects from which the object can receive an Entity.
- *receiver:* a variable that holds the successor object that the CoreObject gives an Entity at a given moment off simulation time.
- *giver:* a variable that holds the predecessor object from which the CoreObject receives an Entity at a given moment off simulation time

Note that *next* and *previous* lists may be empty. This can happen for several reasons:

- For certain objects it is not logical to have both lists. For example an Exit object should not have any successors
- Sometimes the flow is completely dependent on Entities attributes (e.g. in a jobshop). In such cases objects do not need to have predecessors or successors.

All objects also share variables that allow avoiding conflicts when signalling of a receiver or of a giver may be performed at the same simulation time by multiple givers/receivers respectively:

- *exitAssignedToReceiver*: a variable that shows whether the exit of station is reserved for a specific successor for the current simulation time. An instance of a buffer may be able to perform more than one transaction at a certain simulation time. To avoid signalling a receiver (or being signalled by a station) and disposing the entity in question to a different successor station, the exit of a giver is reserved for the current simulation time till the transaction is concluded.
- *entryAssignedToGiver*: Similarly to above, a variable that shows whether the entry of an object is reserved for a specific giver for the current simulation time.

CoreObjects share also several other attributes that hold certain important values. For example *timeLastEntityEntered* holds the simulation time that an Entity entered in the CoreObject. Also they have counters that hold certain results. For example *totalFailureTime* holds the failure time for a CoreObject, which can be divided by the length of the simulation run in order to give the percentage of time that the CoreObject was in failure. An API list of such variables is currently populated.

In addition, CoreObjects have a common arsenal of signals which allow them to communicate with each other whenever needed. These set of signals include the following:

- *isRequested*: A signal that informs the object that it is requested from a predecessor to take part in a transaction. After an object receives such a signal, it can proceed with getting an entity.
- *canDispose*: A signal that shows an object that it can proceed with disposing an entity to the receiver sending the signal. The object receiving that signal can try to signal a receiver.
- *interruptionStart*: A signal showing that an interruption on the object receiving it has started.
- *interruptionEnd*: A signal communicating the end of the interruption.
- *entityRemoved*: Whenever an object is in position to dispose of an entity signals a receiver. This object should then wait until the receiver resumes control and removes the entity from it, before it can wait to receive another entity. This applies to objects that perform certain processing on the entities such as a station.

Apart from the signals described above, there are other object specific signals used to pass the control among the objects according to a state change of another object. For example, *loadOperatorAvailable* is used to signal a station that a resource needed for the loading of the station became available. Furthermore, *preemptQueue* is used to inform a station that it must preemptied.

Below we will discuss the methods of each of the 7 categories of generic methods.

3.1.1.2 Definition methods

These are used for the instantiation of the object. 3 such methods exist:

- *__init__*: this is the python constructor method. This method is ran when the instance is created.
- *initialize*: this method initializes the object for a simulation replication. It should not be confused with the constructor above. The constructor is ran only in the creation of the object, while initialize must be ran in the beginning of every replication.
- *defineRouting*: it defines the *next* and *previous* lists, i.e. successor and predecessor objects.

3.1.1.3 Signalling methods

These methods handle the communications between CoreObjects. In every communication/signalling two CoreObjects are involved. The *giver* and the *receiver* object of the transaction (see 3.1.1.4) are decided before the signalling takes place. The *giver* object is the one that gives the entity and the *receiver* object is the one that obtains it.

- *signalReceiver*: it is ran by a possible *giver* object whenever the state of the object changes to being able to dispose an entity. Within the method the possible receivers are defined, and among them one candidate receiver is chosen and finally signalled to perform the transaction. Before the communication takes place, the entry of the receiver and the exit of the giver are reserved to each other. Returns true only when the signal was successfully sent.
- *signalGiver*: it is ran by a possible receiver object whenever its state changes to being able to receive an entity. The possible givers are pinpointed and one is selected for the transaction. The chosen giver of the two-side transaction after resuming the control will perform the *signalReceiver* method, checking again the possible receivers and locking the transaction as described in the previous (*signalReceiver* method). Returns true only when the signal was successfully sent.

3.1.1.4 Transaction methods

These handle the transactions of Entities between CoreObjects. In every transaction two CoreObjects take part after the signalling is concluded. Two such methods exist:

- *removeEntity*: it is ran on the giver object and it removes an Entity from it. The objects sort the Entities they hold in such a way, so that the object that will be removed is the first object of the internal queue.
- *getEntity*: it is ran on the receiver object and it obtains an Entity from the giver. In essence it calls the *removeEntity* method of the giver object and adds the Entity to its internal queue. Within the *getEntity* method further controls are performed concluding the transaction that started with the signalling between the *receiver* and the *giver*. The exit and the entry of the *giver* and the *receiver* are unblocked respectively. The *next* list is updated if needed by the *updateNext* method etc.

3.1.1.5 State Control methods

For every object they provide information about its state. They return true or false. 3 basic such methods exist:

- *canAccept*: returns true if the object is in a state to receive an Entity. The logic depends on the type of the object. For example in a Queue the capacity might need to be checked, while an Exit object might always be in the state of receiving an Entity. Note that sometimes it is needed that this method should return true only to the object that it can receive the Entity from. In this case, the object that calls the method must be passed as an argument.
- *haveToDispose*: returns true if the object is in a state to give an Entity. The logic depends on the type of the object. For example a Queue may need to check only if it does hold one or more Entities, while a Machine might need to check also if the Entity that it holds has ended its processing. Note that sometimes it is needed that this method should return true only to the object that it can give the Entity to. In this case, the object that calls the method must be passed as an argument.
- *canAcceptAndIsRequested*: returns true only when both conditions are satisfied: the object is in the state to accept an Entity and also another object is requesting to give one Entity to it. As we will see, only when this method returns true a transaction (and signalling) between two objects can take place. Note that contrary to the other methods described in this section, this one is expected to be called on a receiver object only by the signalling methods signalReceiver/signalGiver after the giver-receiver pair is chosen. Thus, as it is always run on the possible receiver object, the giver object must always be provided as an argument.

In addition, there exist a number of supplementary control methods returning simple information on the state of an object.

- *activeQueueIsEmpty*: returns true if the internal queue of an object is empty.
- *checkIfActive*: returns true if the object is in active state. Note that an object can be in two states, active or inactive due to a failure or a different kind of interruption, e.g. shift or maintenance.
- *isInActiveQueue*: takes as argument an Entity and returns true if the Entity is in the internal queue of the object.

3.1.1.6 Flow Control methods

The methods described here are used to control the flow of information and more specifically route the entities through the network of objects:

- *findGiversFor*: is a static method that takes as argument the object requesting a list of objects that can dispose an Entity to the Object requesting the list. After invoking the *haveToDispose* method of the *previous* objects, it returns the list of objects that are in position to dispose entities and have not already been signalled to give the current simulation time.
- *selectGiver*: is a static method that chooses from a list of possible givers provided as argument to the method, and returns the object that has been waiting the most time. The logic can be altered and return different object as *giver* according to the needs of a specific case.
- *findReceiversFor*: is a static method that takes as argument the object requesting a list of objects that can receive the Entity the requesting Object holds. After invoking the *canAccept* method of the *next* objects, it returns the list of objects that can accept and have not already been signalled to receive the current simulation time.
- *selectReceiver*: static method which chooses from a list of possible receivers provided as argument to the method, and returns the object that has been waiting the most time. The logic can be altered and return different object as *receiver* according to the needs of a specific case.

Additional methods that are used to update the status of the objects and their attributes used to identify the route of the entities that they hold, are described below:

- *updateNext*: updates the *next* list of an object according the route of the entity received last. If the entity has no specified route then method performs nothing.
- *assignExitTo/assignEntryTo*: assigns the exit and the entry of an object according to the description given in 3.1.1.11.
- *unAssignExit/unAssignEntry*: unblocks the exit and the entry respectively following the convention of assignExit/EntryTo.
- *exitIsAssignedTo/entryIsAssignedTo*: returns the object the exit/entry of an object is assigned to.
- *preempt*: whenever pre-emption needs to be performed, the *next* and *previous* lists need to be updated as well as the route of the entity preemptied (if any). A signal *preemptQueue* is also sent to the object that needs to perform the pre-emption.

3.1.1.7 Supplementary methods

These methods are used to obtain specific objects that are needed for the transaction and control methods. Six such methods exist:

- *getActiveObject*: returns the active object in the transaction. This always returns *self*, and they can be used interchangeably (though *self* should be faster since it does not call a method).
- *getActiveObjectQueue*: returns the internal queue of the active object. This always returns *self.Res.activeQ*, but it may be preferred to use the method since it makes the code cleaner and lesser need of knowledge of the internals of ManPy is achieved.
- *getGiverObject*: returns the giver object in a transaction.

- *getGiverObjectQueue*: returns the internal queue of the giver object in a transaction.
- *getReceiverObject*: returns the receiver object in a transaction.
- *getReceiverObjectQueue*: returns the internal queue of the receiver object in a transaction.

3.1.1.8 Output and calculation methods

Perform calculations or output data. Nine such methods exist:

- *sortEntities*: it sorts the Entities in the internal queue of the CoreObject. Many times this method might not be needed. However, there are times when it is essential. E.g. when a Queue needs to sort its Entities according to a predefined rule.
- *calculateProcessingTime:* Calculates the processing time every time one Entity gets into the CoreObject for processing.
- *interruptionActions*: performs actions that are carried out whenever an object is interrupted (applies mainly to Stations that perform processing on the entities).
- *postInterruptionActions*: performs actions that must be carried out after the *interruptionEnd* signal is received.
- *endProcessingActions*: actions that are performed after the processing of an Entity ends.
- *postProcessing*: is called for every object in the end of a simulation replication, The purpose is to perform certain calculations. For example, if a Machine is still processing an Entity when the simulation ends, this processing time should be added so that the results are accurate. Note that when an object is complex, sometimes it is difficult to debug such a method. On the other hand, in a long simulation run a mistake in this method would most probably not introduce a large error.
- *outPutResultsJSON*: outputs the results of the object in a JSON format. All the objects output to the same JSON file. If we have more than one replications, the results are given in confidence intervals.
- *outPutResultsXL*: outputs the results of the object in an Excel file. All the objects output to the same Excel file. If we have more than one replications, the results are given in confidence intervals. To save the excel file the user should add *G.outputFile.save("filename.xls")* in the main script.
- *outPutTrace*: outputs trace in an Excel sheet when an important event happens (e.g. an Entity gets into the CoreObject). All the objects output to the same Excel file and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to be verified, it should be turned off since it slows the program significantly.

Note that contrary to the other methods described in this section, these methods are expected to be called only internally from an object or from a main script (e.g. there is no need for a CoreObject to call *outPutResultsJSON* of another). So it is not obligatory that the name is the same for all CoreObjects. Nevertheless, for reasons of coherence these methods are mentioned here.

ManPy users are invited to write new methods for objects, in case they desire to output results in different format (e.g. XML). Also it is logical that ManPy users would like to override these methods to customize the results that they get.

3.1.1.9 Main simulation method

Here the logic that the CoreObject follows as it evolves through time. There is only one such method:

- *run*: this is a generator method and it is the main method where the *yield* commands of SimPy can be used. For this reason *run* requires that the user knows the internals of SimPy in order to customize. It is common (but not obligatory) that in such a method there is a *while* loop that runs all through the simulated time. The logic followed in every CoreObject's run method is:

1. Wait for an *isRequested* or other type of signal that informs the object that it is now permitted to proceed with getting an entity from a predecessor defined as its *giver.*
2. If the signal (other than *isRequested*) is not sent by the *giver* but another object, then a possible giver must be signalled before any other action. Step 1 is then repeated again.
3. Call object's *getEntity* method so that it obtains the Entity from the *giver* object.
4. Carry on the logic of the object (unique for every different type).
5. When the process is ended the object tries signalling a receiver by the use of *signalReceiver* method.
6. If the signalling was not successful, the object should wait till it receives a *canDispose* signal.
    a. After receiving a *canDispose* signal, the control tries to signal the receiver. In the case of objects that perform certain processing on the entity it holds (e.g. Machine), the object must wait till it receives an acknowledgement (*entityRemoved* signal) from the receiver, before it proceeds with receiving a new entity.
7. Repeat step 1. The loop cannot start again if it should not, since step 1 takes care of it. When at some point some receiver object sends an *isRequested* signal this may change and the loop will restart.

In the case of objects that perform no processing on the entities they hold (e.g. Queue); steps 1 and 6 can be combined on the same yield command. Note that a buffer can wait for any of the two *canDispose* or *isRequested* as long as it can simultaneously dispose and receive more than one entity, given the capacity is not exceeded.

There can be object classes that have auxiliary generator methods. These methods perform specific controls and make use of yield commands whenever they take control of program flow. Though, *run* must always be considered the main generator method that is invoked from the main script, while others are invoked by the object internally.

3.1.2 ObjectInterruption

As ObjectInterruptions are categorized all the objects that can affect the availability of another object. For example failures, scheduled breaks, shifts etc.

The most important attribute of an ObjectInterruption is *victim* which is the CoreObject whose the availability the ObjectInterruption handles. This CoreObject is also the one that creates and activates the instance of the ObjectInterruption object.

