Multi-method modeling

The three modeling methods, or paradigms, are essentially the three different viewpoints the modeler can take when mapping the real world system to its image in the world of models. The system dynamics paradigm suggests to abstract away from individual objects, think in terms of aggregates (stocks, flows), and the feedback loops. The discrete event modeling adopts a process-oriented approach: the dynamics of the system are represented as a sequence of operations performed over entities. In an agent based model the modeler describes the system from the point of view of individual objects that may interact with each other and with the environment.

Depending on the simulation project goals, the available data, and the nature of the system being modeled, different problems may call for different methods. Also, sometimes it is not clear at the beginning of the project which abstraction level and which method should be used. The modeler may start with, say, a highly abstract system dynamics model and switch later on to a more detailed discrete event model. Or, if the system is heterogeneous, the different components may be best described by using different methods. For example in the model of a supply chain that delivers goods to a consumer market the market may be described in system dynamics terms, the retailers, distributors, and producers may be modeled as agents, and the operations inside those supply chain components – as process flowcharts.

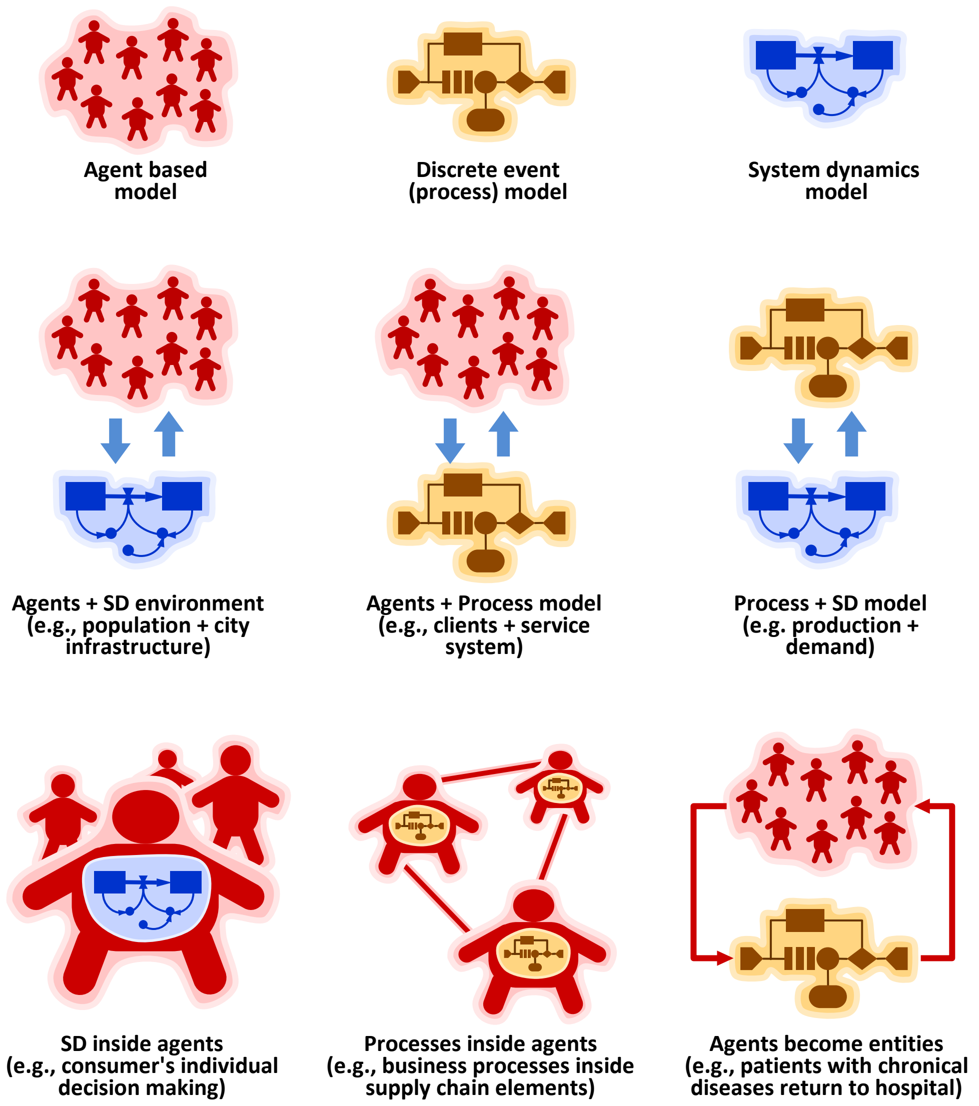
Frequently, the problem cannot completely conform to one modeling paradigm. Using a traditional single-method tool, the modeler inevitably either starts using workarounds (unnatural and cumbersome language constructs), or just leaves part of the problem outside the scope of the model (treats it as exogenous). If our goal is to capture business, economic, and social systems in their interaction, this becomes a serious limitation.

AnyLogic meets this challenge by supporting all three modeling methods on a single modern object-oriented platform. With AnyLogic, a modeler can choose from a wide range of abstraction levels, can efficiently vary them while working on the model, and can combine different methods in one model.

In this chapter we offer an overview of most used multi-method model architectures, discuss the technical aspects of linking different methods within one model, and consider examples of multi-method models.

Architectures

The number of possible multi-method model architectures is infinite, and many are used in practice. Popular examples are shown in the Figure. We will briefly discuss the problems where these architectures may be useful.


Agent based model
Discrete event (process) model
System dynamics model
Agents + SD environment (e.g., population + city infrastructure)
Agents + Process model (e.g., clients + service system)
Process + SD model (e.g. production + demand)
SD inside agents (e.g., consumer's individual decision making)
Processes inside agents (e.g., business processes inside supply chain elements)
Agents become entities (e.g., patients with chronical diseases return to hospital)

Popular multi-method model architectures

Agents in an SD environment. Think of a demographicmodel of a city. People work, go to school, own or rent homes, have families, and so on. Different neighborhoods have different levels of comfort, including infrastructure and ecology, cost of housing, and jobs. People may choose whether to stay or move to a different part of the city, or move out of the city altogether. People are modeled as agents. The dynamics of the city neighborhoods may be modeled in system dynamics way, for example, the home prices and the overall attractiveness of the neighborhood may depend on crowding, and so on. In such a model agents' decisions depend on the values of the system dynamics variables, and agents, in turn, affect other variables.

