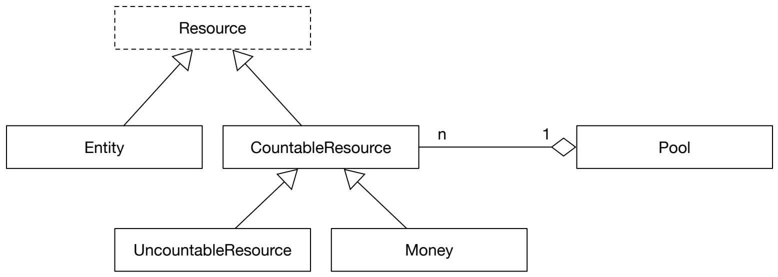
MASON Discrete Event Simulation Extension

Sean Luke
Department of Computer Science
George Mason University

Manual Version 2
June 2023

Contents



0 Introduction

0.1 Installing the DES Extension

0.2 What You Need to Know about MASON

0.3 About Discrete Event Simulation

0.4 About Hybrid ABM and DES Simulation

1 Resources

The DES Extension has two basic kinds of classes: **Resources** and **Processes**. Processes perform actions in response to events. In most cases, this action involves Processes handing off Resources to other Processes. This could be for many reasons. Perhaps a Resource moving from Process A to Process B to another represents A paying B for some service. Perhaps Processes A and B are warehouses and the Resource represents a truck moving from one to the other. Perhaps A and B are countries and the Resource represents a family migrating. Maybe A and B are spies and the Resource holds a critical communication between them.

Resources are also held in **Pools**. A Pool can be dipped into by certain Processes to allocate a Resource for some temporary function. For example, imagine if a factory floor had four assembly lines but only two lathes. Widgets moving through an assembly line at some point had to be lathed, and if a lathe was not available, the widget (and its assembly line) would have to wait until one came available.

To do this, there are two special kinds of Processes, copies of which are part of each assembly line, called **Lock** and **Unlock**. For a widget to pass through a Lock, the Lock must first allocate a Resource (representing the lathe) from a shared Pool. If it cannot, the widget must wait until the Lock can allocate the Resource. Then the widget may pass through to other Processes. Ultimately when widget is finished with the lathe, the widget can pass through an **Unlock** Process, which gives a Resource back to the Pool. The Pool only holds two lathe Resources.[1]

MASON provides several kinds of Resources, depending on your modeling needs:

- **Entities** are atomic elements, like tickets or tokens or cars. They cannot be broken up into smaller Entities, nor joined together to form larger ones. They are not worth different amounts: and Entity is an Entity.
- **Composite Entities** are Entities which hold a collection of other Resources inside them, plus a manifest. For example, a Composite Entity might be used to model a shipping container filled with teddy bears. The manifest can indicate anything you'd like: expiration dates, country of origin, who manufactured each bear, the amount of love that went into each bear, and so on. There is a special Process called a **Composer** which takes a collection of Resources and builds a Composite Entity

[1]These are called Lock and Unlock because they are reminiscent of locking or unlocking on a mutex or a semaphore in computer science terminology. So sue me, I'm a computer science professor. Others in the Discrete Event Simulation world might instead refer to **seizing** and **releasing** resources from a pool.

- **Countable Resources** are Resources which can be divided, but only as integers. For example, packages of medicine pills are countable resources: the package could be 0 pills, or 1 pill, or 2 pills, or 1500 pills. But it cannot be 1.5 pills. You cannot have a negative or infinite value.
- **Money** is an obvious example of a Countable Resource. In US currency, the fundamental unit of money is the penny. You cannot have a half a penny, and you cannot have 3.279 pennies. Because money is so common, MASON has a dedicated kind of Countable Resource just to represent it: it prints it out in a pleasing fashion (like "$14.23" for 1423 pennies).
- **Uncountable Resources** are Resources which can be divided indefinitely. An example of an uncountable resource is water. You can have 3.239122 liters of water, and you can divide that into five parts any way you like, including some parts holding 0 liters. You cannot have a negative or infinite value.