Currently there are six generic methods for these objects:

- *outPutTrace*: outputs trace in an Excel sheet when an important event happens (e.g. a Machine gets a failure). All the ObjectInterruptions output to the same Excel file as the CoreObjects and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to be verified, it should be turned off since it slows the program significantly.
- *getVictimQueue*: returns the internal queue of the victim CoreObject.
- *invoke*: signals the interruption object. After the receiving the signal *isCalled* the generator of the interruption object performs the requested actions.
- *interruptVictim*: signals (*interruptionStart*) the *victim* of the interruption object that an interruption is just started.
- *reactivateVictim*: informs the *victim* that all the interruption related commands are carried out and that the control should now return to the victim.
- *run*: this is a generator method and it is the only one where the *yield* commands of SimPy can be used. For this reason *run* requires that the user knows the internals of SimPy in order to customize. Generally the victim CoreObject is the one that activates the ObjectInterruption, but this is not obligatory. It is common (but not obligatory) that in such a method there is a *while*

loop that runs all through the simulated time. The logic followed in an ObjectInterruption's *run* method is:

1. Hold until an interruption should happen or until it is signalled by the object to be interrupted or any other object.
2. carry out the logic of the interruption:
    a. Passivate (*interruptVictim*) the victim,
    b. Hold until the interruption should be stopped or perform the requested actions (e.g. request a resource),
3. Reactivate (*reactivateVictim*) the victim
4. Restart the loop

3.1.3 Entity

As Entities are categorized all objects that get processed by or wait in CoreObjects and they are not permanent in a model. For example parts in a production line, customers in a shop, calls in a call centre etc.

Entities can get into the model from a Source type CoreObject or be set as Work In Progress (WIP) at the start of the simulation run. They hold certain general attributes such as *creationTime* that holds the time that the Entity entered the model.

In alignment with the philosophy of having the CoreObjects handling most of the simulation logic, Entities are kept reasonably simple. This is also efficient for models that we may have few CoreObjects (e.g. 3 Queues and 3 Machines) but thousands of Entities (e.g. the production Parts of one month). So the CoreObjects handle how the Entities move and evolve through simulated time. Of course it is possible that certain properties (such as routing or processing time needed) may be kept in an Entity's attributes, which the CoreObject will read.

Currently there are two generic methods for these objects:

- *outPutResultsJSON*: outputs the results of the object in a JSON format. All the Entities output to the same JSON file as the CoreObjects.
- *initialize*: initializes the Entity at the start of each replication.

3.1.4 ObjectResource

As ObjectResource are categorized all the resources that might be necessary for certain operation of a CoreObject. For example repairman, operator, electric power etc. An ObjectResource is necessary in modelling when two or more CoreObjects compete for the same resource (e.g. two machines competing for the same operator).

CoreObjects handle how the ObjectResources move and evolve through simulated time. Of course it is possible that certain properties may be kept in an ObjectResource's attributes, which the CoreObject will read.

One important attribute if the ObjectResource is *Res*. *Res* is an instance SimPy.resource type and it allows other objects to request or release the resource (SimPy *yield.request* and *release* respectively).

ObjectResource implements the following methods:

- *postProcessing:* Same functionality with CoreObject method with the same name
- *outPutResultsJSON:* Same functionality with CoreObject method with the same name
- *outPutResultsXL:* Same functionality with CoreObject method with the same name
- *outPutTrace:* Same functionality with CoreObject method with the same name
- *initialize:* Same functionality with CoreObject method with the same name
- *checkIfResourceIsAvailable*: returns true if there is one or more available units of the ObjectResource.

- *getResource*: returns the resource (*self.Res*)
- *getResourceQueue*: returns the activeQueue of the resource (*self.Res.users*)

workingStation is another important attribute of the Operator ObjectResource. It is the station where the resource is currently occupied.

3.1.5 Auxiliary

These are auxiliary classes that are needed for different simulation functionalities. Unlike the other categories described here, auxiliary classes do not inherit from one parent class, even though it is considered a good practice that they are grouped and presented here for reasons of coherence.

Three categories of auxiliary classes exist currently in ManPy.

- *G*: contains global variables for the simulation such as the length of the simulation run, the number of the simulation replications etc. *G* can be imported with the line *from Globals import G.* Some important conventions:
    - G.ObjList is a list that should hold all the CoreObjects.
    - G.maxSimTime is a float that defines the length of the simulation run.
    - G.seed is an integer that holds the seed for random number generation.
- *RandomNumberGenerator:* contains methods to create random variables that follow certain distributions. In the current version of ManPy only a few distributions listed below are supported, but this is to be expanded:
    - Fixed
    - Exponential
    - Normal
    - Erlang
- *MainScript*: as main script we name every script (it is not necessary a class) that reads a ManPy simulation model, creates it, runs it and returns the results. The input and output can be of whatever form. There are currently two different main scripts *LineGenerationJSON* and *LineGenerationCMSD* that read the data using different formats. Also, all the examples demonstrated in the next section are main scripts. Nevertheless, it is desired that users can implement and use different main scripts according to their needs. A main script should perform the following operations:
    1. Read or define the objects
    2. Create the objects
    3. Define the structure and set the topology of the model (predecessors and successors) if needed
    4. In every replication:
        i. initialize the simulation (SimPy.*initialize*)
        ii. initialize CoreObjects, ObjectResources and Entities
        iii. set the WIP if needed
        iv. activate the objects
        v. run the simulation (SimPy.simulate)
        vi. call postProcessing method of the objects
    5. After the simulation is over output the results in a desirable way

3.2 Expanding the Code

In the last subsection the architecture, generic methods and the logic of ManPy were described. Understanding the above, it should be possible for someone to make a new object of any of the 5 categories described and incorporate it into the platform. New objects may be:

- Customized objects that inherit from an existing one and override certain methods
- Completely new objects, that implement their versions of the methods

In order to reduce the learning curve, it is desired that ManPy keeps the set of methods as short as possible. However, adding a new generic method in a new object is also possible. Let's suppose for example that a CoreObject named *newCoreObject* requires having *newCoreObjectMethod* that will also be called by other objects in the model. Then the developer can implement the version for *newCoreObjectMethod* that he wishes for the *newCoreObject,* but he should also add an empty version of the method to the parent object. So in CoreObject the following should be added:

```
def newCoreObjectMethod(self):
    pass
```

In case *newCoreObjectMethod* requires arguments, they should be defined as optional. In this way the method can be called for every CoreObject without causing the code to crash and the objects can still interact as black boxes.

In the next section examples of how to construct, customize and run a ManPy model will be given.

4 Examples

4.1 A single server model

The first example shown here is a simple model of a production line that consists only from a point of entry (Source) one server (Machine) and a point of exit (Exit). A graphical representation of the model is shown in Figure 2(Note, Figure 2 and other figures in this section, are printscreens from the DREAM GUI. They are presented here for convenience, in order to make the text more understandable. This documentation is specific for ManPy and does NOT cover the DREAM GUI).

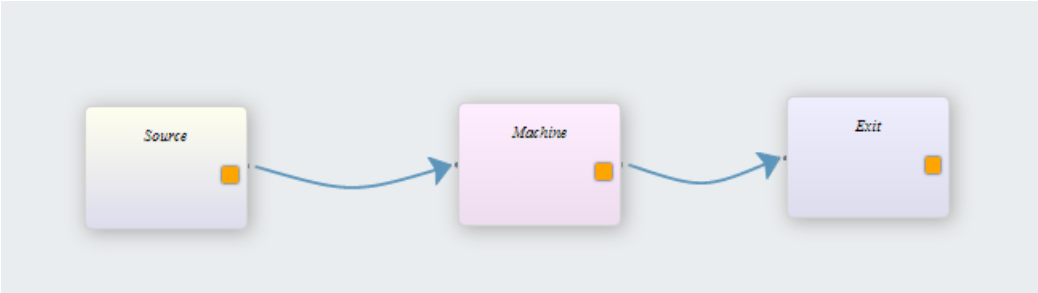

Figure 2: Single server model

As values we have the following:

- The source produces parts. One part is produced every 30 seconds
- The Machine processes one part at a time. The processing time is 15 seconds
- We want to study the system for 24 hours

Below is the ManPy main script to run this model (dream\simulation\Examples\SingleServer.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part
from dream.simulation.Globals import runSimulation

#define the objects of the model
S=Source('S1','Source',interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
M=Machine('M1','Machine',
processingTime={'distributionType':'Fixed','mean':0.25})
E=Exit('E1','Exit')

#define predecessors and successors for the objects
S.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[S],successorList=[E])
E.defineRouting(predecessorList=[M])

def main():
    # add all the objects in a list
    objectList=[S,M,E]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio = (M.totalWorkingTime/maxSimTime)*100
    print "the total working ratio of the Machine is", working_ratio, "%"
    return {"parts": E.numOfExits,
```

```
            "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 2880 parts
the working ratio of the Machine is 50.0 %
```

Some notes on the code:

- In the first line we import all the ManPy objects that will be used in the model. The user could also use from *dream.simulation.imports import *,* but it is generally lighter to import only what is needed
- In the second line we import *Globals.runSimulation* method. This is a supplementary method that runs the simulation. Two needed arguments are a list with all the simulation objects and the length of the experiment. As we see this is called as runSimulation(objectList, maxSimTime). There are other optional arguments as we will see in other examples.
- In the examples of this tutorial we always define a function *main()* which is the "main" program to be run. In the end we include the line if __name__ == '__main__': main() so that the program executes
- In every example there is a *return* statement. This is done for reasons of testing and it should not bother the reader. We keep the results under unit testing (check here -> https://github.com/nexedi/dream/blob/master/dream/tests/testSimulationExamples.py), so that we can ascertain changes in the code do not affect the execution of the examples.
- ManPy needs an abstract time unit. The user defines what this is. In this model we picked minutes. The length of the simulation is set to 1440.0 (so it is minutes it corresponds to 24 hours). We give this as a float to use it later in calculations.
- ManPy time units are decimals. So 30 and 15 seconds have been translated to 0.5 and 0.25 minutes respectively.
- *defineRouting* in most CoreObjects gets two lists as arguments (perdecessorList, successorList) with this sequence. In special cases like the Source and the Exit only one list is required. In this example the name of the argument is specified when the method is called, but if the user gives the inputs with the same sequence (see next examples) the result shall be the same.
- We see that distributions such as interarrival and processing times are defined for the objects as Python dictionaries. This way there is more flexibility in the attributes a distribution needs.
- Attributes like *totalWorkingTime* or *numOfExits* are part of the ManPy API. A full description of this is going to be developed.

We see the results are logical:

- In 1440 minutes and a part coming every 0.5 minutes and staying in the system 0.25 minutes (no blocking) it is normal to produce 2880 parts
- Since parts come every 30 seconds and the machine processes them for 15 seconds it is logical

(Note: all the programs presented here, and generally ManPy objects are verified against a commercial simulation package. We use Plant Simulation - http://www.plm.automation.siemens.com/en_us/products/tecnomatix/plant_design/plant_simulation.shtml)

4.2 Two servers model with failures and repairman

The second model is a bit more complex. The graphical representation is available in Figure 3

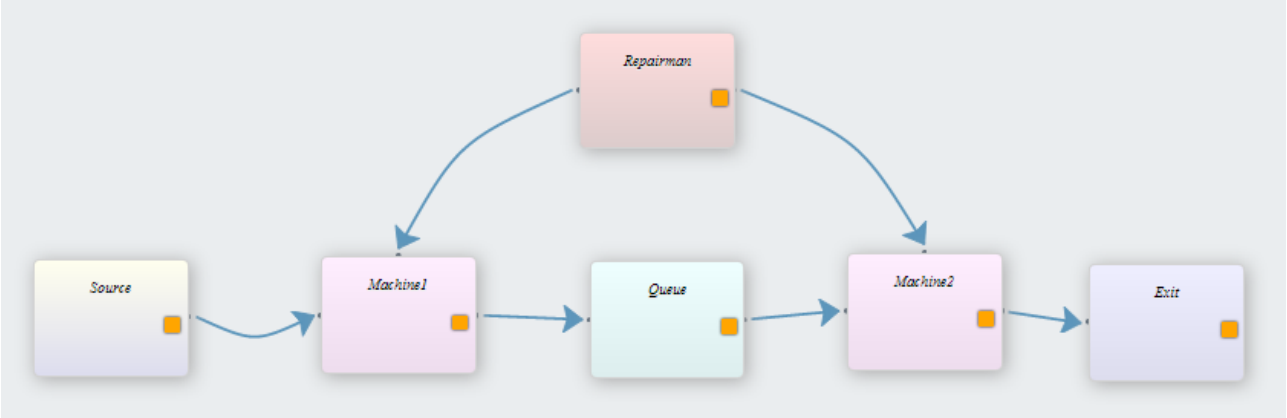

Figure 3: Two servers model with failures and repairman

In this model we have two Machines and a Queue between them. The Machines are vulnerable to failures and when a failure happens then they need a repairman to get fixed. In our model there is only one repairman named Bob available. We have the following data:

- The source produces parts. One part is produced every 30 seconds
- For Machine1
  - Processing time is Fixed to 15 seconds
  - MTTF is 1 hour
  - MTTR is 5 minutes
- For Machine2
  - Processing time is Fixed to 90 seconds
  - MTTF is 40 minutes
  - MTTR is 10 minutes
- The capacity of the Queue is 1
- We want to study the system in a 24 hours period and identify the number of items that are produced, the blockage ratio in Machine1 and the working ration of the repairman.

Below is the ManPy main script to run this model (dream\simulation\Examples\TwoServers.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part, Repairman,
Queue, Failure
from dream.simulation.Globals import runSimulation

#define the objects of the model
R=Repairman('R1', 'Bob')
S=Source('S1','Source',
interarrivalTime={'distributionType':'Fixed','mean':0.5}, entity='Dream.Part')
M1=Machine('M1','Machine1',
processingTime={'distributionType':'Fixed','mean':0.25})
Q=Queue('Q1','Queue')
M2=Machine('M2','Machine2',
processingTime={'distributionType':'Fixed','mean':1.5})
E=Exit('E1','Exit')
#create failures
F1=Failure(victim=M1,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5}, repairman=R)
```

```
F2=Failure(victim=M2,
distribution={'distributionType':'Fixed','MTTF':40,'MTTR':10}, repairman=R)

#define predecessors and successors for the objects
S.defineRouting([M1])
M1.defineRouting([S],[Q])
Q.defineRouting([M1],[M2])
M2.defineRouting([Q],[E])
E.defineRouting([M2])

def main():

    # add all the objects in a list
    objectList=[S,M1,M2,E,Q,R,F1,F2]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    blockage_ratio = (M1.totalBlockageTime/maxSimTime)*100
    working_ratio = (R.totalWorkingTime/maxSimTime)*100
    print "the blockage ratio of", M1.objName,  "is", blockage_ratio, "%"
    print "the working ratio of", R.objName,"is", working_ratio, "%"
    return {"parts": E.numOfExits,
          "blockage_ratio": blockage_ratio,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