The same architecture is used to model the interaction of public policies (SD) with people (agents). Examples: a government effort to reduce the number of insurgents in the society; policies related to drug users or alcoholics.

Agents interacting with a process model. Think of a business where the service system is one of the essential components. It may be a call center, a set of offices, a Web server, or an IT infrastructure. As the client base grows, the system load increases. Clients who have different profiles and histories use the system in different ways, and their future behavior depends on the response. For example, low-quality service may lead to repeated requests, and, as a result, frustrated clients may stop being clients. The service system is naturally modeled in a discrete event style as a process flowchart where requests are the entities and operators, tellers, specialists, and servers are the resources. The clients who interact with the system are the agents who have individual usage patterns.

Note that in the previous example the agents can be created directly from the company CRM database and acquire the properties of the real clients. This also applies to the modeling of the company's HR dynamics. You can create an agent for every real employee of the company and place them in the SD environment that describes the company's integral characteristics (the first architecture type).

A process model linked to a system dynamics model. The SD aspect can be used to model the change in the external conditions for an established and ongoing process: demand variation, raw material pricing, skill level, productivity, and other properties of the people who are part of the process.

The same architecture may be used to model manufacturing processes where part of the process is best described by continuous time equations – for example, tanks and pipes, or a large number of small pieces that are better modeled as quantities rather than as individual entities. Typically, however, the rates (time derivatives of stocks) in such systems are piecewise constants, so simulation can be done analytically, without invoking numerical methods.

System dynamics inside agents. Think of a consumer market model where consumers are modeled individually as agents, and the dynamics of consumer decision making is modeled using the system dynamics approach. Stocks may represent the consumer perception of products, individual awareness, knowledge, experience, and so on. Communication between the consumers is modeled as discrete events of information exchange.

A larger-scale example is interaction of organizations (agents) whose internal dynamics are modeled as stock and flow diagrams.

Processes inside agents. This is widely used in supply chain modeling. Manufacturing and business processes, as well as the internal logistics of suppliers, producers, distributors and retailers are modeled using process flowcharts. Each element of the supply chain is at the same time an agent. Experience, memory, supplier choice, emerging network structures, orders and shipments are modeled at the agent level.

Agents temporarily act as entities in a process. Consider patients with chronic diseases who periodically need to receive treatment in a hospital (sometimes planned, sometimes because of acute phases). During treatment, the patients are modeled as entities in the process. After discharge from the hospital, they do not disappear from the model, but continue to exist as agents with their diseases continuing to progress until they are admitted to the hospital again. The event of admission and the type of treatment needed depend on the agent's condition. The treatment type and timeliness affect the future disease dynamics.

There are models where each entity is at the same time an agent exhibiting individual dynamics that continue while the entity is in the process, but are outside the process logic – for example, the sudden deterioration of a patient in a hospital.

The choice of model architecture and methods

AnyLogic, designed as a multi-method object-oriented tool, allows you to create model architectures of any type and complexity, including those previously mentioned. You can develop complex, simple, flat, hierarchical, replicated, static, or dynamically changing structures.

The choice of the model architecture depends on the problem you are solving. The model structure reflects the structure of the system being modeled – not literally, however, but as seen from the problem viewpoint. The choice of modeling method should be governed by the criterion of *naturalness*. Compact, minimalistic, clean, beautiful, easy to understand and explain – if the internal texture of your model is like that, then you chose the right method.

Technical aspect of combining modeling methods

In this section we will consider the techniques of linking different modeling methods in AnyLogic.

The very first thing you should know is that all model elements of all methods, be they SD variables, statechart states, entities, process blocks, and even animation shapes or business charts exist in the "same namespace": any element is accessible from any other element by name (and, sometimes, "path" – the prefix describing the location of the element).

The following examples are all taken from the real projects and purged of all unnecessary details. This set, of course, does not cover everything, but it does give a good overview of how you can build interfaces between different methods.

System dynamics -> discrete elements

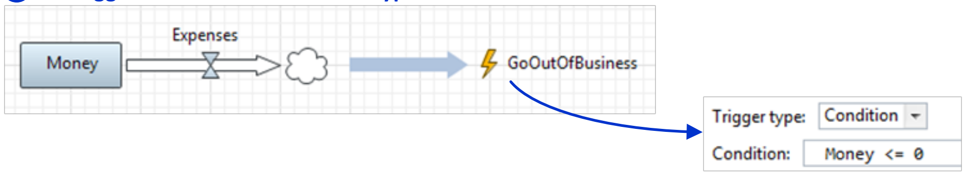
The system dynamics model is a set of continuously changing variables. All other elements in the model work in discrete time (all changes are associated with events). SD itself does not generate any events, so it cannot *actively* make an impact on agents, process flowcharts, or other discrete time constructs. The only way for the SD part of the model to affect a discrete element is to let that element watch on a condition over SD variables, or to use SD variables when making a decision. The Figure shows some possible constructs.

A and B. SD triggers an event or a statechart transition. Events (low-level constructs that allow scheduling a one-time or recurrent action) and statechart transitions are frequent elements of agent behavior. Among other types of triggers, both can be triggered by a condition – a Boolean expression.

If the model contains dynamic variables, all conditions of events and statechart transitions are evaluated *at each integration step*, which ensures that the event or transition will occur exactly when the (continuously changing) condition becomes true.

In the Figure A and B the discrete elements are waiting for the **Money** stock to fall below zero, and for the **Interest** to rise higher than a given threshold value. The event and the statechart can be located on the same level as the SD, or in a different active object.