Resources are typed, and you can have as many different types of Resources as you like. For example, you could have a type of Countable Resource representing pills and another one representing punches in the face. You could have a type of Uncountable Resource representing water and another one representing love. You could have both US dollars and Hong Kong dollars. You could have both cargo containers and suitcases of stuff. These types are not exchangeable: you can't give a Process water when it's expecting love. However there exists a special Process, called a **Transformer** which can convert certain Resources into Countable Resources: for example, it could be used as a currency converter.

If Resources are typed, how do you make a new type of Resource, and how do you make more of that same type? Each Resource type has a unique name. To make a new type of resource, just instantiate a Resource class with that new name. Later Resources of that type are allocated by copying them from earlier Resources of the same type. This can be done either by calling `duplicate()` or by calling a copy constructor on the earlier Resource, indicating the amount of the new Resource.

Processes deal with Resources differently based on the kind of Resources. For example, many Processes store Resources to eventually hand off to downstream Processes. If the Resource is an Entity, then it is atomic and the Process must store it in a collection of Entities. If the Resource is a Countable or Uncountable Resource, then the Process can simply merge it into one pile or blob of Countable (or Uncountable) Resource, and dole it out as appropriate later on.

All Resources have **amounts**. For Countable or Uncountable resources, the amount is simply the size of the resource: such as 1500 pills or 1423 pennies or 3.239122 liters of water. For Entities, including Composite Entities, the amount is always 1.

1.1 Resource

Resource is the abstract superclass of all resources. All Resources have a **unique type** (an integer) shared by resources of that type, and a **name** (a String) which you can stipulate. For example, all Electricity resources might have type 0, and all Gas resources might have type 1 and all Cars might have type 2 and all Dollars might have type 3. It is possible to construct two Resources with the same name, but they will have different types even so: don't do that. For your own sanity, make sure that you only construct a Resource once with a given name, and copy Resources from it to make more of the same type.

sim.des.Resource Constructor Methods

`public Resource(String name)`
Builds a new Resource, of a new unique type, with the given name.

`protected Resource()`
Builds a new Resource, but does not set the name or type. This exists to permit copy constructors in subclasses. Normally you'd leave it alone.

---

sim.des.Resource Methods

public void clear()
: Clears the resource. What this does varies depending on the subclass.

public void toString()
: Prints the resource in a pleasing manner.

public boolean equals(Object other)
: Returns true if *other* is a Resource, is not null, and if it is both the same type and amount as this Resource. The storage and info elements inside Composite Entities are not checked for equality.

public int hashCode()
: Returns an appropriate hash code for hash tables.

public double getAmount()
: Returns the amount of the Resource. Note that this is always double, even for Countable Resources and for Entities, both of which return integers.

public boolean isSameType(Resource other)
: Returns true if *other* is not null and the same type as this Resource.

public int getType()
: Returns the type of this resource.

public String getName()
: Returns the name of this resource.

protected void setName(String name)
: Sets the name of this resource. You shouldn't call this probably.

public Resource duplicate()
: Exactly duplicates the Resource once and returns the result. The storage and info objects are not copied — just pointer-copies. You'll have to deep copy them as you see fit.

public Resource[] duplicate(int times)
: Exactly duplicates the Resource *times* times and returns the result. The storage and info objects are not copied — just pointer-copies. You'll have to deep copy them as you see fit.

1.2 Entity

An Entity is a Resource that cannot be subdivided into smaller Resources of the same time. For example: a Car might be an Entity. But Water is not, as you can divide Water up into smaller amounts. Entities can **store** other resources inside them: that is, they can be **composed** of them. For example, a given Car might contain Wheels, and Engine, and some amount of Gasoline. Different Cars are permitted to store different things. Entities can also contain **info** objects, essentially manifests.

sim.des.Entity Constructor Methods

public Entity(String name)
: Builds a new Entity, of a new unique type, with the given name.

public Entity(Entity other)
: Makes a copy of the Entity, including its type and name. The storage and info objects are pointer-copied, not deep-copied. This is the standard copy constructor for Entity.