```
the system produced 732 parts
the blockage ratio of Machine1 is 78.1770833333 %
the working ratio of Bob is 26.7361111111 %
```

Note that it is handy to declare the other objects first so that the failures take them as arguments (in this example the Machine and Repairman types)

4.3 An assembly line

In this example we use another ManPy object. Assembly takes two types of Entities, parts and frames. A frame can be loaded with a number of parts. The logic is that the Assembly waits first for a frame and when it has one then it loads the parts to it when they arrive. Figure 4 gives a graphical representation of the system.

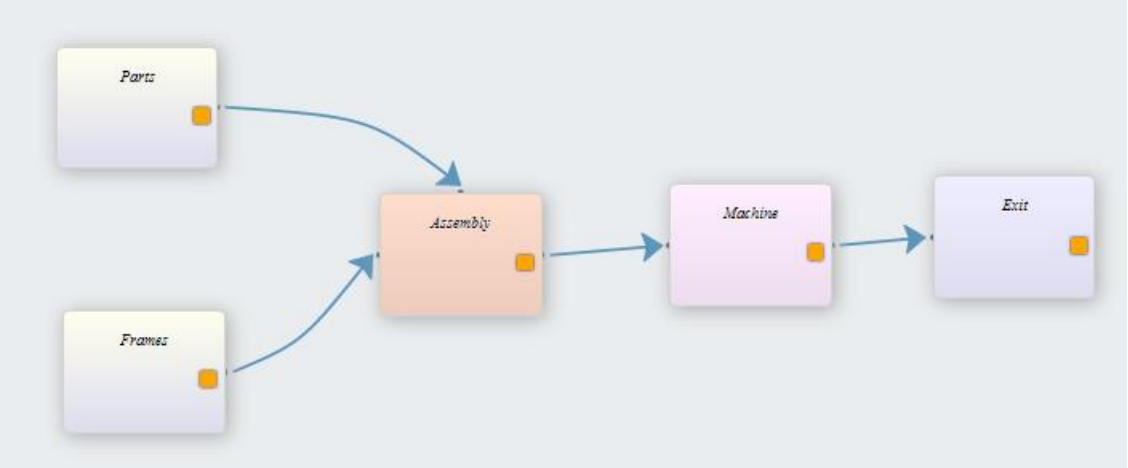

Figure 4: An assembly line

- "Parts" produces parts. One part is produced every 30 seconds
- "Frames" produces parts. One frame is produced every 2 minutes
- A Frame has a fixed capacity of 4 parts
- The Assembly has a fixed processing time of 2 minutes
- For Machine
    - Processing time is Fixed to 15 seconds
    - MTTF is 1 hour
    - MTTR is 5 minutes
- We want to study the system in 24 hours and identify the number of items that are produced and the blockage ratio in Assembly.

Below is the ManPy main script to run this model (dream\simulation\Examples\AssemblyLine.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part, Frame,
Assembly, Failure
from dream.simulation.Globals import runSimulation

#define the objects of the model
Frame.capacity=4
Sp=Source('SP','Parts',
interarrivalTime={'distributionType':'Fixed','mean':0.5}, entity='Dream.Part')
Sf=Source('SF','Frames', interarrivalTime={'distributionType':'Fixed','mean':2},
entity='Dream.Frame')
M=Machine('M','Machine',
processingTime={'distributionType':'Fixed','mean':0.25})
A=Assembly('A','Assembly', processingTime={'distributionType':'Fixed','mean':2})
E=Exit('E1','Exit')

F=Failure(victim=M,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5})

#define predecessors and successors for the objects
Sp.defineRouting([A])
Sf.defineRouting([A])
A.defineRouting([Sp,Sf],[M])
M.defineRouting([A],[E])
E.defineRouting([M])

def main():
```

```
    # add all the objects in a list
    objectList=[Sp,Sf,M,A,E,F]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "frames"
    working_ratio=(A.totalWorkingTime/maxSimTime)*100
    print "the working ratio of", A.objName,  "is", working_ratio, "%"
    return {"frames": E.numOfExits,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 664 frames
the working ratio of Assembly is 92.3611111111 %
```

Note that the capacity of the frames is set as an attribute of the class with *Frame.capacity=4*

4.4 Setting WIP

In the previous examples we had a Source object that created Entities in the system following a distribution for inter-arrival times. This is classical DES modelling, but it is also common that Entities are needed to be set as work in progress (WIP) in a model. In this section we will see 3 simple examples of this.

4.4.1 WIP in a buffer

The system is the one of Figure 5. It looks like the SingleServer example, but instead of a Source now we have a Queue preceding the Machine. The Machine is identical to the one og the SingleServer example. Let's say that we have one Part in this Queue as WIP.

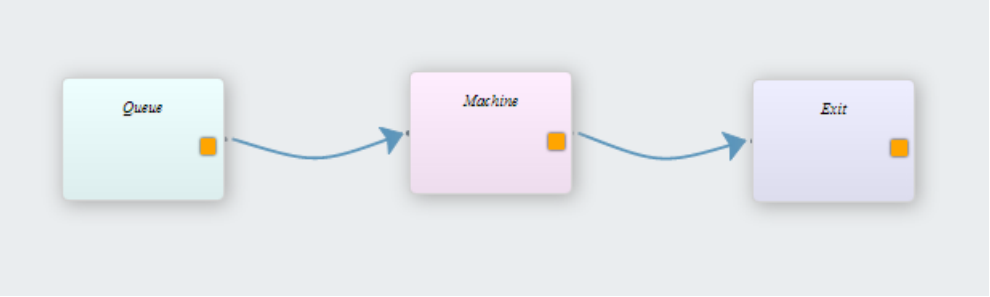

Figure 5: A server with a queue

Following is the code to model this system (dream\simulation\Examples\SettingWip1.py)

```
from dream.simulation.imports import Machine, Queue, Exit, Part, ExcelHandler
from dream.simulation.Globals import runSimulation, G

#define the objects of the model
```

```
Q=Queue('Q1','Queue', capacity=1)
M=Machine('M1','Machine',
processingTime={'distributionType':'Fixed','mean':0.25})
E=Exit('E1','Exit')
P1=Part('P1', 'Part1', currentStation=Q)

#define predecessors and successors for the objects
Q.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[Q],successorList=[E])
E.defineRouting(predecessorList=[M])

def main():
    # add all the objects in a list
    objectList=[Q,M,E,P1]
    # set the length of the experiment
    maxSimTime=float('inf')
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime, trace='Yes')

    #print the results
    print "the system produced", E.numOfExits, "parts in", E.timeLastEntityLeft,
"minutes"
    working_ratio = (M.totalWorkingTime/G.maxSimTime)*100
    print "the total working ratio of the Machine is", working_ratio, "%"
    ExcelHandler.outputTrace('Wip1')
    return {"parts": E.numOfExits,
            "simulationTime":E.timeLastEntityLeft,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 1 parts in 0.25 minutes
the total working ratio of the Machine is 100.0 %
```

Some notes on the code:

- We created the Part as a part type and we set its *currentStation* attribute to Q which was already defined.
- In this example we let the model run for infinite time (*maxSimTime=float('inf')*). The simulation will stop when there are no more events, which in this case happens when the Part finishes in the Exit. Notice that if a user applies infinite time in a simulation that does not stop to produce events (like in the previous examples), then the execution of the model will never stop.
- In previous examples we used the *maxSimTime* we defined to calculate percentages. Now that this was given as infinite this would not do. So we used the global attribute *G.maxSimTime*, which is calculated by the simulation itself. To be able to do this, we imported G in the beginning (*from dream.simulation.Globals import G*).
- *timeLastEntityLeft* is an attribute of ManPy CoreObjects that holds the simulation time that the object last dsposed an Entity. In this example it *E.timeLastEntityLeft* coincides to *G.maxSimTime*
- One asset that ManPy objects offer in order to enhance debugging it the feature of outputting trace to Excel. All ManPy objects output to the same Excel file and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to

be verified, it should be turned off since it slows the program significantly. In order to enable trace in this example:

- We imported ExcelHandler in the beginning. This is a ManPy script that holds Excel related methods.
- We invoked *runSimulation* giving *trace= 'Yes'* (default value is *'No'*)
- We saved the trace in the end with (*ExcelHandler.outputTrace('Wip1')*), "Wip1" is the name of the *.xls* file that will be saved.

The trace of this example is:

0	Part1	released Queue
0	Part1	got into Machine
0.25	Part1	ended processing in Machine
0.25	Part1	released Machine
0.25	Part1	got into Exit

Note that every object has its own *outputTrace* method which a user can customize. Of course this can also be omitted if it is not desirable for the object to output trace at all.

4.4.2 WIP in a server

Sometimes we need to define that the WIP is not in a buffer but in the middle of processing in a Machine. In the previous example we will add a new Part that is in the Machine (dream\simulation\Examples\SettingWip2.py):

```
P2=Part('P2', 'Part2', currentStation=M)
```

Running the model we get the following in our console:

```
the system produced 2 parts in 0.5 minutes
the total working ratio of the Machine is 100.0 %
```

The trace is:

0.25	Part2	ended processing in Machine
0.25	Part2	released Machine
0.25	Part2	got into Exit
0.25	Part1	released Queue
0.25	Part1	got into Machine
0.5	Part1	ended processing in Machine
0.5	Part1	released Machine
0.5	Part1	got into Exit

In the previous situation Part2 needed all the processing time of the Machine, so it finished in 0.25. But sometimes the WIP is in the middle of processing. To define what processing time is remaining in the current station we set the *remainingProcessingTime* attribute (dream\simulation\Examples\SettingWip2.py):

```
P2=Part('P2', 'Part2', currentStation=M,
remainingProcessingTime={'distributionType':'Fixed','mean':0.1})
```

The attribute is defined also as a dictionary since it may be stochastic.
Running the model we get the following in our console:

```
the system produced 2 parts in 0.35 minutes
the total working ratio of the Machine is 100.0 %
```

The trace certifies that P2 was processed for 0.1 minutes (assuming that it had already got processed for 0.15).

0.1	Part2	ended processing in Machine
0.1	Part2	released Machine
0.1	Part2	got into Exit
0.1	Part1	released Queue
0.1	Part1	got into Machine
0.35	Part1	ended processing in Machine
0.35	Part1	released Machine
0.35	Part1	got into Exit

4.5 Parallel stations

In this section we will see some examples of object customization.