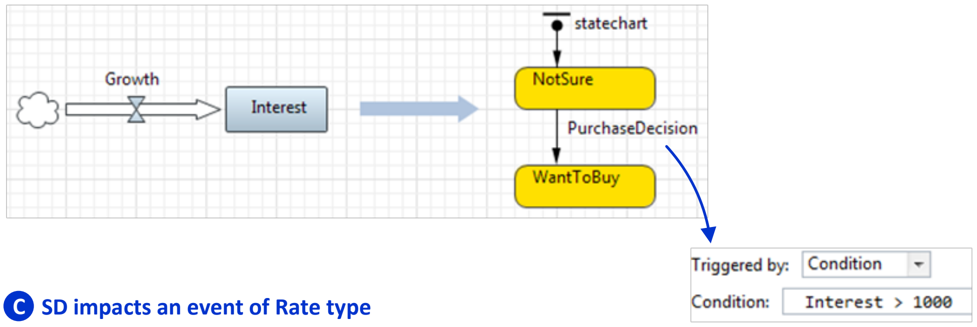
A SD triggers an event of Condition type



Trigger type: Condition

Condition: Money <= 0

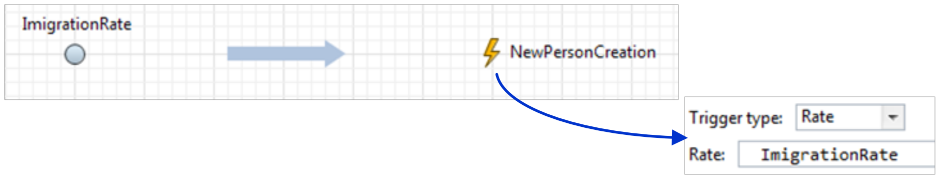
B SD triggers a statechart transition of Condition type



Triggered by: Condition

Condition: Interest > 1000

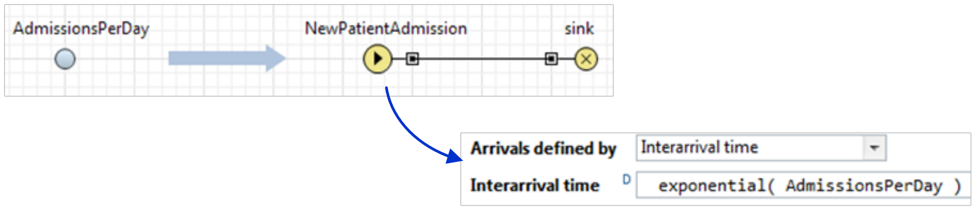
C SD impacts an event of Rate type



Trigger type: Rate

Rate: ImigrationRate

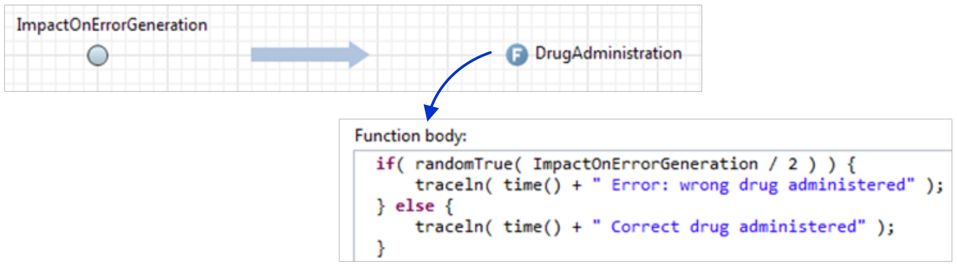
D SD controls the enity generation in a process flowchart



Arrivals defined by Interarrival time

Interarrival time exponential(AdmissionsPerDay)

E SD impacts a decision made when a function is called



Function body:

```
if( randomTrue( ImpactOnErrorGeneration / 2 ) ) {
    traceln( time() + " Error: wrong drug administered" );
} else {
    traceln( time() + " Correct drug administered" );
}
```

SD impacts discrete elements of the model

C and D. SD controls the rate of recurring discrete events. In the case of C there is an event with a rate trigger type (the time between subsequent occurrences is exponentially distributed). The event rate is set to the dynamic variable, and each subsequent occurrence is scheduled according to the current value of the variable.

In the case D the **Source** block **NewPatientAdmissions** generates new entities at the rate defined by the dynamic variable **AdmissionsPerDay**. The arrivals are defined in the form of interarrival time and not in the form of rate, because the rate is not re-evaluated during the simulation, whereas the interarrival time is re-evaluated after each new entity.

Note that if the value of the dynamic variable changes *in between* two subsequent event occurrences (or in between two entity arrivals), this will not be "noticed" immediately, but only at the next event occurrence (or next entity arrival).

E. The SD variable is used in a decision made in the DE part of the model. This example just shows that SD variables can be freely used in the actions and expressions inside the discrete elements of the model: conditions, functions, actions of events and transitions, **On enter**/**On exit** fields of process objects, and so on. Dynamic variables in that sense do not differ at all from plain (Java) variables.

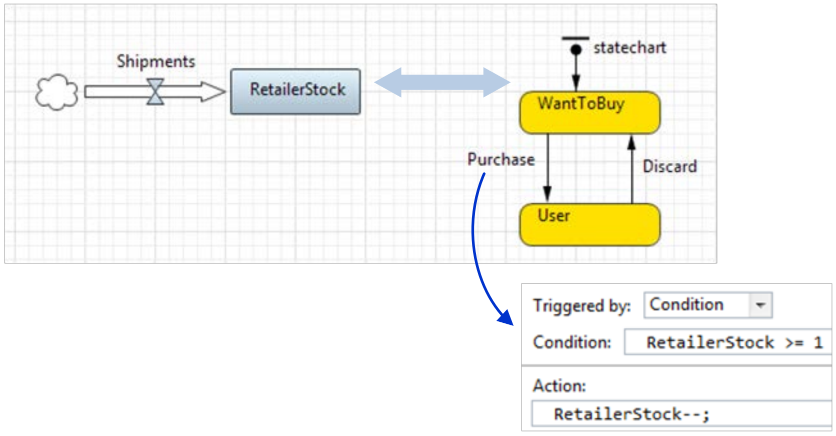
Discrete elements -> system dynamics

F. The SD stock triggers a statechart transition, which, in turn, modifies the stock value. Here, the interface between the SD and the statechart is implemented in the pair condition/action. In the state **WantToBuy**, the statechart tests if there are products in the retailer stock, and if there are, buys one and changes the state to **User**.