Here are the custom methods for Entity. It also implements the methods defined in Resource.

sim.des.Entity Methods

public Resource[] getStorage()
: Returns the actual storage array of the Entity, or null if there isn't one.

public void setStorage(Resource[] val)
: Sets the storage array of the Entity. This can be set to null.

public Object getInfo()
: Returns the actual info object of the Entity, or null if there isn't one.

public void setInfo(Object val)
: Sets the info object of the Entity. This can be set to null.

public boolean isComposite()
: Returns true if the storage is non-null.

public void clear()
: Sets the storage to null.

public double getAmount()
: Always returns 1.0.

public boolean equals(Object other)
: Returns true if *other* is an Entity, is not null, and if it is the same type as this Entity. The storage and info elements inside Composite Entities are not checked for equality.

public int hashCode()
: Returns an appropriate hash code for hash tables, not considering the storage or info elements.

1.3 CountableResource

A CountableResource is a Resource that has an **amount** (an integer) and can be subdivided into CountableResources with smaller integer amounts. For example: a Population might be divided into smaller subpopulations. There is an atomic, smallest, non-divisible amount of CountableResources: 1. You can also set the amount to 0.

Even though CountableResource only stores integers for amounts, it stores them as doubles. This means that its maximum integer value is larger than that of an int. Specifically, it is equal to sim.des.CountableResource.MAXIMUM_INTEGER

sim.des.CountableResource Constructor Methods

public CountableResource(String name, double intialAmount)
: Builds a new CountableResource, of a new unique type, with the given name and amount.

public CountableResource(String name)
: Builds a new CountableResource, of a new unique type, with the given name and zero amount.

public CountableResource(CountableResource other)
: Makes a copy of the CountableResource, including its type, name, and amount. If the provided object is actually an UncountableResource, this will throw an exception.

public CountableResource(CountableResource other, double intialAmount)
: Makes a copy of the CountableResource, including its type and name, but with the new amount provided. If the provided object is actually an UncountableResource, this will throw an exception.

Here are the custom methods for CountableResource. It also implements the methods defined in Resource.

sim.des.CountableResource Methods

public double doubleValue()
: Returns the amount as a double value.

public boolean isUncountable()
: Returns false (note that this is overridden by UncountableResource).

public boolean isCountable()
: Returns true (note that this is overridden by UncountableResource).

public double getAmount()
: Returns the amount stored in the CountableResource, which will always be a nonnegative integer (but possibly larger than an int).

public void clear()
: Sets the amount to 0.

public double getAmount()
: Returns the amount.

public void setAmount(double val)
: Sets the amount

public void bound(double min, double max)
: Bounds the resource to be no more than max and no less than min. It must be the case that max $\geq$ min ≥ 0.

public void bound(double max)
: Bounds the resource to be no more than max and no less than zero. It must be the case that max ≥ 0.

public boolean increase(double val)
: Increases the amount by the given value and returns true, unless val is not an integer, or unless this would exceed sim.des.CountableResource.MAXIMUM_INTEGER, in which case nothing happens and false is returned.

public boolean decrease(double val)
: Decrements the amount by the given value and returns true, unless val is not an integer, or unless this would drop below zero, in which case nothing happens and false is returned.

public boolean increment()
: Increments the amount by 1 and returns true, unless this would exceed sim.des.CountableResource.MAXIMUM_INTEGER, in which case nothing happens and false is returned.

public boolean decrement()
: Decrements the amount by 1 and returns true, unless this would drop below zero, in which case nothing happens and false is returned.

public CountableResource reduce(double atLeast, double atMost)
: Subtracts at least a certain amount and at most a certain amount from this CountableResource, placing that amount into a new CountableResource and returning it, unless there is not enough amount to do so, in which case nothing happens and false is returned. atLeast and atMost must be integers, with atMost $\geq$ atLeast ≥ 0.