4.5.1 Default behaviour

Our model consists of a source, a buffer and two Milling machines that work in parallel. A graphical representation is given in Figure 6. We have the following data:

- The source produces parts. One part is produced every 30 seconds
- For Machine1
    - Processing time is Fixed to 15 seconds
    - MTTF is 1 hour
    - MTTR is 5 minutes
- For Machine2
    - Processing time is Fixed to 15 seconds
    - No failures
- The capacity of the Queue is infinite
- We want to study the system in 24 hours and identify the number of items that are produced, the working ratio of both Machines

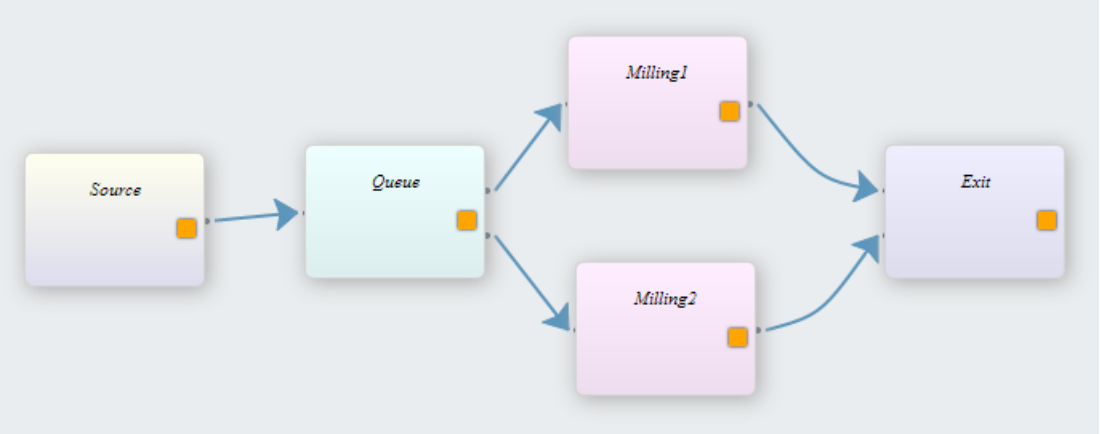

Figure 6: Parallel stations and Queue customization

To model this scenario we need nothing more than we already described. The code is given below (dream\simulation\Examples\ParallelServers1.py)

```
from dream.simulation.imports import Machine, Source, Exit, Part, Queue, Failure
from dream.simulation.Globals import runSimulation

#define the objects of the model
S=Source('S','Source', interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
Q=Queue('Q','Queue', capacity=float("inf"))
M1=Machine('M1','Milling1',
processingTime={'distributionType':'Fixed','mean':0.25})
M2=Machine('M2','Milling2',
processingTime={'distributionType':'Fixed','mean':0.25})
E=Exit('E1','Exit')
F=Failure(victim=M1,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5})

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

def main():

    # add all the objects in a list
    objectList=[S,Q,M1,M2,E,F]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio_M1=(M1.totalWorkingTime/maxSimTime)*100
    working_ratio_M2=(M2.totalWorkingTime/maxSimTime)*100
    print "the working ratio of", M1.objName,  "is", working_ratio_M1, "%"
    print "the working ratio of", M2.objName,  "is", working_ratio_M2, "%"
    return {"parts": E.numOfExits,
          "working_ratio_M1": working_ratio_M1,
          "working_ratio_M2": working_ratio_M2}
```

```
if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 2880 parts
the working ratio of Milling1 is 23.0902777778 %
the working ratio of Milling2 is 26.9097222222 %
```

We see that Milling2 is slightly busier than Milling1. This is logical since Milling1 gets also failures.

4.5.2 Queue customization

Let's assume now, that in our real system, Milling1 has a greater priority than Milling2, i.e. a part will go to Milling1, unless it is not available so it will go to Milling2.

The default behaviour of Queue is to handle things in a cyclic way (if both successors available select first Milling1 then Milling2 etc). To change this we have to override Queue's *selectReceiver* method.

The code is given below (dream\simulation\Examples\ParallelServers2.py).

```
from dream.simulation.imports import Machine, Source, Exit, Part, Queue, Failure
from dream.simulation.Globals import runSimulation

#the custom queue
class SelectiveQueue(Queue):
    # override so that it first chooses M1 and then M2
    def selectReceiver(self,possibleReceivers=[]):
        if M1.canAccept():
            return M1
        elif M2.canAccept():
            return M2
        return None

#define the objects of the model
S=Source('S','Source', interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
Q=SelectiveQueue('Q','Queue', capacity=float("inf"))
M1=Machine('M1','Milling1',
processingTime={'distributionType':'Fixed','mean':0.25})
M2=Machine('M2','Milling2',
processingTime={'distributionType':'Fixed','mean':0.25})
E=Exit('E1','Exit')
F=Failure(victim=M1,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5})

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

def main():

    # add all the objects in a list
    objectList=[S,Q,M1,M2,E,F]
```

```
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio_M1=(M1.totalWorkingTime/maxSimTime)*100
    working_ratio_M2=(M2.totalWorkingTime/maxSimTime)*100
    print "the working ratio of", M1.objName,  "is", working_ratio_M1, "%"
    print "the working ratio of", M2.objName,  "is", working_ratio_M2, "%"
    return {"parts": E.numOfExits,
          "working_ratio_M1": working_ratio_M1,
          "working_ratio_M2": working_ratio_M2}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 2880 parts
the working ratio of Milling1 is 46.1805555556 %
the working ratio of Milling2 is 3.81944444444 %
```

We see now that the working ration of Milling2 is drastically reduced that is natural since it takes parts only when Milling1 is busy or failed.

Some notes on the code:

- SelectiveQueue is a new custom object. It has its own version of *selectReceiver*, but in everything else it is identical to Queue.
- Q is now of type SelectiveQueue
- The implementation of SelectiveQueue is highly customized. It works only in this model with the given ids (‘M1’ and ‘M2’). A more generic approach is given in the next example.

4.5.3 More generic implementation of selective queue

Now we make a more generic implementation of SelectiveQueue, so that it sorts the receivers of the object according to an attribute we name priority. The new object is (dream\simulation\Examples\ParallelServers3.py):

```
#the custom queue
class SelectiveQueue(Queue):
    #override so that it chooses receiver according to priority
    def selectReceiver(self,possibleReceivers=[]):
        # sort the receivers according to their priority
        possibleReceivers.sort(key=lambda x: x.priority, reverse=True)
        if possibleReceivers[0].canAccept():
            return possibleReceivers[0]
        elif possibleReceivers[1].canAccept():
            return possibleReceivers[1]
        return None
```

In the main script we define also the priority of the Machines like below:

```
#create priority attribute in the Machines
M1.priority=10
M2.priority=0
```

As expected, the model gives the same result as the one of the previous example. However, this is a cleaner implementation since it does not involve specific instances. This SelectiveQueue could be used at any model that we need a Queue to prioritize its successors.

Generally, users are welcome to customize their objects at different levels:

- Objects for specific models like the SelectiveQueue shown in the first example
- More generic objects so that the user can re-use the in different models
- Even more generic objects so that the user can share them with other users. Here documentation would be essential.

4.5.4 Counting the parts each machine produced

In the previous example, we assume that in the Exit we want to count how many parts were processed by Milling1 and how many by Milling2. For this we need to make 3 modifications:

- Create two new global variables. Note that Globals.G is a class to store global variables for a model:
  - G.NumM1 as a counter that counts the parts that were processed by Milling1
  - G.NumM2 as a counter that counts the parts that were processed by Milling2
- Create a new Machine type named Milling. This will override the *getEntity* method so that it sets an attribute to the part that shows from which Milling it passed
- Create a new Exit type named CountingExit. This will override the *getEntity* method so that it reads the attribute of the part and increments the global counters accordingly

The code is given below (dream\simulation\Examples\ParallelServers4.py).

```
from dream.simulation.imports import Machine, Source, Exit, Part, Queue,
Globals, Failure, G
from dream.simulation.Globals import runSimulation

#the custom queue
class SelectiveQueue(Queue):
    #override so that it chooses receiver according to priority
    def selectReceiver(self,possibleReceivers=[]):
        # sort the receivers according to their priority
        possibleReceivers.sort(key=lambda x: x.priority, reverse=True)
        if possibleReceivers[0].canAccept():
            return possibleReceivers[0]
        elif possibleReceivers[1].canAccept():
            return possibleReceivers[1]
        return None

#the custom machine
class Milling(Machine):
    def getEntity(self):
        activeEntity=Machine.getEntity(self)        #call the parent method to
get the entity
        part=self.getActiveObjectQueue()[0]         #retrieve the obtained part
        part.machineId=self.id                      #create an attribute to the
obtained part and give it the value of the object's id
        return activeEntity                         #return the entity obtained

#the custom exit
```

```
class CountingExit(Exit):
    def getEntity(self):
        activeEntity=Exit.getEntity(self)                          #call the
parent method to get the entity
        #check the attribute and update the counters accordingly
        if activeEntity.machineId=='M1':
            G.NumM1+=1
        elif activeEntity.machineId=='M2':
            G.NumM2+=1
        return activeEntity             #return the entity obtained

#define the objects of the model
S=Source('S','Source', interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
Q=SelectiveQueue('Q','Queue', capacity=float("inf"))
M1=Milling('M1','Milling1',
processingTime={'distributionType':'Fixed','mean':0.25})
M2=Milling('M2','Milling2',
processingTime={'distributionType':'Fixed','mean':0.25})
E=CountingExit('E1','Exit')
F=Failure(victim=M1,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5})

#create the global counter variables
G.NumM1=0
G.NumM2=0

#create priority attribute in the Machines
M1.priority=10
M2.priority=0

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

def main():

    # add all the objects in a list
    objectList=[S,Q,M1,M2,E,F]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio_M1=(M1.totalWorkingTime/maxSimTime)*100
    working_ratio_M2=(M2.totalWorkingTime/maxSimTime)*100
    print "the working ratio of", M1.objName,  "is", working_ratio_M1, "%"
    print "the working ratio of", M2.objName,  "is", working_ratio_M2, "%"
    print M1.objName, "produced", G.NumM1, "parts"
    print M2.objName, "produced", G.NumM2, "parts"
    return {"parts": E.numOfExits,
          "working_ratio_M1": working_ratio_M1,
          "working_ratio_M2": working_ratio_M2,
          "NumM1":G.NumM1,
          "NumM2":G.NumM2}
```

```
if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 2880 parts
the working ratio of Milling1 is 46.1805555556 %
the working ratio of Milling2 is 3.81944444444 %
Milling1 produced 2660 parts
Milling2 produced 220 parts
```

4.6 Stochastic model

All the models so far have been deterministic. Real systems tend to be random with different reasons of stochasticity. In stochastic models we have to run many replications with different random seeds and give the results in confidence intervals.

We take our second example (dream\simulation\Examples\TwoServers.py) and we extend it into a stochastic situation. The model is the same, the only change is that the machines have stochastic processing times. More specifically:

- Machine1 processing time follows the normal distribution with mean=0.25, stdev=0.1, min=0.1, max=1 (all in minutes)
- Machine2 processing time follows the normal distribution with mean=1.5, stdev=0.3, min=0.5, max=5 (all in minutes)

The failures and the interarrival times remain deterministic as before.

Below is the ManPy main script to run this model (dream\simulation\Examples\TwoServersStochastic.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part, Repairman,
Queue, Failure
from dream.simulation.Globals import runSimulation

#define the objects of the model
R=Repairman('R1', 'Bob')
S=Source('S1','Source', interarrivalTime={'distributionType':'Exp','mean':0.5},
entity='Dream.Part')
M1=Machine('M1','Machine1',
processingTime={'distributionType':'Normal','mean':0.25,'stdev':0.1,'min':0.1,'m
ax':1})
M2=Machine('M2','Machine2',
processingTime={'distributionType':'Normal','mean':1.5,'stdev':0.3,'min':0.5,'ma
x':5})
Q=Queue('Q1','Queue')
E=Exit('E1','Exit')
#create failures
F1=Failure(victim=M1,
distribution={'distributionType':'Fixed','MTTF':60,'MTTR':5}, repairman=R)
F2=Failure(victim=M2,
distribution={'distributionType':'Fixed','MTTF':40,'MTTR':10}, repairman=R)

#define predecessors and successors for the objects
S.defineRouting([M1])
M1.defineRouting([S],[Q])
Q.defineRouting([M1],[M2])
M2.defineRouting([Q],[E])
```

```
E.defineRouting([M2])

def main():

    # add all the objects in a list
    objectList=[S,M1,M2,E,Q,R,F1,F2]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime, numberOfReplications=10, seed=1)

    print 'The exit of each replication is:'
    print E.Exits

    # calculate confidence interval using the Knowledge Extraction tool
    from dream.KnowledgeExtraction.ConfidenceIntervals import Intervals
    from dream.KnowledgeExtraction.StatisticalMeasures import
BasicStatisticalMeasures
    BSM=BasicStatisticalMeasures()
    lb, ub = Intervals().ConfidIntervals(E.Exits, 0.95)
    print 'the 95% confidence interval for the throughput is:'
    print 'lower bound:', lb
    print 'mean:', BSM.mean(E.Exits)
    print 'upper bound:', ub