You can freely change the values of the system dynamic *stocks* from outside the system dynamics part of the model. This does not interfere with the differential equation solving: the integrator will just start with the new value. However, trying to change the value of a flow or auxiliary variable that has an equation associated with it, is not correct: the assigned value will be immediately overridden by the equation, so assignment will have no effect.

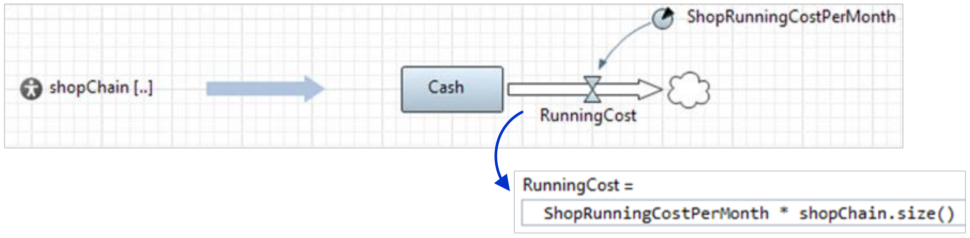
G. The size of the agent population is used in the SD equation. The equations in the system dynamics part of the model can reference not only the SD variables and functions, but also the arbitrary discrete elements in the model. The flow **RunningCost** out of the **Cash** stock depends on the current number of shops in the chain, and each shop is an agent. The function call **shopChain.size()** returns the number of agents in the population **shopChain**.

F The SD stock triggers a statechart transition, which, in turn, modifies the stock value



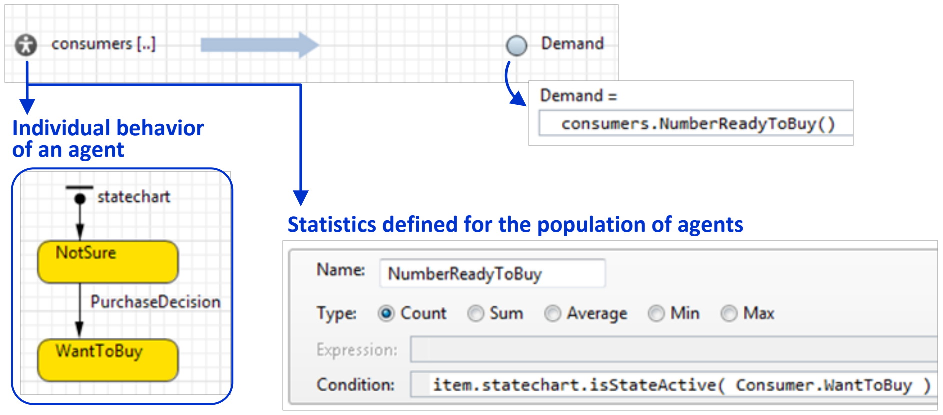
Triggered by: Condition
Condition: `RetailerStock >= 1`
Action:
```
RetailerStock--;
```

G Number of agents impacts an SD flow



RunningCost =
```
ShopRunningCostPerMonth * shopChain.size()
```

H Statistics on agent population impact an SD variable [may be inefficient!]


consumers [..]
Demand
Individual behavior of an agent
statechart
NotSure
PurchaseDecision
WantToBuy
Statistics defined for the population of agents

Demand =
```
consumers.NumberReadyToBuy()
```

Name: NumberReadyToBuy
Type: [✓ Count] Sum Average Min Max
Expression:
Condition: `item.statechart.isStateActive( Consumer.WantToBuy )`

Discrete model elements impact SD (part 1)

H and I. The statistics on agent population impacts an SD variable. Here, we are calculating the number of agents in a population that are in a particular state **WantToBuy**, and provide that value to the SD variable **Demand**.

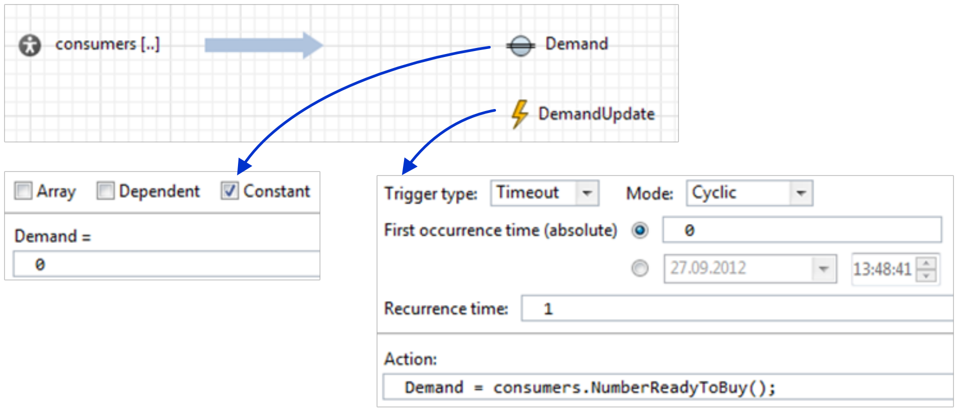
In the case H this is implemented in a straightforward way: the call to the **NumberReadyToBuy()** is typed directly into the equation field of **Demand**, and, therefore, statistics are re-evaluated at each integration step. If there are many agents in the model, this may be very time consuming and slow down the simulation. In case I the same functionality is implemented more efficiently: the variable **Demand** is declared as "constant" (no equation), and its value is periodically updated by the recurrent event **DemandUpdate**, whose frequency is significantly lower than the frequency of the numeric integrator.

J. The SD variable is controlled by the statechart state. Consider a model of a consumer who has a certain degree of interest in the product modeled by an "individual" SD stock **Interest** (located inside the consumer model). As the consumer is waiting for the product to become available, his interest decreases. This is captured by the equation of **LossDueToUnavailability** flow that contains the function call **statechart.isStateActive(Waiting)**. This function returns **true** if the statechart is in a given state, and **false** otherwise. In this model you also can implement a loop back from the SD to the statechart if you let the transition **InterestLost** occur when the value of the stock falls below a certain threshold.