public CountableResource reduce(double byExactly)
: Subtracts exactly a certain amount from this CountableResource, placing that amount into a new CountableResource and returning it, unless there is not enough amount to do so, in which case nothing happens and false is returned. byExactly must be an integer ≥ 0.

public void add(CountableResource other)
: Adds the other CountableResource's amount into this one, setting the other CountableResource's amount to zero afterwards.

public void add(CountableResource other, double atMostThisMuch)
: Adds atMostThisMuch of the other CountableResource's amount into this one, setting the other CountableResource's amount to the remainder afterwards.

public void add(CountableResource[] other)
Adds all the other CountableResources' amounts into this one, setting the other CountableResources' amounts to zero afterwards.

public boolean greaterThan(CountableResource other)
Returns true if this amount is greater than the other resource's amount.

public boolean greaterThanOrEquals(CountableResource other)
Returns true if this amount is greater than or equal to the other resource's amount.

public boolean lessThan(CountableResource other)
Returns true if this amount is less than the other resource's amount.

public boolean lessThanOrEquals(CountableResource other)
Returns true if this amount is less than or equal to the other resource's amount.

public int compareTo(Object other)
Returns 0 if the other object's amount is equal to, -1 if it is greater than, and 1 if it is less than my amount.

1.4 Money

Money is a CountableResource with a cute printing function which prints it with a currency sign. For example, a Dollar is Money, hence a CountableResource, where the smallest amount (1) would be the Cent. Perhaps it should be renamed Cents.

sim.des.Money Constructor Methods

public Money(String name, double intialAmount)
Builds a new Money, of a new unique type, with the given name and amount. The name will be used as a currency symbol during printing.

public Money(String name)
Builds a new Money, of a new unique type, with the given name and zero amount. The name will be used as a currency symbol during printing.

public Money(Money other)
Makes a copy of the Money, including its type, name, and amount.

public Money(Money other, double intialAmount)
Makes a copy of the Money, including its type and name, but with the new amount provided.

Here are the custom methods for Money. It also implements the methods defined in CountableResource.

sim.des.Money Methods

public String toString()
Returns the amount in decimal format as $CX.Y$, where C is the name (a currency symbol), X is the amount / 100, and Y is the amount mod 100. For example, if "$" was the name, and the amount was 1423, then this would print as "$14.23"

1.5 UncountableResource

An UncountableResource is a CountableResource which can be subdivided infinitely and has real-valued amounts. Thus while you can only have a CountableResource with values 0, 1, 2, 3, ..., an UncountableResource could be any positive amount, such as 0 or 2.34129 or 92.3 or Infinity.

If you have a variable holding a CountableResource, how do you know it is a CountableResource versus an UncountableResource? You could use `instanceof`, or you could call `isCountable()` or `isUncountable()`.

sim.des.UncountableResource Constructor Methods

public UncountableResource(String name, double intialAmount)
Builds a new UncountableResource, of a new unique type, with the given name and amount.

public UncountableResource(String name)
Builds a new UncountableResource, of a new unique type, with the given name and zero amount.

public UncountableResource(UncountableResource other)
Makes a copy of the UncountableResource, including its type, name, and amount.

public UncountableResource(UncountableResource other, double intialAmount)
Makes a copy of the UncountableResource, including its type and name, but with the new amount provided.

Here are the custom methods for UncountableResource. It also implements the myriad of methods defined in CountableResource.

sim.des.UncountableResource Methods

public UncountableResource[] divide(int times)
Divides the amount by *times*, which must be > 0, and builds *times* -1 new UncountableResources, each with the new amount. Also reduces the amount of this UncountableResource to the amount as well.

public UncountableResource halve()
Builds a new UncountableResources with half the amount, leaving the other half as the current amount.

public void scale(double value)
Changes the amount by multiplying it by the given value.

public boolean increase(double val)
Increases the amount by the given value and returns true.

public boolean decrease(double val)
Decrements the amount by the given value and returns true, unless this would drop below zero, in which case nothing happens and false is returned.