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
The exit of each replication is:
[729, 728, 732, 739, 729, 732, 727, 724, 721, 728]
the 95% confidence interval for the throughput is:
lower bound: 725.420720244
mean: 728.9
upper bound: 732.379279756
```

Some notes:

- Here *runSimulation* got two additional parameters:
  - *numberOfReplications*: this is how many times we want the simulation to be run. Default value is 1, but in stochastic cases we need many replications to be able to statistically evaluate the results.
  - *seed*: this is the seed of random number generation. This will be used in the first replication and then the seed will be incremented by 1 for each replication
- In order to calculate confidence intervals ManPy uses again DREAM Knowledge extraction tool.
- In the normal distribution it is on the developer's responsibility not to give irrational values. For example, if a processing time is negative ManPy will crash. Another example, if *min* is larger than *max* in normal distribution, ManPy would also raise an error.

4.7 Job-Shop Examples

4.7.1 A simple Job-Shop

So far all the CoreObjects had dedicated predecessors and successors. There are situations where it is desirable to model a job shop system where CoreObjects can give/receive to/from whichever other CoreObject in the model. The information of which CoreObject is the next station is an attribute of the Entity. As an example we give the model of **Figure 7**. In this model there are 3 Queues, 3 Machines and an Exit. Every entity will have to start from a CoreObject and have its route and processing times assigned to its attributes.

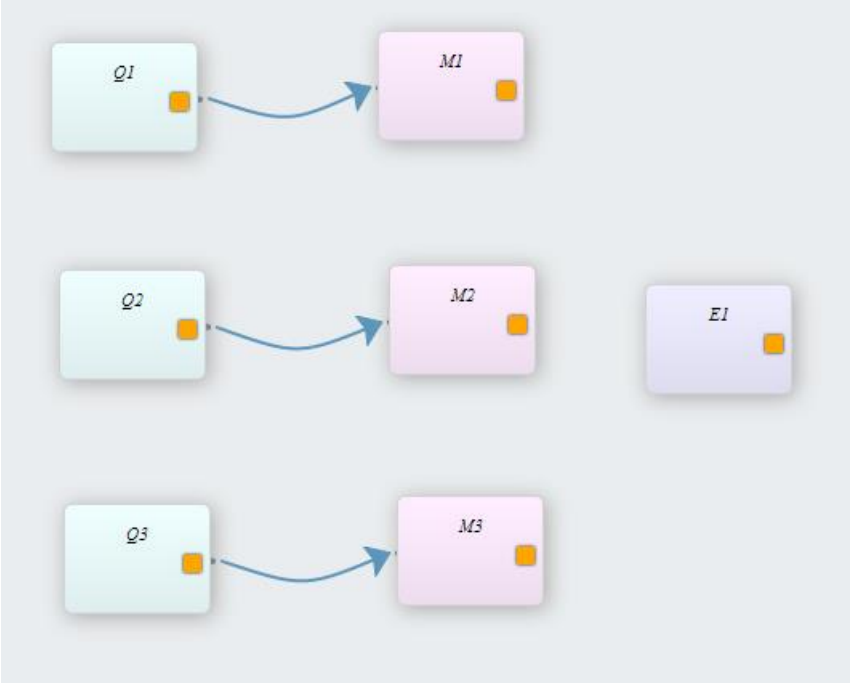

Figure 7: a job shop model

To model such situations ManPy object repository has the following objects:

- **MachineJobShop**: inherits from Machine and overrides the logic of methods such as *updateNext* in order to be able to give to every CoreObject in the model. The next CoreObject is read by the Entity's attributes and the method *updateNext* is invoked within *getEntity.* Also, the methods *canAccept and canAcceptAndIsRequested* are overridden in order to render the MachineJobShop objects able to receive from every CoreObject in the model. For this purpose, an extra method *isInRoute* is created in order check if the object receiving an entity is in the *route* of the entity to be disposed. The next CoreObject is read by the Entity's attributes and this is done in *getEntity.* Finally, it overrides *calculateProcessingTime* in order to calculate the processing time according to the Entity's attributes.
- **QueueJobShop**: inherits from Queue and overrides the logic of methods such as or *updateNext* in order to be able to give to every CoreObject in the model. Also, it overrides *canAccept and canAcceptAndIsRequested* in order to be able to receive from every CoreObject in the model. Again, *isInRoute* is used in order check if the object receiving an entity is in the *route* of the entity to be disposed.
- **ExitJobShop**: inherits from Exit but overrides the logic of methods such as *isInRoute* in order to be able to receive from every CoreObject.
- **Job**: inherits from Entity. One of its attributes is a list named *route*. This list has the following form [[id1,processingTime1], [id2,processingTime2], …, [idN,processingTimeN]]. Every item in *route* corresponds to the id of a CoreObject and the processing time in this CoreObject.

Another attribute called *remainingRoute* is also a list that holds the future stops of a Job at any moment of simulation time. In the beginning of the simulation these lists are equal. Current implementation of Job can be used only for Fixed processing times. Job has also a list named *schedule*, which is updated by the CoreObject every time it receives the Job. This holds the output for the Job, i.e. which stations it entered and when.

In our first simple example we assume that we have only one Job in the model shown in Figure 7. Our data for this Job is:

- It starts in Queue1 and it has to visit Machine1, Machine3 and Machine2 (in this sequence) before it exits the system
- Its processing time in M1 is 1
- Its processing time in M3 is 3
- Its processing time in M2 is 2

Below is the ManPy main script to run this model (dream\simulation\Examples\JobShop1.py):

```
from dream.simulation.imports import MachineJobShop, QueueJobShop, ExitJobShop,
Job
from dream.simulation.Globals import runSimulation

#define the objects of the model
Q1=QueueJobShop('Q1','Queue1', capacity=float("inf"))
Q2=QueueJobShop('Q2','Queue2', capacity=float("inf"))
Q3=QueueJobShop('Q3','Queue3', capacity=float("inf"))
M1=MachineJobShop('M1','Machine1')
M2=MachineJobShop('M2','Machine2')
M3=MachineJobShop('M3','Machine3')
E=ExitJobShop('E','Exit')

#define the route of the Job in the system
route=[{"stationIdsList": ["Q1"]},
          {"stationIdsList": ["M1"],"processingTime":{"distributionType":
"Fixed","mean": "1"}},
          {"stationIdsList": ["Q3"]},
          {"stationIdsList": ["M3"],"processingTime":{"distributionType":
"Fixed","mean": "3"}},
          {"stationIdsList": ["Q2"]},
          {"stationIdsList": ["M2"],"processingTime":{"distributionType":
"Fixed","mean": "2"}},
          {"stationIdsList": ["E"],}]
#define the Jobs
J=Job('J1','Job1',route=route)

def main():
    # add all the objects in a list
    objectList=[M1,M2,M3,Q1,Q2,Q3,E,J]
    # set the length of the experiment
    maxSimTime=float('inf')
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #loop in the schedule to print the results
    returnSchedule=[]     # dummy variable used just for returning values and
testing
    for record in J.schedule:
        returnSchedule.append([record[0].objName,record[1]])
        print J.name, "got into", record[0].objName, "at", record[1]
    return returnSchedule

if __name__ == '__main__':
```

```
main()
```

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 0
Job1 got into Queue3 at 1.0
Job1 got into Machine3 at 1.0
Job1 got into Queue2 at 4.0
Job1 got into Machine2 at 4.0
Job1 got into Exit at 6.0
```

Having only one Job it is very easy to confirm that we got the correct result.

Some notes on the code:

- We see that the *route* of the Job is given as a list of dictionaries. In every step the user has to give a list with the ids of the possible CoreObjects that the step can happen and also the data for the processing time if this is needed.

4.7.2 A Job-Shop with scheduling rules

For the model described in the previous example and in Figure 7 we assume now that we have 3 Jobs. For these Jobs we know:

- Job1:
    - It starts in Queue1 and it has to visit Machine1, Queue3, Machine3, Queue2 and Machine2 (in this sequence) before it exits the system
    - Its processing time in M1 is 1
    - Its processing time in M3 is 3
    - Its processing time in M2 is 2
    - Its priority is 1
    - Its due date is 100
- Job2:
    - It starts in Queue1 and it has to visit Machine1, Queue2, Machine2, Queue3 and Machine3 (in this sequence) before it exits the system
    - Its processing time in M1 is 2
    - Its processing time in M2 is 4
    - Its processing time in M3 is 6
    - Its priority is 1
    - Its due date is 90
- Job3
    - It starts in Queue1 and it has to visit Machine1, Queue1 and Machine3 (in this sequence) before it exits the system
    - Its processing time in M1 is 10
    - Its processing time in M3 is 3
    - Its priority is 0
    - Its due date is 110

We see above two new attributes of the Job class. These are in reality optional arguments of the parent class (Entity).

- *priority* is an integer. The higher the value the higher the priority assigned to the Entity.
- *dueDate* is a float. It shows the time that the Entity should be out of the system (in case the Entity represents an order or something similar). If our simulation units are minutes and the due date is in exactly one week after the start of the simulation run, then *dueDate*=10080 (60*24*7).

As we see, all the Jobs start from Queue1. The default scheduling rule of a Queue object is FIFO, i.e. the Entity to arrive first in the Queue will be the first to be given in another CoreObject. Nevertheless, there are several more scheduling rules supported.

- **Priority**: the Entities are sorted in order of ascending predefined priority (the lowest priority is to leave the Queue first)
- **EDD**: the Entities are sorted in order of ascending predefined due date (Earliest Due Date)
- **EOD**: the Entities are sorted in order of ascending predefined order date (Earliest Order Date)
- **NumStages**: the Entities are sorted in order of descending number of stages that they have to pass.
- **RPC**: the Entities are sorted in order of descending total processing time of stages that they have to pass (Remaining Processing Time).
- **SPT**: the Entities are sorted in order of ascending processing time of the next Machine they have to pass (Shortest Processing Time).
- **LPT**: the Entities are sorted in order of descending processing time of the next Machine they have to pass (Logest Processing Time).
- **MS**: the Entities are sorted in order of ascending slack time. Slack time is defined as due date minus the remaining processing time
- **WINQ**: the Entities are sorted in order of ascending number of Entities in the next stage that the Entity has to pass through (Work In Next Queue).
- **MC**: This stands for Multiple Criteria and it is applied when we have many scheduling rules used. For example we may need to use Priority, but for the Entities that have equal *priorities* EDD will be applied.

(*Note*: An advanced user can add new scheduling rules by creating a CoreObject that inherits from Queue and overrides the *activeQSorter* method)

We start our model with the assumption that Priority is applied as scheduling rule in Queue1. The other 2 Queues will remain FIFO.

Below is the ManPy main script to run this model (dream\simulation\Examples\JobShop2Priority.py):

```
from dream.simulation.imports import MachineJobShop, QueueJobShop, ExitJobShop,
Job
from dream.simulation.Globals import runSimulation

#define the objects of the model
Q1=QueueJobShop('Q1','Queue1', capacity=float("inf"), schedulingRule="Priority")
Q2=QueueJobShop('Q2','Queue2', capacity=float("inf"))
Q3=QueueJobShop('Q3','Queue3', capacity=float("inf"))
M1=MachineJobShop('M1','Machine1')
M2=MachineJobShop('M2','Machine2')
M3=MachineJobShop('M3','Machine3')
E=ExitJobShop('E','Exit')

#define predecessors and successors for the objects
Q1.defineRouting(successorList=[M1])
Q2.defineRouting(successorList=[M2])
Q3.defineRouting(successorList=[M3])
M1.defineRouting(predecessorList=[Q1])
M2.defineRouting(predecessorList=[Q2])
```

```
M3.defineRouting(predecessorList=[Q3])

#define the routes of the Jobs in the system
J1Route=[{"stationIdsList": ["Q1"]},
         {"stationIdsList": ["M1"],"processingTime":{"distributionType":
"Fixed","mean": "1"}},
         {"stationIdsList": ["Q3"]},
         {"stationIdsList": ["M3"],"processingTime":{"distributionType":
"Fixed","mean": "3"}},
         {"stationIdsList": ["Q2"]},
         {"stationIdsList": ["M2"],"processingTime":{"distributionType":
"Fixed","mean": "2"}},
         {"stationIdsList": ["E"],}]
J2Route=[{"stationIdsList": ["Q1"]},
         {"stationIdsList": ["M1"],"processingTime":{"distributionType":
"Fixed","mean": "2"}},
         {"stationIdsList": ["Q2"]},
         {"stationIdsList": ["M2"],"processingTime":{"distributionType":
"Fixed","mean": "4"}},
         {"stationIdsList": ["Q3"]},
         {"stationIdsList": ["M3"],"processingTime":{"distributionType":
"Fixed","mean": "6"}},
         {"stationIdsList": ["E"],}]
J3Route=[{"stationIdsList": ["Q1"]},
         {"stationIdsList": ["M1"],"processingTime":{"distributionType":
"Fixed","mean": "10"}},
         {"stationIdsList": ["Q3"]},
         {"stationIdsList": ["M3"],"processingTime":{"distributionType":
"Fixed","mean": "3"}},
         {"stationIdsList": ["E"],}]

#define the Jobs
J1=Job('J1','Job1',route=J1Route, priority=1, dueDate=100)
J2=Job('J2','Job2',route=J2Route, priority=1, dueDate=90)
J3=Job('J3','Job3',route=J3Route, priority=0, dueDate=110)

def main():
    # add all the objects in a list
    objectList=[M1,M2,M3,Q1,Q2,Q3,E,J1,J2,J3]
    # set the length of the experiment
    maxSimTime=float('inf')
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #output the schedule of every job
    returnSchedule=[]     # dummy variable used just for returning values and
testing
    for job in [J1,J2,J3]:
        #loop in the schedule to print the results
        for record in job.schedule:
            #schedule holds ids of objects. The following loop will identify the
name of the CoreObject with the given id
            name=None
            returnSchedule.append([record[0].objName,record[1]])
            print job.name, "got into", record[0].objName, "at", record[1]
        print "-"*30
    return returnSchedule

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 10.0
Job1 got into Queue3 at 11.0
Job1 got into Machine3 at 13.0
Job1 got into Queue2 at 16.0
Job1 got into Machine2 at 17.0
Job1 got into Exit at 19.0
------------------------------
Job2 got into Queue1 at 0
Job2 got into Machine1 at 11.0
Job2 got into Queue2 at 13.0
Job2 got into Machine2 at 13.0
Job2 got into Queue3 at 17.0
Job2 got into Machine3 at 17.0
Job2 got into Exit at 23.0
------------------------------
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10.0
Job3 got into Machine3 at 10.0
Job3 got into Exit at 13.0
------------------------------
```

We see that Job3 having the highest (lowest value) *priority* was the first to go to Machine1. The other two Jobs had equal priorities, so FIFO was applied (observing the loop where the WIP is set one can see the Job1 was added to Queue1 before Job2).