K. A message received by an agent modifies the internal SD stock. Such agent can be an element of a supply chain where orders and shipments are modeled as messages between agents. In the case K, the incoming message is of the type **Integer** (same as **int**) and is treated as the amount of raw material shipped to this agent. The internal dynamics of the agent are modeled in SD way, and the stock **RawMaterialInventory** is incremented upon the message arrival (see the **On message received** code of the agent). In more complex models the message can include the supplier ID, the type of product, and so on.

L. The bi-directional flow between the SD stocks is implemented outside the SD. The two stocks model available and occupied houses in the two neighborhoods of NYC. Both stocks are arrays with the dimension {**Midtwon**, **Soho**}. People (agents) may move from Midtown to SoHo and vice versa. When leaving, they decrement the bucket of the **Occupied** stock and increment the bucket of the **Available** stock. Then they do the reverse operation on the pair of buckets belonging to their new location.

I Statistics on agent population impact an SD variable [efficient]



Demand =

```
0
```

Trigger type: Timeout Mode: Cyclic

First occurrence time (absolute) ● 0

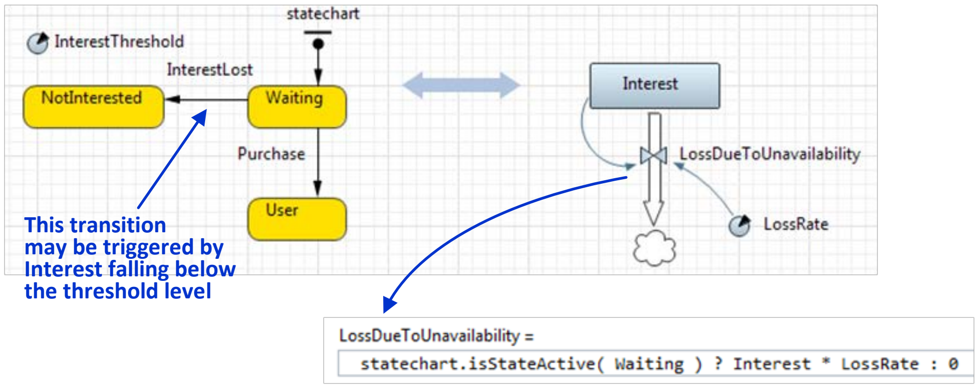
○ 27.09.2012 13:48:41

Recurrence time: 1

Action:

```
Demand = consumers.NumberReadyToBuy();
```

J The statechart state impacts the SD rate

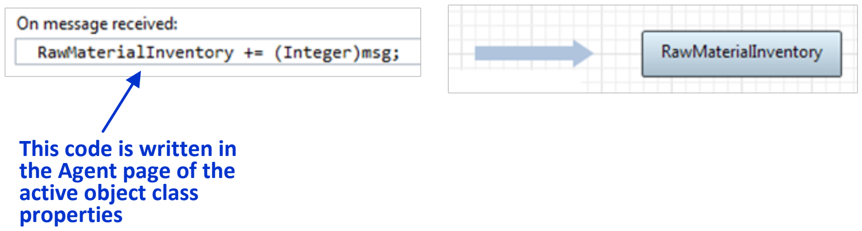


This transition may be triggered by Interest falling below the threshold level

LossDueToUnavailability =

```
statechart.isStateActive( Waiting ) ? Interest * LossRate : 0
```

K The inciming message modifies the SD stock



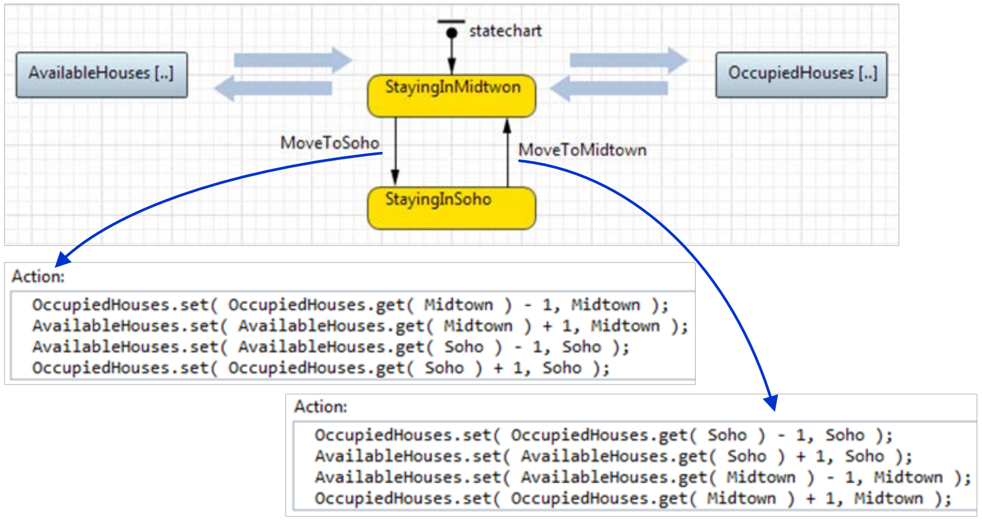
On message received:

```
RawMaterialInventory += (Integer)msg;
```

This code is written in the Agent page of the active object class properties

Discrete model elements impact SD (part 2)