1.6 Pool

A **Pool** is a storage of a single kind of CountableResource, UncountableResource, or Money. Various Process objects, namely **Lock** and **Unlock**, dip into a shared Pool to add or remove resources or to wait until resources have come available.

Pool is very simple: it has a **maximum** amount of resource, a **minimum** (always zero), and an **initial resource allocation**. Beyond that, its current resource can be set and queried. And that's it!

A Pool extends sim.des.portrayal.DESPortrayal, so it can serve as a MASON SimplePortrayal. DESPortrayal objects are also sim.des.Named, so the Pool can be given a name via getName() and setName(...).

A Pool is sim.des.Resettable, meaning that it can be reset to its original state via a method called reset(...).

sim.des.Pool Constructor Methods

public Pool(CountableResource resource, double maximum)
: Builds a Pool of the given type of CountableResource, with an initial amount copied from the CountableResource, and the provided maximum.

public Pool(CountableResource resource)
: Builds a Pool of the given type of CountableResource, with an initial amount copied from the CountableResource, and a maximum of infinity (if it's an UncountableResource), or CountableResource.MAXIMUM_INTEGER (if it's a CountableResource or Money).

public Pool(double initialResourceAllocation)
: Builds a Pool of a new type of CountableResource, with an initial amount as specified, and a maximum of infinity (if it's an UncountableResource), or CountableResource.MAXIMUM_INTEGER (if it's a CountableResource or Money).

public Pool(double initialResourceAllocation, double maximum)
: Builds a Pool of a new type of CountableResource, with an initial amount as specified, and the specified maximum.

A Pool only has a few methods:

sim.des.Pool Methods

public void reset(SimState state)
: Resets the Pool. This simply resets its current amount to the initial amount: the name and maximum stay as you had last set them.

public CountableResource getResource()
: Returns the current available resource.

public void setResource(CountableResource val)
: Sets the current available resource.

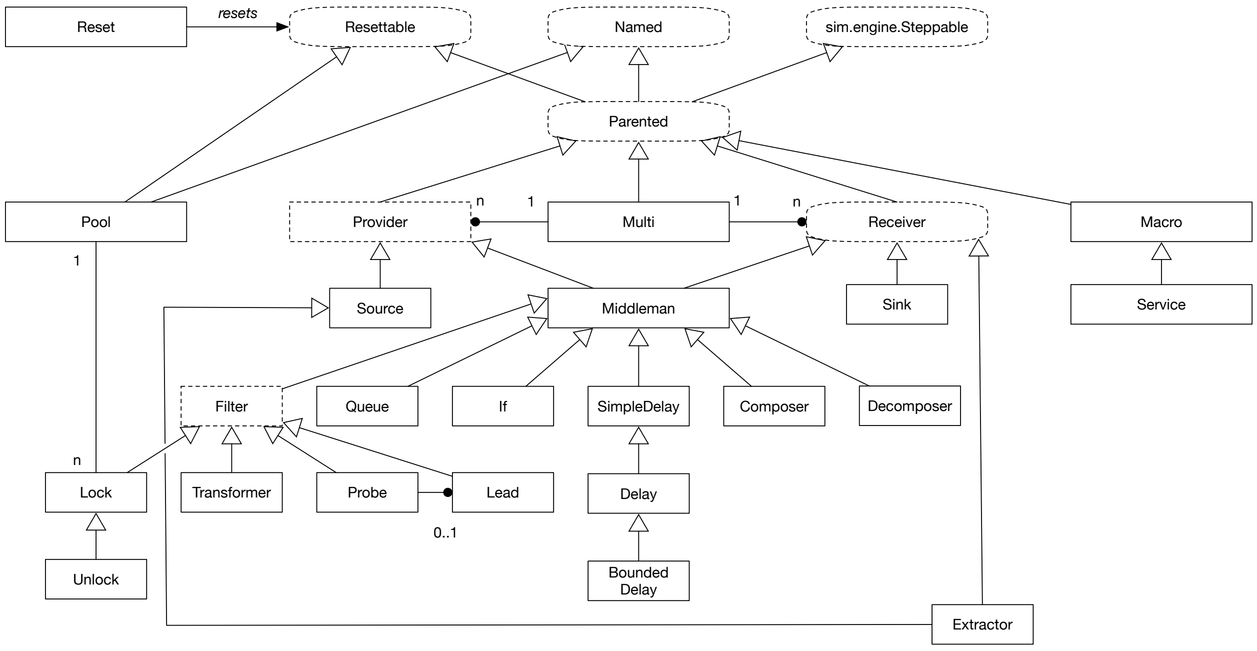
public double getMaximum()
: Returns the maximum allowed resource.