To test how the model works if Queue1 follows the Earliest Due Date rule we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2EDD.py):

Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="EDD")

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 2.0
Job1 got into Queue3 at 3.0
Job1 got into Machine3 at 3.0
Job1 got into Queue2 at 6.0
Job1 got into Machine2 at 6.0
Job1 got into Exit at 8.0
------------------------------
Job2 got into Queue1 at 0
Job2 got into Machine1 at 0
Job2 got into Queue2 at 2.0
Job2 got into Machine2 at 2.0
Job2 got into Queue3 at 6.0
Job2 got into Machine3 at 6.0
Job2 got into Exit at 12.0
------------------------------
Job3 got into Queue1 at 0
Job3 got into Machine1 at 3.0
Job3 got into Queue3 at 13.0
Job3 got into Machine3 at 13.0
Job3 got into Exit at 16.0
------------------------------
```

We see that Job2 having the earliest *dueDate* was the first to go to Machine1. Then Job1 followed and Job3 was the last.

To test how the model works if Queue1 follows the Remaining Process Time rule we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2RPC.py):

Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="RPC")

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 12.0
Job1 got into Queue3 at 13.0
Job1 got into Machine3 at 13.0
Job1 got into Queue2 at 16.0
Job1 got into Machine2 at 16.0
Job1 got into Exit at 18.0
------------------------------
Job2 got into Queue1 at 0
Job2 got into Machine1 at 10.0
Job2 got into Queue2 at 12.0
Job2 got into Machine2 at 12.0
Job2 got into Queue3 at 16.0
Job2 got into Machine3 at 16.0
Job2 got into Exit at 22.0
------------------------------
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10.0
Job3 got into Machine3 at 10.0
Job3 got into Exit at 13.0
------------------------------
```

We see that Job3 having the greatest remaining processing time was the first to go to Machine1. Then Job2 followed and Job1 was the last.

Finally, we want to test how the model works if Queue1 follows a multi criteria rule. First Priority is applied, and if Jobs have equal priorities, then EDD is applied we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2MC.py):

Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="MC-Priority-EDD")

We see that to define a multi criteria rule, we use MC and then the scheduling rules according to their sequence. All the scheduling rules are separated with “-”.

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 12.0
Job1 got into Queue3 at 13.0
Job1 got into Machine3 at 13.0
Job1 got into Queue2 at 16.0
Job1 got into Machine2 at 16.0
Job1 got into Exit at 18.0
------------------------------
Job2 got into Queue1 at 0
Job2 got into Machine1 at 10.0
Job2 got into Queue2 at 12.0
Job2 got into Machine2 at 12.0
Job2 got into Queue3 at 16.0
Job2 got into Machine3 at 16.0
Job2 got into Exit at 22.0
------------------------------
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10.0
Job3 got into Machine3 at 10.0
Job3 got into Exit at 13.0
------------------------------
```

We see that having the highest (lowest value) *priority*, Job3 was the first to get into Machine1. Contrary to the first example of this subsection though (JobShop2Priority.py), now Job2 is the second Job to go to the Machine. This happens because it has an earlier due date (*dueDate*) than Job1.

4.8 Batches and SubBatches

4.8.1 Batch decomposition

There are cases in production lines where units are grouped in batches. The units belonging to the same batch carry the same identification parameters. For further processing in different stations the batches are segregated in sub-batches. Sub-batches or batches cannot be mixed during the processing throughout the line. In order to model this behaviour, a number of new objects are introduced. In this example, a source creating butches and an object breaking the batches into sub-batches are presented.

Figure 8 depicts the model discussed in this example. A source creates batches with a specified number of units which then enter an input buffer of a machine. The machine can only operate on sub-batches. Thus, just before the entry of the machine the batches have to be broken into a specified number of sub-batches depending on a predefined number of units per sub-batches. For this purpose, a batch decomposition object is placed between the buffer unit and the machine. The exit acts as a drain for the already processed sub-batches.

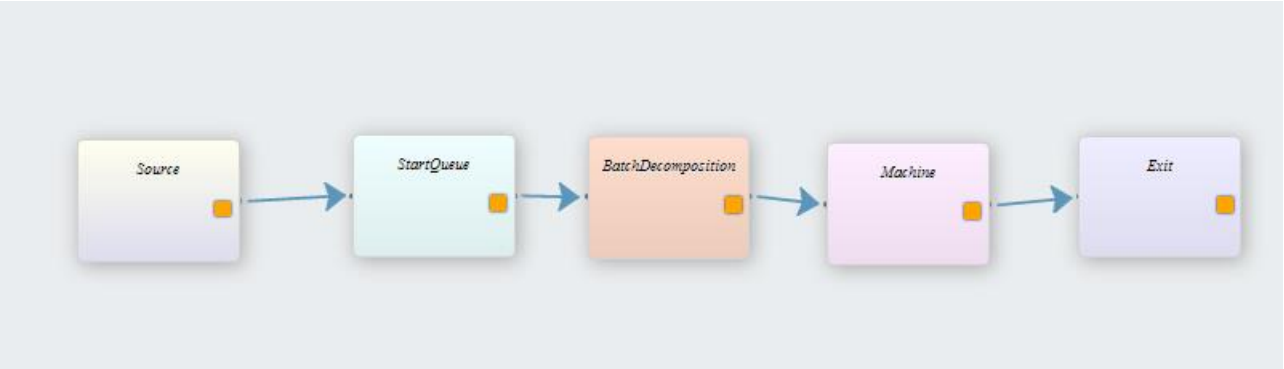

Figure 8: a simple batch decomposition example

ManPy object repository contains the following objects in order to model the described behaviour:

- **BatchSource**: inherits from Source and overrides the logic of the methods *__init__* and *createEntity* so as to create entities of type Batch with a specified number of units.
- **BatchDecomposition**: inherits from the CoreObject and introduces a new method *decompose* in order to provide the functionality of splitting a batch into sub-batches. It also overrides the logic of the methods *canAccept, haveToDispose*, *canAcceptAndIsRequested*, and *run* in order to prohibit the mixing up of the sub-batches (should not be able to able to accept a new Batch if there are already SubBatches in the object). *run* method should also be able to hold a track of the batches already decomposed which may later on reassembled.
- **Batch**: inherits from Entity but introduces the attributes *numberOfUnits* that holds, *numberOfSubBatches* that it is broken into, and *subBatchList* that holds the sub-batches that it is broken into.
- **SubBatch**: inherits from Entity also. It holds in one of its attributes an identifier parameter of the Batch it derived from.

In our first simple example we assume that we have only one Machine operating on SubBatches and its corresponding BatchDecomposition object. Our data for this example is:

- The BatchSource Source creates Batches with a certain *numberOfUnits*,
- The newly created Batches enter the buffer of the machine (StartQueue),
- The Batches are decomposed into SubBatches in the BatchDecomposition with a processing time of 1,

- the processing time of the Machine is 0.5

Below is the ManPy main script to run this model (dream\simulation\Examples\DecompositionOfBatches.py):

```
from dream.simulation.imports import Machine, BatchSource, Exit, Batch,
BatchDecomposition, Queue
from dream.simulation.Globals import runSimulation

# define the objects of the model
S=BatchSource('S','Source',interarrivalTime={'distributionType':'Fixed','mean':0
.5}, entity='Dream.Batch', batchNumberOfUnits=4)
Q=Queue('Q','StartQueue',capacity=100000)
BD=BatchDecomposition('BC', 'BatchDecomposition', numberOfSubBatches=4,
processingTime={'distributionType':'Fixed','mean':1})
M=Machine('M','Machine',processingTime={'distributionType':'Fixed','mean':0.5})
E=Exit('E','Exit')

# define the predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[BD])
BD.defineRouting([Q],[M])
M.defineRouting([BD],[E])
E.defineRouting([M])

def main():

    # add all the objects in a list
    objectList=[S,Q,BD,M,E]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    # print the results
    print "the system produced", E.numOfExits, "subbatches"
    working_ratio = (M.totalWorkingTime/maxSimTime)*100
    blockage_ratio = (M.totalBlockageTime/maxSimTime)*100
    waiting_ratio = (M.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M.objName, "is", working_ratio
    print "the blockage ratio of", M.objName, 'is', blockage_ratio
    print "the waiting ratio of", M.objName, 'is', waiting_ratio
    return {"subbatches": E.numOfExits,
            "working_ratio": working_ratio,
           "blockage_ratio": blockage_ratio,
           "waiting_ratio": waiting_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 2302 parts
the working ratio of Machine is 79.9652777778
the blockage ratio of Machine is 0.0
the waiting ratio of Machine is 20.0347222222
```

Some notes on the code:

- Batch and SubBatch are normal Entities with some additional attributes. No individual units for each Batch or SubBatch are taken into consideration for the modelling of the behaviour of these lines.
- BatchSource is in all aspects a normal Source creating Entities of type Batch.
- The newly introduced method *decompose* of the BatchDecomposition object is complementary to the method *reassemble* which will be presented later on.
- These operations (decomposing or processing on a station) are performed by operators. Such functionality will be later on introduced.

4.8.2 Serial Batch Processing

In this example we will introduce one more object developed in order to model the behaviour of a manufacturing line operating on Batches and SubBatches. As mentioned earlier, the *decompose* method of BatchDecomposition object should have a complementary method in order to output full Batches at the exit of the manufacturing line. The object implementing this functionality is named BatchReassembly.

The example presenting the use of this object is depicted in Figure 9. A source creates batches with a specified number of units which then enter an input buffer of a machine. The newly created Batches enter first a buffer Queue1 of Machine1. Machine1 can process only a smaller number of units, thus a group of units named sub-batch. A BatchDecomposition unit is placed before the Machine1. The SubBatch after being processed by Machine1 enters Queue2 which act as a buffer for Machine2. After being processed by Machine2 the SubBatches must be reassembled into Batches before being processed by Machine3 which operates only on Batches. For this purpose a BatchReassembly object is placed after Machine2 and before Machine3. BatchReassembly can only assemble SubBatches which are derived from the same Batch. Finally, the exit acts as a drain for the already processed sub-batches.

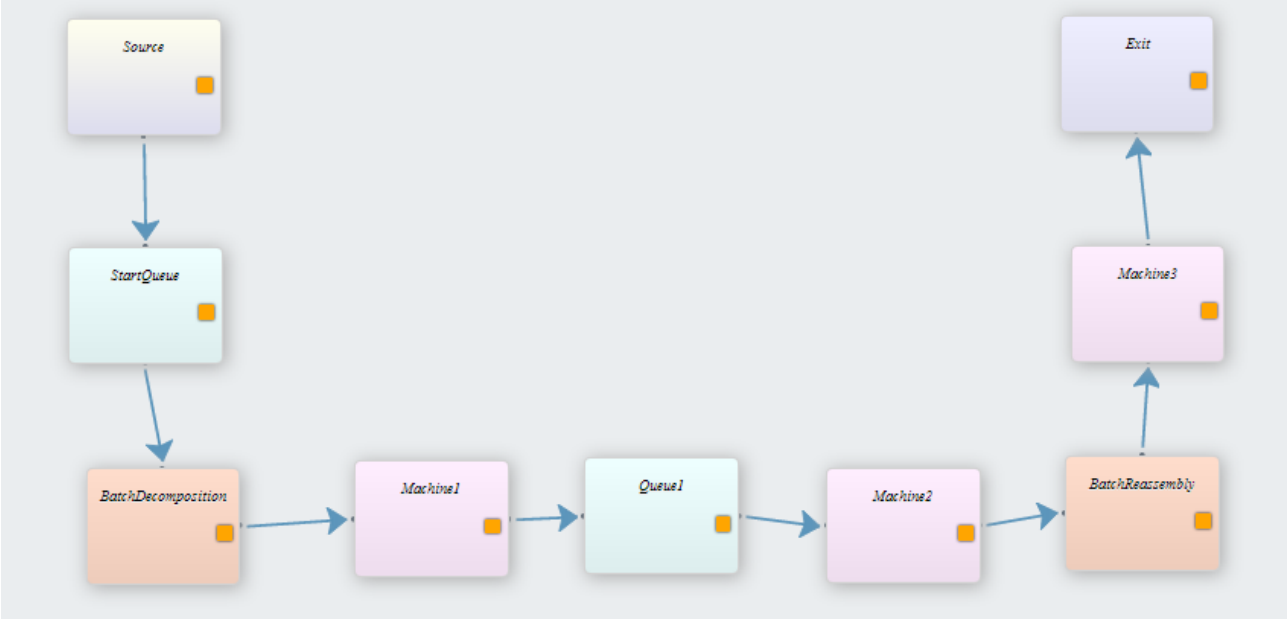

Figure 9: a simple batch decomposition and batch reassembly example

ManPy object repository contains the following object in order to model the described behaviour:

- **BatchReassembly**: inherits from the CoreObject and introduces a new method *reassemble* which reassembles a number of SubBatches derived from the same Batch. It also overrides the logic of the methods *canAccept, haveToDispose*, *canAcceptAndIsRequested*, and *run* in order to prohibit the mixing up of the sub-batches. The BatchReassembly should not be able to able to accept new SubBatches if they are not derived from the same Batch or if it holds an Entity of type Batch. The Batches reassembled should be removed from the list of Batches that wait to be reassembled. In addition it should be able to hand in an Entity to its successors only if the Entity is of type Batch.

In the current example, we consider 3 Machines of two different types, two Machines operating on SubBatches and one Machine operating on Batches. The Machine operating on Batches follows the processing done on Machine1 and Machine2. Therefore, a need for the use of a BatchReassembly object is introduced. Our data for this example is:

- BatchSource Source creates Batches with a certain *numberOfUnits*,
- The newly created Batches enter the buffer of the Machine1 (StartQueue),
- The Batch is then decomposed into SubBatches in the the BatchDecomposition. The processing time of the BatchDecomposition has a value of 1.
- The SubBatches are then processed by Machine1 and Machine2. Between a buffer Queue1 with capacity of 2 is placed. The processing times for these machines is 0.5 and 1 respectively.
- The Batches are reassembled into Batches in BatchReassembly. The reassembly is performed instantly while the processing time of Machine3, which lies just after BatchReassembly, is 1.

Below is the ManPy main script to run this model (dream\simulation\Examples\SerialBatchProcessing.py):