L The Statechart controls flow between SD stocks



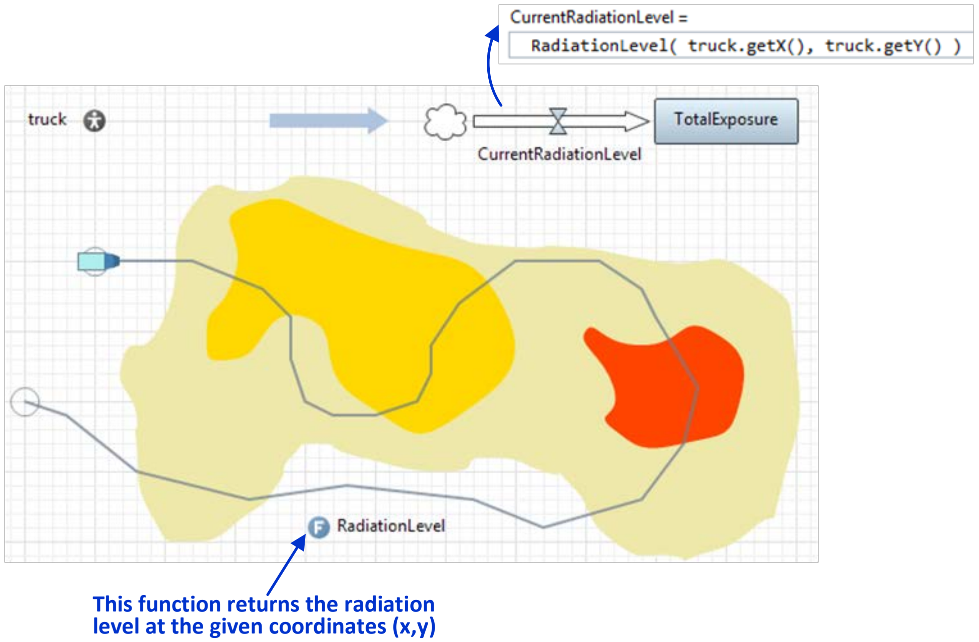
Action:

```
OccupiedHouses.set( OccupiedHouses.get( Midtown ) - 1, Midtown );
AvailableHouses.set( AvailableHouses.get( Midtown ) + 1, Midtown );
AvailableHouses.set( AvailableHouses.get( Soho ) - 1, Soho );
OccupiedHouses.set( OccupiedHouses.get( Soho ) + 1, Soho );
```

Action:

```
OccupiedHouses.set( OccupiedHouses.get( Soho ) - 1, Soho );
AvailableHouses.set( AvailableHouses.get( Soho ) + 1, Soho );
AvailableHouses.set( AvailableHouses.get( Midtown ) - 1, Midtown );
OccupiedHouses.set( OccupiedHouses.get( Midtown ) + 1, Midtown );
```

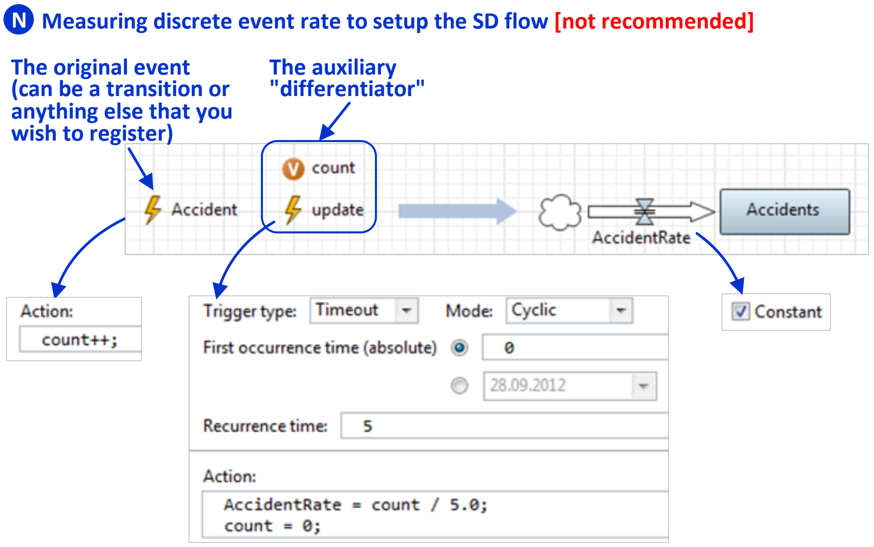
M The SD stock accumulates position properties of a moving agent


CurrentRadiationLevel =

```
RadiationLevel( truck.getX(), truck.getY() )
```

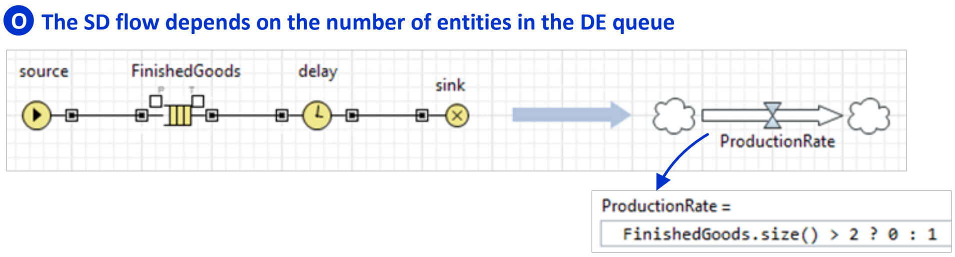
truck
TotalExposure
CurrentRadiationLevel
RadiationLevel
This function returns the radiation level at the given coordinates (x,y)

Discrete model elements impact SD (part 3)


N Measuring discrete event rate to setup the SD flow [not recommended]
The original event (can be a transition or anything else that you wish to register)
The auxiliary "differentiator"
count
Accident
update
AccidentRate
Accidents
Constant

```
Action:
count++;
```

```
Trigger type: Timeout    Mode: Cyclic
First occurrence time (absolute) ● 0
                                 ○ 28.09.2012
Recurrence time: 5
Action:
  AccidentRate = count / 5.0;
  count = 0;
```


O The SD flow depends on the number of entities in the DE queue
source
FinishedGoods
delay
sink
ProductionRate

```
ProductionRate =
  FinishedGoods.size() > 2 ? 0 : 1
```