public void setMaximum(double value)
: Sets the maximum allowed resource.

public String getName()
: Returns the name (which can be null).

public void getName(String name)
: Sets the name (which can be null).

public String toString()
: Returns the Pool and its state in a pleasing fashion.



2 Processes

Processes are the verbs of the DES system. They perform actions in response to *events* (or if you like, *signals*) they receive.

An event can be one of two things typically:

- Another Process can *offer* some Resources to the Process.
- Another Process can *request* that the Process offer it a resource.
- Another Process can *offer a transaction* (an exchange) of a Resource for some other Resource.
- MASON's Schedule can send an event to the Process to wake it up to do some work.

Various Process objects can be **Receivers** or **Providers** or both, or compose these objects into groups using the **Macro** and **Multi** classes. Some Process objects can be placed on MASON's schedule to be stepped when appropriate. Other Processes do not require the schedule to pass work through them and onto other Processes.

The Processes, and support objects, implement a few interfaces.

2.1 sim.engine.Steppable

An object which can be placed on MASON's schedule to be stepped in the future. All processes are Steppable, but only a few need it. [Perhaps in the future we should be more selective in this regard].

2.2 Named

An object which has a **name** (a String).

2.3 Parented

An object which can be part of a parent process. For example, Macros may contain many Parented objects within them, and are their parents.

2.4 Resettable

An object which can be **reset** by a **Reset** object.

2.5 Reset

Collects a bunch of **Resettable** objects and allows you to easily **reset** all of them. This way you can create your DES graph in the model's constructor, then reset it each time you **start()** the model, by just calling **reset()** on your Reset in the body of the start() method.

2.6 Provider

An object which can make offers to one or more Receivers. Receivers are **registered** with the Provider. When the Provider has a resource to offer, it will go to each of its Receivers according to some **offer policy** and ask them to **accept** offers of the Resource until it is depleted. Providers can also be asked by a Receiver to make an offer to it. Providers have a **typical Resource type** of the resource that they offer. Rarely, a Provider can provide more than one resource.

2.7 Receiver

An object which can *receive* an offer from a Provider. That is, the Provider will offer the Receiver some amount of Resource, asking it to **accept** the offer. Offers have minimum and maximum amounts that the Receiver may select from. The receiver accepts some appropriate amount (between the min and max) of the Resource being offered by the Provider, or it refuses it. A Receiver can also go to a Provider and **ask** that the Provider make an offer to the Receiver. Receivers have a **typical Resource type** of the resource that they accept. Rarely, a Receiver can receive more than one resource.

2.8 Sink

A Receiver which accepts all offers (of the appropriate Resource type) and throws the resulting Resource away. **By default, this class's step() method does nothing, so there's no need to schedule it.**

2.9 Source

A Provider which generates Resources and offers it to downstream registered Receivers. You can customize your Source however you want, but the default form works as follows. Each time the Source is ready to **produce** some Resource, the amount it produces is determined either by a **distribution** or a **deterministic amount**. It then adds this Resource to a pile, and offers all the Resources currently in the pile to its receivers. Sources have a maximum **capacity** for their pile and will not generate more than this. After making offers, the Source then uses *another* distribution or deterministic amount to determine the **next time** it will generate Resources and make offers. It schedules itself appropriately.