```
from dream.simulation.imports import Machine, BatchSource, Exit, Batch,
BatchDecomposition, BatchReassembly, Queue
from dream.simulation.Globals import runSimulation

# define the objects of the model
S=BatchSource('S','Source',interarrivalTime={'distributionType':'Fixed','mean':1
.5}, entity='Dream.Batch', batchNumberOfUnits=100)
Q=Queue('Q','StartQueue',capacity=100000)
BD=BatchDecomposition('BC', 'BatchDecomposition', numberOfSubBatches=4,
processingTime={'distributionType':'Fixed','mean':1})
M1=Machine('M1','Machine1',processingTime={'distributionType':'Fixed','mean':0.5
})
Q1=Queue('Q1','Queue1',capacity=2)
M2=Machine('M2','Machine2',processingTime={'distributionType':'Fixed','mean':1})
BRA=BatchReassembly('BRA', 'BatchReassembly', numberOfSubBatches=4,
processingTime={'distributionType':'Fixed','mean':0})
M3=Machine('M3','Machine3',processingTime={'distributionType':'Fixed','mean':1})
E=Exit('E','Exit')

# define the predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[BD])
BD.defineRouting([Q],[M1])
M1.defineRouting([BD],[Q1])
Q1.defineRouting([M1],[M2])
M2.defineRouting([Q1],[BRA])
BRA.defineRouting([M2],[M3])
M3.defineRouting([BRA],[E])
E.defineRouting([M3])

def main():
    # add all the objects in a list
    objectList=[S,Q,BD,M1,Q1,M2,BRA,M3,E]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    # print the results
    print "the system produced", E.numOfExits, "batches"
    working_ratio_M1 = (M1.totalWorkingTime/maxSimTime)*100
    blockage_ratio_M1 = (M1.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M1 = (M1.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M1.objName, "is", working_ratio_M1
    print "the blockage ratio of", M1.objName, 'is', blockage_ratio_M1
    print "the waiting ratio of", M1.objName, 'is', waiting_ratio_M1
    working_ratio_M2 = (M2.totalWorkingTime/maxSimTime)*100
```

```
    blockage_ratio_M2 = (M2.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M2 = (M2.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M2.objName, "is", working_ratio_M2
    print "the blockage ratio of", M2.objName, 'is', blockage_ratio_M2
    print "the waiting ratio of", M2.objName, 'is', waiting_ratio_M2
    working_ratio_M3 = (M3.totalWorkingTime/maxSimTime)*100
    blockage_ratio_M3 = (M3.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M3 = (M3.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M3.objName, "is", working_ratio_M3
    print "the blockage ratio of", M3.objName, 'is', blockage_ratio_M3
    print "the waiting ratio of", M3.objName, 'is', waiting_ratio_M3

    return {"batches": E.numOfExits,
           "working_ratio_M1": working_ratio_M1,
          "blockage_ratio_M1": blockage_ratio_M1,
          "waiting_ratio_M1": waiting_ratio_M1,
           "working_ratio_M2": working_ratio_M2,
          "blockage_ratio_M2": blockage_ratio_M2,
          "waiting_ratio_M2": waiting_ratio_M2,
           "working_ratio_M3": working_ratio_M3,
          "blockage_ratio_M3": blockage_ratio_M3,
          "waiting_ratio_M3": waiting_ratio_M3,
          }

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 359 parts
the working ratio of Machine1 is 50.0694444444
the blockage ratio of Machine1 is 49.8263888889
the waiting ratio of Machine1 is 0.104166666667
the working ratio of Machine2 is 99.8958333333
the blockage ratio of Machine2 is 0.0
the waiting ratio of Machine2 is 0.104166666667
the working ratio of Machine3 is 24.9305555556
the blockage ratio of Machine3 is 0.0
the waiting ratio of Machine3 is 75.0694444444
```

4.8.3 Clearing batch lines

In the previous example, there exists a buffer between the two consequent stations that are processing the SubBatches. In such stations, there may be a case were the units constituting the SubBatches are processed separately. For reasons of simplicity we assume that the SubBatches are processed as a bulk group of units which cannot be further segregated. Contrary to the modelling practice though, operators perform work on each individual unit of the SubBatch. For fear that the units may get mixed up and thus “dirty” the SubBatches, it is a common practice to try keeping the Buffer before each Machine/station clear from other SubBatches other than the one being currently processed in the station. The object LineClearance is introduced to model this behaviour.

The flow described in Figure 9 is the same with the one used in the current example. The common Queue2 between Machine1 and Machine2 is replaced with the LineClearance object though.

ManPy object repository contains the following objects in order to model the described behaviour:

- **LineClearance**: inherits from the Queue generic object and overrides the *canAccept* and *canAcceptAndIsRequested* methods. These methods should now return true if the buffer is empty or if the predecessor requests to hand in a SubBatch with the same batchId as the ones that the buffer holds.

The data of this example are similar to example 4.10. The Queue2 Queue object is replaced by a LineClearance Queue with capacity of 2 SubBatches. In addition, the processing time of Machine2 is increased to 4 time units. This will eventually lead to an increased waiting time for Machine3.

Below is the ManPy main script to run this model (dream\simulation\Examples\ClearBatchLines.py):

```
from dream.simulation.imports import Machine, Source, Exit, Batch,
BatchDecomposition,\
                              BatchSource, BatchReassembly, Queue, LineClearance,
ExcelHandler, ExcelHandler
from dream.simulation.Globals import runSimulation

# define the objects of the model
S=BatchSource('S','Source',interarrivalTime={'distributionType':'Fixed','mean':1
.5}, entity='Dream.Batch', batchNumberOfUnits=100)
Q=Queue('Q','StartQueue',capacity=100000)
BD=BatchDecomposition('BC', 'BatchDecomposition', numberOfSubBatches=4,
processingTime={'distributionType':'Fixed','mean':1})
M1=Machine('M1','Machine1',processingTime={'distributionType':'Fixed','mean':0.5
})
Q1=LineClearance('Q1','Queue1',capacity=2)
M2=Machine('M2','Machine2',processingTime={'distributionType':'Fixed','mean':4})
BRA=BatchReassembly('BRA', 'BatchReassembly', numberOfSubBatches=4,
processingTime={'distributionType':'Fixed','mean':0})
M3=Machine('M3','Machine3',processingTime={'distributionType':'Fixed','mean':1})
E=Exit('E','Exit')

# define the predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[BD])
BD.defineRouting([Q],[M1])
M1.defineRouting([BD],[Q1])
Q1.defineRouting([M1],[M2])
M2.defineRouting([Q1],[BRA])
BRA.defineRouting([M2],[M3])
M3.defineRouting([BRA],[E])
E.defineRouting([M3])

def main():

    # add all the objects in a list
    objectList=[S,Q,BD,M1,Q1,M2,BRA,M3,E]
    # set the length of the experiment
    maxSimTime=1440.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime, trace='Yes')

    # print the results
    print "the system produced", E.numOfExits, "batches"
    working_ratio_M1 = (M1.totalWorkingTime/maxSimTime)*100
    blockage_ratio_M1 = (M1.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M1 = (M1.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M1.objName, "is", working_ratio_M1
```

```
    print "the blockage ratio of", M1.objName, 'is', blockage_ratio_M1
    print "the waiting ratio of", M1.objName, 'is', waiting_ratio_M1
    working_ratio_M2 = (M2.totalWorkingTime/maxSimTime)*100
    blockage_ratio_M2 = (M2.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M2 = (M2.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M2.objName, "is", working_ratio_M2
    print "the blockage ratio of", M2.objName, 'is', blockage_ratio_M2
    print "the waiting ratio of", M2.objName, 'is', waiting_ratio_M2
    working_ratio_M3 = (M3.totalWorkingTime/maxSimTime)*100
    blockage_ratio_M3 = (M3.totalBlockageTime/maxSimTime)*100
    waiting_ratio_M3 = (M3.totalWaitingTime/maxSimTime)*100
    print "the working ratio of", M3.objName, "is", working_ratio_M3
    print "the blockage ratio of", M3.objName, 'is', blockage_ratio_M3
    print "the waiting ratio of", M3.objName, 'is', waiting_ratio_M3
    ExcelHandler.outputTrace('TRACE')

    return {"batches": E.numOfExits,
            "working_ratio_M1": working_ratio_M1,
           "blockage_ratio_M1": blockage_ratio_M1,
           "waiting_ratio_M1": waiting_ratio_M1,
            "working_ratio_M2": working_ratio_M2,
           "blockage_ratio_M2": blockage_ratio_M2,
           "waiting_ratio_M2": waiting_ratio_M2,
            "working_ratio_M3": working_ratio_M3,
           "blockage_ratio_M3": blockage_ratio_M3,
           "waiting_ratio_M3": waiting_ratio_M3,
           }

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 89 parts
the working ratio of Machine1 is 12.6041666667
the blockage ratio of Machine1 is 87.3263888889
the waiting ratio of Machine1 is 0.0694444444444
the working ratio of Machine2 is 99.8958333333
the blockage ratio of Machine2 is 0.0
the waiting ratio of Machine2 is 0.104166666667
the working ratio of Machine3 is 6.18055555556
the blockage ratio of Machine3 is 0.0
the waiting ratio of Machine3 is 93.8194444444
```

The blockage ratio of Machine1 is drastically increased as LineClearance buffer of Machine2 has to be cleared from the the currently processed Batch first before it is loaded with SubBatches from a different Batch. ClearBatchLines.xls is also generated and has the following contents:

0	Batch0	generated
0	Batch0	released Source
0	Batch0	got into StartQueue
0	Batch0	released StartQueue
0	Batch0	got into BatchDecomposition
1	Batch0_SB_0	released BatchDecomposition

1	Batch0_SB_0	got into Machine1
1.5	Batch1	generated
1.5	Batch0_SB_0	ended processing in Machine1
1.5	Batch1	released Source
1.5	Batch1	got into StartQueue
1.5	Batch0_SB_0	released Machine1
1.5	Batch0_SB_0	got into Queue1
1.5	Batch0_SB_0	released Queue1
1.5	Batch0_SB_0	got into Machine2
1.5	Batch0_SB_1	released BatchDecomposition
1.5	Batch0_SB_1	got into Machine1
2.0	Batch0_SB_1	ended processing in Machine1
2.0	Batch0_SB_1	released Machine1
2.0	Batch0_SB_1	got into Queue1
2.0	Batch0_SB_2	released BatchDecomposition
2.0	Batch0_SB_2	got into Machine1
2.5	Batch0_SB_2	ended processing in Machine1
2.5	Batch0_SB_2	released Machine1
2.5	Batch0_SB_2	got into Queue1
2.5	Batch0_SB_3	released BatchDecomposition
2.5	Batch0_SB_3	got into Machine1
2.5	Batch1	released StartQueue
2.5	Batch1	got into BatchDecomposition
3.0	Batch2	generated

The notation Batch1, 2, etc. denote the Batches generated by the BatchSource Source. Respectively, the suffixes _SB_0, 2, etc. of the names in the second column denote each separate SubBatch belonging to Batch Batch0, 1, etc.

4.9 Output Analysis

Dream aims to offer methods for output analysis of the simulation results. This is currently work in progress existing in dream/simulation/outputanalysis. In order to be able to use this modules, R (http://www.r-project.org/) and Rpy2 (http://rpy.sourceforge.net/rpy2.html) should be installed.

As an example we demonstrate dream\simulation\Examples\TwoServersPlots.py that is similar to the TwoServers example but it also outputs a pie that presents graphically the percentage of time that the repairman is busy or idle.

The new entries on the code are:

- In the beginning the Graphs module is imported:

```
from dream.KnowledgeExtraction.Plots import Graphs
```

- After the simulation run the values for the pie are calculated:

```
#calculate the percentages for the pie
working_ratio = (R.totalWorkingTime/maxSimTime)*100
waiting_ratio = (R.totalWaitingTime/maxSimTime)*100
```

- Then a Graph object is created and the *Pie* method is called in order to create the output file

```
#create a graph object
graph=Graphs()
#create the pie
graph.Pie([working_ratio,waiting_ratio], "repairmanPie.jpg")
```

Running the script the user gets in addition to the console output repairmanPie.jpg that contains the following graph:

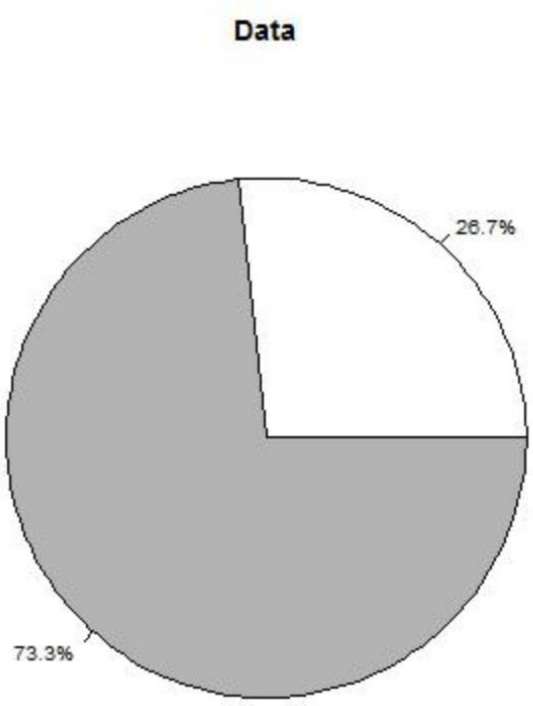

Figure 10: the pie chart of repairman utilisation

4.10 Shifts

4.10.1 A simple shift pattern

It is typical in production lines to have shifts in our stations. We consider a system like the one of the single server example (Figure 2), but now our server will also have off-shift time. Shifts are handled in ManPy with the ShiftScheduler object which is of ObjectInterruption type.

To make the first example simple, we will run the simulation for 20 time units. We suppose that the machine starts on shift, its shift lasts 5 units and then it is on shift for 5 units and so on. The interarrival time is 0.5 and the processing time of the machine 3.

Below is the ManPy main script to run this model (dream\simulation\Examples\ServerWithShift1.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part, ShiftScheduler
from dream.simulation.Globals import runSimulation

#define the objects of the model
S=Source('S1','Source',interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
M=Machine('M1','Machine', processingTime={'distributionType':'Fixed','mean':3})
E=Exit('E1','Exit')

SS=ShiftScheduler(victim=M, shiftPattern=[[0,5],[10,15]])

#define predecessors and successors for the objects
S.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[S],successorList=[E])
E.defineRouting(predecessorList=[M])

def main():

    # add all the objects in a list
    objectList=[S,M,E,SS]
    # set the length of the experiment
    maxSimTime=20.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio = (M.totalWorkingTime/maxSimTime)*100
    off_shift_ratio=(M.totalOffShiftTime/maxSimTime)*100
    print "the total working ratio of the Machine is", working_ratio, "%"
    print "the total off-shift ratio of the Machine is", off_shift_ratio, "%"
    return {"parts": E.numOfExits,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
the system produced 3 parts
the total working ratio of the Machine is 50.0 %
the total off-shift ratio of the Machine is 50.0 %
```

We see that the shift pattern is defined as a list of lists with the following structure [[start_of_shift1, end_of_shift1], [start_of_shift2, end_of_shift2], … [start_of_shiftN, end_of_shiftN]]. ShiftScheduler does not accept stochastic shift patterns.

If the Machine was always available, it would be always working, since parts arrive at higher rate than its processing. It would also process 6 parts. However, now it is 50% off shift and it produced only 3 parts that could finish in these 10 on shift units.

4.10.2 A repeated shift pattern

The previous example it would be cumbersome to model the system for a greater running time. It is common to have shift patterns that repeat in time. Current ShiftScheduler needs all the information on the pattern, so the whole list described above. Nonetheless, the user can easily develop the list programmatically as in the below model (dream\simulation\Examples\ServerWithShift2.py):

```
from dream.simulation.imports import Machine, Source, Exit, Part, ShiftScheduler
from dream.simulation.Globals import runSimulation

#define the objects of the model
S=Source('S1','Source',interarrivalTime={'distributionType':'Fixed','mean':0.5},
entity='Dream.Part')
M=Machine('M1','Machine', processingTime={'distributionType':'Fixed','mean':3})
E=Exit('E1','Exit')

# create a repeated shift pattern
shiftPattern=[]
i = 0
while i<100:
    shiftPattern.append([i,i+5])
    i+=10
print shiftPattern

#create the shift
SS=ShiftScheduler(victim=M, shiftPattern=shiftPattern)

#define predecessors and successors for the objects
S.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[S],successorList=[E])
E.defineRouting(predecessorList=[M])

def main():

    # add all the objects in a list
    objectList=[S,M,E,SS]
    # set the length of the experiment
    maxSimTime=100.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print "the system produced", E.numOfExits, "parts"
    working_ratio = (M.totalWorkingTime/maxSimTime)*100
    off_shift_ratio=(M.totalOffShiftTime/maxSimTime)*100
    print "the total working ratio of the Machine is", working_ratio, "%"
    print "the total off-shift ratio of the Machine is", off_shift_ratio, "%"
    return {"parts": E.numOfExits,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
[[0, 5], [10, 15], [20, 25], [30, 35], [40, 45], [50, 55], [60, 65],
[70, 75], [80, 85], [90, 95]]
the system produced 16 parts
the total working ratio of the Machine is 50.0 %
the total off-shift ratio of the Machine is 50.0 %
```

The shift pattern was externally defined in a loop. This way there is flexibility to define whatever pattern for the simulation time.

4.10.3 Ending unfinished work at the end of the shift

It is common that if a server is processing at the end of the shift it would end this processing and then go off-shift. To model this behaviour ShiftSheduler has the *endUnfinished* flag. Default value is False. If in the model of the first example of this section we want to define that the ShiftSheduler we just change its definition to the below (dream\simulation\Examples\ServerWithShift3.py):

```
SS=ShiftScheduler(victim=M, shiftPattern=[[0,5],[10,15]], endUnfinished=True)
```

Running the model we get the following in our console:

```
the system produced 4 parts
the total working ratio of the Machine is 60.0 %
the total off-shift ratio of the Machine is 40.0 %
```

Comparing with the first example we see that now the Machine is off-shift only 40% of the time. This is because at the time it was to go off-shift (e.g. at 5) it was processing a Part. So it went off-shift after it ended (at 6). Also we see that 4 Parts were now produced.

4.10.4 Not accepting work if the shift is ending

It is also common that servers will not accept new work near the end of shift. This does not mean that they go off-shift, since if they are processing something they will not stop (even if *endUnfinished* is set to False), just that they will not commence a new processing. To model this behaviour the ShiftScheduler has the *receiveBeforeEndThreshold* attribute, which has default value of 0. Setting this to 3 it means that the Machine will not accept new parts 3 units or closer to the end of the shift. The definition is (dream\simulation\Examples\ServerWithShift4.py):

```
SS=ShiftScheduler(victim=M, shiftPattern=[[0,5],[10,15]],
receiveBeforeEndThreshold=3)
```

Running the model we get the following in our console:

```
the system produced 2 parts
the total working ratio of the Machine is 30.0 %
the total off-shift ratio of the Machine is 50.0 %
```

We see now that only 2 parts were produced. The Machine ended processing of a part at 3 and did not accept the next one because of the threshold. The same happened at 13.

4.11 Generation of events at specific moments

In discrete event simulation, events happen at specific moments in time based on distributions. For example, if a Machine has a defined distribution for its processing time, then each time it receives a Part at time *t* it will create a number *dt* out of this distribution. So the event of the end of processing will be scheduled for *t+dt*.

Nevertheless, there are certain models that require events to be triggered at specific time intervals. For these situations ManPy employs the EventGenerator object. The functionality of the object is the following:

- It sleeps for a specific interval
- When it is activated it invokes a method

In the following example we will demonstrate the EventGenerator.

4.11.1 Balancing a buffer

Let's assume that we have the simple buffer-server system we seen in the examples of setting WIP (Figure 5). In our system a supervisor comes exactly every 10 minutes and checks the Queue. If the Queue is empty he adds 5 parts in it. Otherwise he does nothing.

The easiest way to model this behaviour is to make use of the EventGenerator object. The full example is given below (dream\simulation\Examples\BalancingABuffer.py):

```
from dream.simulation.imports import Machine, Queue, Exit, Part, EventGenerator
from dream.simulation.Globals import runSimulation, setWIP, G

# method to check if the buffer is starving and refill it
def balanceQueue(buffer, refillLevel=1):
    # get the internal queue of the buffer
    objectQueue=buffer.getActiveObjectQueue()
    numInQueue=len(objectQueue)
    print '-'*50
    print 'at time=', G.env.now
    # check if the buffer is empty and if yes fill it with 5 parts
    if numInQueue==0:
        print 'buffer is starving, I will bring 5 parts'
        for i in range(refillLevel):
            # calculate the id and name of the new part
            partId='P'+str(G.numOfParts)
            partName='Part'+str(G.numOfParts)
            # create the Part
            P=Part(partId, partName, currentStation=buffer)
            # set the part as WIP
            setWIP([P])
            G.numOfParts+=1
    # else do nothing
    else:
        print 'buffer has', numInQueue, 'parts. No need to bring more'

#define the objects of the model
Q=Queue('Q1','Queue', capacity=float('inf'))
M=Machine('M1','Machine', processingTime={'distributionType':'Fixed','mean':6})
E=Exit('E1','Exit')
EV=EventGenerator('EV', 'EntityCreator', start=0, stop=float('inf'),
interval=20,method=balanceQueue,
                  argumentDict={'buffer':Q, 'refillLevel':5})

# counter used in order to give parts meaningful ids (e.g P1, P2...) and names
(e.g. Part1, Part2...)
```

```
G.numOfParts=0

#define predecessors and successors for the objects
Q.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[Q],successorList=[E])
E.defineRouting(predecessorList=[M])

def main():
    # add all the objects in a list
    objectList=[Q,M,E,EV]
    # set the length of the experiment
    maxSimTime=100.0
    # call the runSimulation giving the objects and the length of the experiment
    runSimulation(objectList, maxSimTime)

    #print the results
    print '='*50
    print "the system produced", E.numOfExits, "parts"
    working_ratio = (M.totalWorkingTime/maxSimTime)*100
    print "the total working ratio of the Machine is", working_ratio, "%"
    return {"parts": E.numOfExits,
          "working_ratio": working_ratio}

if __name__ == '__main__':
    main()
```

Running the model we get the following in our console:

```
--------------------------------------------------
at time= 0.0
buffer is starving, I will bring 5 parts
--------------------------------------------------
at time= 20.0
buffer has 1 parts. No need to bring more
--------------------------------------------------
at time= 40.0
buffer is starving, I will bring 5 parts
--------------------------------------------------
at time= 60.0
buffer has 1 parts. No need to bring more
--------------------------------------------------
at time= 80.0
buffer is starving, I will bring 5 parts
==================================================
the system produced 13 parts
the total working ratio of the Machine is 80.0 %
```

Notes on the code:

- Additionally to *id* and *name*, we gave the EventGenerator the following attribute:
    - *start*: the simulation time that the EventGenerator will be activated first
    - *stop*: the simulation time that the EventGenerator will stop. We gave this infinite so that it is active all through the simulation time. This is also the default value
    - *interval*: the interval for which the EventGenerator sleeps before invoking the method again
    - *method*: the method that the EventGenerator invokes.
    - *argumentDict*: the arguments of the method. This should be provided in a dictionary (*keyword:value*)

- *balanceQueue* is the method we wrote in order to implement the behaviour that is conducted every 20 minutes. The logic is simple enough. A couple of notes:
    - The method gets the buffer as argument. Then it obtains the internal queue using the *getActiveObjectQueue* method.
    - In order to add the parts in the Queue, *Globals.setWIP* method is utilized (also imported in the beginning). This method needs a list of Entities as argument, so even if every time we give one Entity, we give it in a list. Note that appending the object in the internal Queue would not be enough, since *setWIP* sends also the *canDispose* signal to the Queue. So if the Part was appended manually then the user should also send the signal.