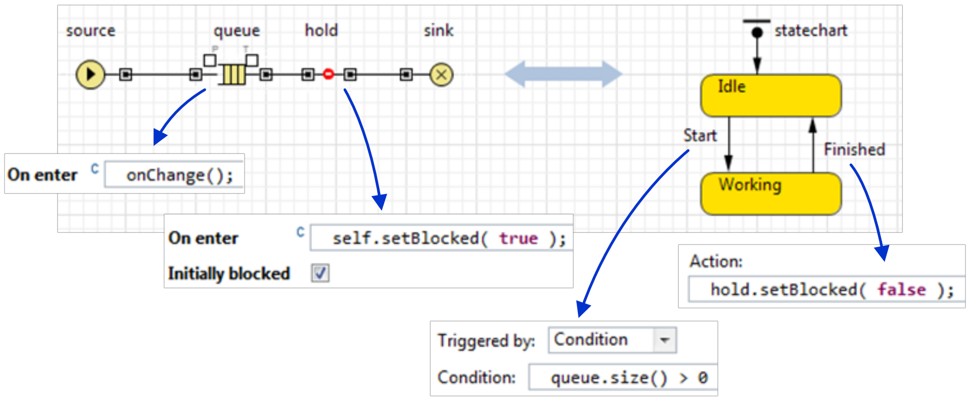
Discrete model elements impact SD (part 4)

M. The SD stock accumulates the "history" of the agent motion. This is an interesting example of SD-AB cooperation. The value of the stock **TotalExposure** is constantly updated as the mobile agent moves through the area contaminated by radiation. The value of the incoming flow **CurrentRadiationLevel** is set to the radiation level at the coordinates of the truck agent. As the truck moves or stays, the stock receives and accumulates a dose of radiation per time unit. Again, in the SD equation, we are referencing the agent and calling its function.

N. Measuring discrete event rate and feeding it to the SD flow. Suppose you want to create an SD reflection of the discrete event flow. The discrete events may model things like purchases, arrivals, accidents, decisions, and so on. The technique shown in the Figure M can be used, but it is not recommended. The technique works as follows: the auxiliary recurring event **update** counts the number of **Accident** occurrences within a fixed interval, and sets up the **AccidentRate** to the number divided by the interval duration. This is, in fact, a rough derivative calculation. This implementation requires two auxiliary elements and the correct choice of the update interval.

Whenever possible, we recommend to directly update the stock, and not the flow. In this case, the original event **Accident** could increment the stock **Accidents** on each occurrence. This will be a 100% accurate solution without any unnatural constructs.

P Agent based server interactes with the DE process



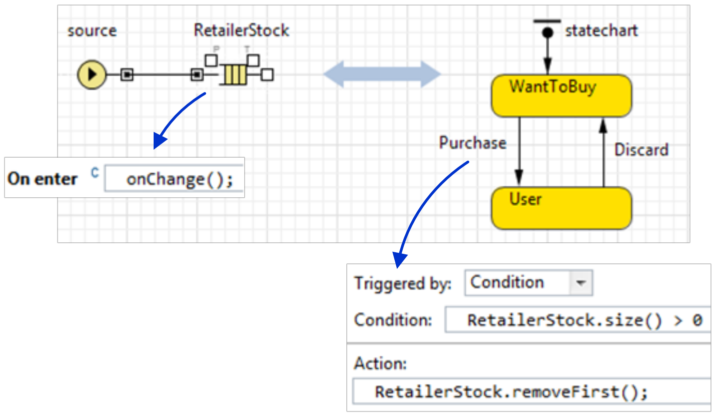
On enter: `onChange();`

On enter: `self.setBlocked( true );`
Initially blocked: ✓

Action: `hold.setBlocked( false );`

Triggered by: Condition
Condition: `queue.size() > 0`

Q The agent removes entities from the DE queue



On enter: `onChange();`

Triggered by: Condition
Condition: `RetailerStock.size() > 0`
Action: `RetailerStock.removeFirst();`

Agent based parts of the model interact with discrete event parts

O. Referencing DE objects in the SD formula. In this example, the flow **ProductionRate** switches between 0 and 1, depending on whether the finished products inventory (the number of entities in the queue **FinishedGoods** returned by the function **size()**) is greater than 2 or not. Again, one can close the loop by letting the SD part control the production process.

Agent based <-> discrete event

P. A server in the DE process model is implemented as an agent. Imagine complex equipment, such as a robot or a system of bridge cranes. The behavior of such objects is often best modeled "in agent based terms" by using events and statecharts. If the equipment is a part of the manufacturing process being modeled, you need to build an interface between the process and the agent representing the equipment.

In this example, the statechart is a simplified equipment model. When the statechart comes to the state **Idle**, it checks if there are entities in the queue. If yes, it proceeds to the **Working** state and, when the work is finished, unblocks the **hold** object, letting the entity exit the queue. The **hold** object is set up to block itself again after the entity passes through.

The next entity will arrive when the equipment is in the **Idle** state. To notify the statechart, we call the function **onChange()** upon each entity arrival (see the **On enter** action of the queue).

Unlike in the models with continuously changing SD elements, in the models built of purely discrete elements events and transitions triggered by a condition, *do not monitor the condition continuously*. The event's condition is evaluated when the event is reset. The transition's condition is evaluated when the statechart comes into the transition's source state. And then the conditions are re-evaluated when something happens to the active object where they are located, or when its **onChange()** function is called.

Q. The agent removes entities from the DE queue. Here, the supply chain is modeled using discrete event constructs; in particular, its end element, the retailer stock, is a **Queue** object. The consumers are outside the discrete event part, and they are modeled as agents. Whenever a consumer comes to the state **WantToBuy**, it checks the **RetailerStock** and, if it is not empty, removes one product unit. Again, as this is a purely discrete model, we need to ensure that the consumers who are waiting for the product are notified about its arrival – that's why the code **onChange()** is placed in the **On enter** action of the **RetailerStock** queue.