2.10 MiddleMan

An object which is both a Provider and a Receiver. You can make your own Provider+Receiver combination, but MiddleMan is a nice abstract superclass.

2.11 Extractor

Both a Receiver and a Source (sort of). An Extractor is works like a Source, except that it “builds” its resources by extracting them from another Provider. You could, for example, attach an Extractor to a Queue and every once i a while the Extractor would pull out of the Queue and offer resources to downstream Receivers. Extractors ignore capacity.

2.12 Queue

Both a Receiver and Provider. When a Queue receives a Resource, it adds it to a pile, then (usually) immediately offers this pile to its own receivers. The Queue’s pile has a maximum capacity. If it cannot add to the pile, it will refuse the offer.

2.13 SimpleDelay

Both a Receiver and Provider. When a SimpleDelay receives a Resource, it adds it to a linked list and, after a delay of some fixed amount of time, then offers the Resource to downstream receivers. If the receivers do not accept, then the Resource is discarded. The SimpleDelay has a maximum capacity of items it may store which are presently delayed.

2.14 Delay

A subclass of SimpleDelay. This is just like a SimpleDelay, except that the delay time for each object received can be *different*, so some Resources can move through the Delay faster than others. Thus the SimpleDelay is implemented using a heap rather than a linked list.

2.15 Composer

Both a Receiver and Provider. Upon receiving a Resource, it adds it to a collection of Resources. These Resources can be of different types. When it has all the necessary amounts of Resources for each type, it gathers them together and produces a Composed Entity holding these Resources, then offers the Entity to downstream receivers. If they do not accept, then the Entity is discarded. The Composer has both minimum and maximum (capacity) amounts for each of the Resources needed to compose into the Entity.

2.16 Decomposer

Both a Receiver and Provider. Upon receiving a Composed Entity, it decomposes it and offers each of its Resources to various downstream receivers, one registered for each Resource type. If they do not accept, then that Resource is discarded. **By default, this class’s step() method does nothing, so there’s no need to schedule it.**

3 Filters

Filters are just Processes which accept offers and immediately turn around and offer them to a single downstream Receiver. Filters are both Providers and Receivers, and never respond to requests to make offers.

3.1 Filter

The abstract superclass of Filters, which simply gathers together variables and methods common to them.

3.2 Transformer

Upon receiving a Resource, Transformer **transforms** the Resource into a different one according to a certain rule, then immediately offers it to downstream receivers. If they do not accept, then the Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

3.3 Lock

Upon receiving a Resource, Lock attempts to **allocate** some amount of *a different* Resource from a **Pool**. If successful, it then offers the original Resource to downstream receivers. If they do not accept, then that Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

3.4 Unlock

A subclass of Lock, but only for convenience's sake. Upon receiving a Resource, it **provides** some amount of *a different* Resource to a **Pool**. Regardless of whether the Pool accepts this generosity, Unlock then offers the original Resource to downstream receivers. If they do not accept, then that Resource is discarded. **By default, this class's step() method does nothing, so there's no need to schedule it.**

3.5 Probe

An object which gathers statistics on the resources which pass through it. A Probe can also be used in conjunction with a **Lead**.

3.6 Lead

An object detached from Probe with can be used in combination with it to determine the amount of resources in a DES subgraph between them.

3.7 If

A Filter which receives and incoming offer, then chooses only one of several possible Receivers to hand the offer to. This choice is specified in a method defined by the modeler.

4 Process Utilities

4.1 Macro

An object which can store a **subgraph** of the DES graph. This subgraph consists of some **Receivers**, some **Providers**, and some **in-between Processes**. When stepped, the Macro steps all of its stored objects in order. You can access the Receivers and Providers to make offers or register stuff as you see fit.

4.2 Service

A simple example of a common Macro consisting of a Lock, then a SimpleDelay, then an Unlock.