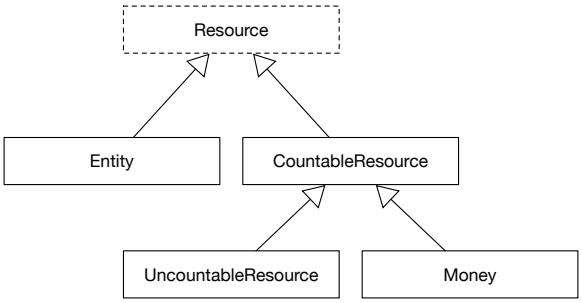
DES documentation

Sean Luke
Department of Computer Science
George Mason University

Manual Version 1
November 2021

Contents



0.0 Resources

Resources have two functions. They either flow from Providers to Receivers, or they reside in a Pool shared by various Lock and Unlock objects.

0.0.1 Resource

Resource is the abstract superclass of resources. All Resources have a **unique type** (an int) shared by resources of that type, and a **name** (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.

0.0.2 Entity

An Entity is a Resource that cannot be subdivided. 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.

0.0.3 CountableResource

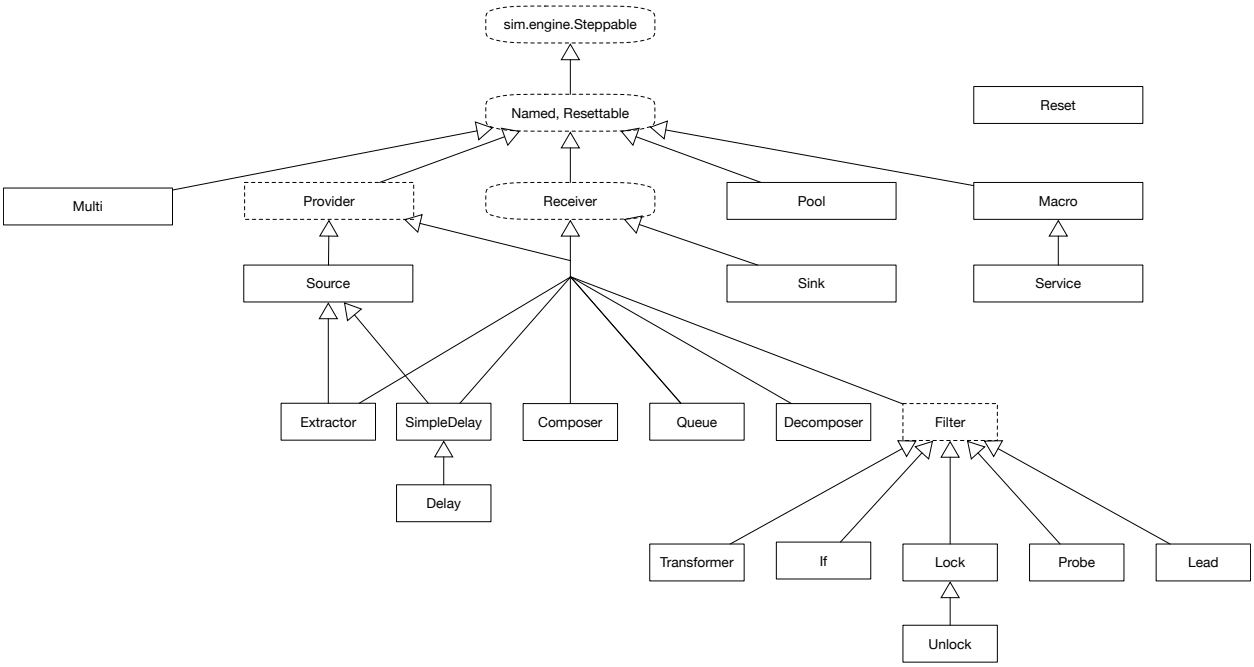
A CountableResource is a Resource that has an **amount** (an integer) and can be subdivided into CountableResources with smaller integer amounts. For example: a Population might be divided into smaller subpopulations. There is an atomic, smallest, non-divisible amount of CountableResources: 1. You can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

0.0.4 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 perhaps the smallest amount is the Cent (1). Perhaps it should be renamed Cents.

0.0.5 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.



0.1 Processes

Various Process objects can be **Receivers** or **Providers** or both, or compose these objects into groups called **Macros**. Most Process objects can be placed on MASON's schedule to be stepped when appropriate.

0.1.1 sim.engine.Steppable

An object which can be placed on MASON's schedule to be stepped in the future.

0.1.2 Named

A Steppable which also has a **name** (a String).

0.1.3 Resettable

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

0.1.4 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.

0.1.5 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.**

0.1.6 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.

0.1.7 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.

0.1.8 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.

0.1.9 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.

0.1.10 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.

0.1.11 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.

0.1.12 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.

0.1.13 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.**

0.2 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.

0.2.1 Filter

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

0.2.2 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.**

0.2.3 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.**

0.2.4 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.**

0.2.5 Probe

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

0.2.6 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.

0.2.7 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.

0.3 Process Utilities

0.3.1 Pool

A shared pool of a Resources which may be allocated or provided by Lock and Unlock objects. **By default, this class's step() method does nothing, so there's no need to schedule it.**

0.3.2 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.

0.3.3 Service

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

0.3.4 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.