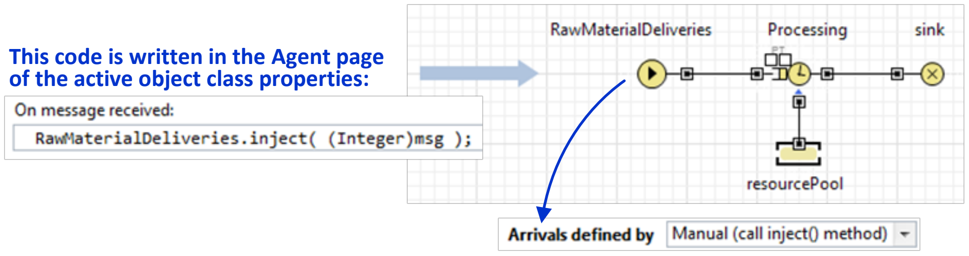
In this simplified version, there is only one consumer whose statechart is located on the same canvas as the supply chain flowchart. In the full version there would be multiple agents-consumers and, instead of calling just **onChange()**, the retailer stock would notify *every consumer* in a loop, see the example in the next subsection.

R. The incoming message injects entities into the DE process. In this example and next process flowcharts are put inside agents. This architecture can be used in supply chain models. In the case Figure R the message received by the agent has a meaning of raw material shipment that starts the manufacturing process. In the **On message received** action the agent injects entities into the flowchart by calling the **Source** object function **inject()**. The source object **RawMaterialDeliveries** works in the "manual" mode when it does not generate entities unless **inject()** is called.

S. A process inside the agent initiates an outgoing message. Similarly, in the Figure S, the manufacturing process ends with the **Sink** object **ShipToConsumer.** When an entity representing the finished product unit gets there, before it disappears from the process, the **Sink** sends out the message using the agent interface function **send().**

In this simplified model, the message is just an integer number 1, but in more realistic cases, you can send out the entity itself, or batches of multiple entities. On the receiving end, the entities can be directly injected into the process. The pair **Exit-Enter** should be used then, instead of the pair **Sink-Source.**

R The incoming message injects entities into the DE process

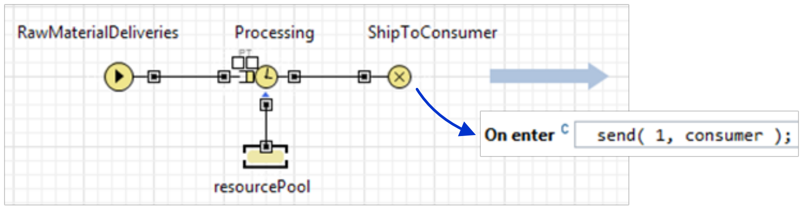

This code is written in the Agent page of the active object class properties:
RawMaterialDeliveries
Processing
sink
resourcePool

On message received:

```
RawMaterialDeliveries.inject( (Integer)msg );
```

Arrivals defined by: Manual (call inject() method)

S A process inside the agent initiates an outgoing message



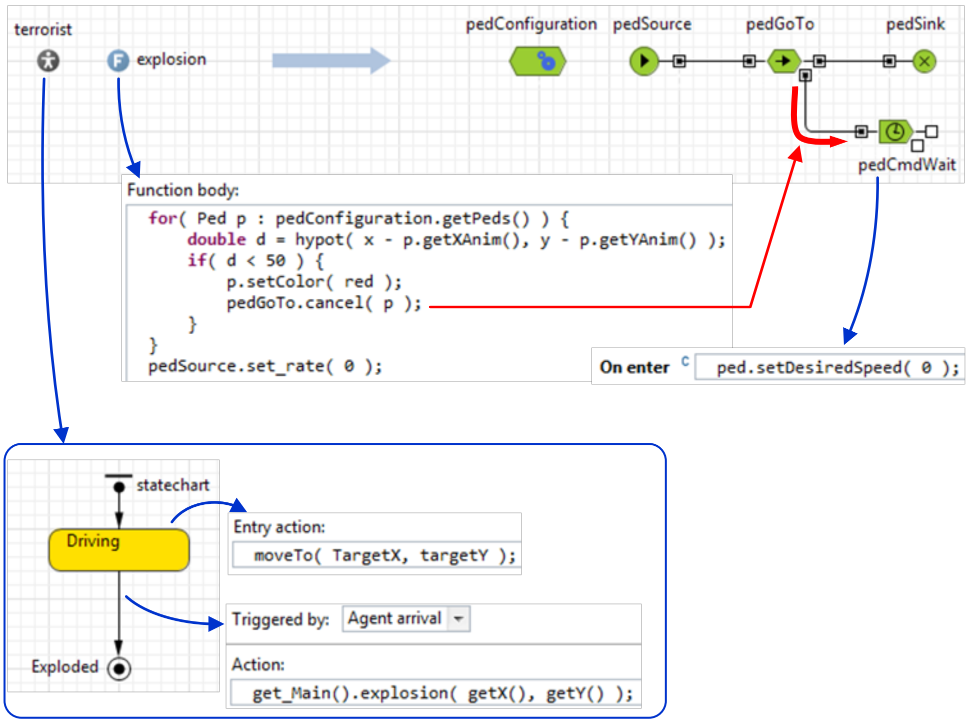
On enter:

```
send( 1, consumer );
```

Incoming and outgoing messages interact with the DE process inside agent

T. The agent shares a 2D space with pedestrians and generates an event that affects them. Pedestrian flow is modeled using **AnyLogic Pedestrian Library.** The agent represents a terrorist who drives a car into a crowd and explodes. Everyone dies who happened to be closer than 50 meters to the car.

T The agent generates event in 2D space that affects pedestrians



Function body:

```
for( Ped p : pedConfiguration.getPeds() ) {
    double d = hypot( x - p.getXAnim(), y - p.getYAnim() );
    if( d < 50 ) {
        p.setColor( red );
        pedGoTo.cancel( p );
    }
}
pedSource.set_rate( 0 );
```

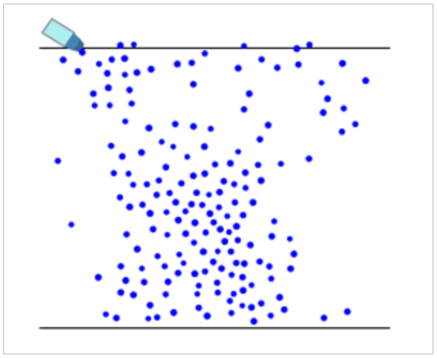
On enter: `ped.setDesiredSpeed( 0 );`

Entry action: `moveTo( TargetX, targetY );`

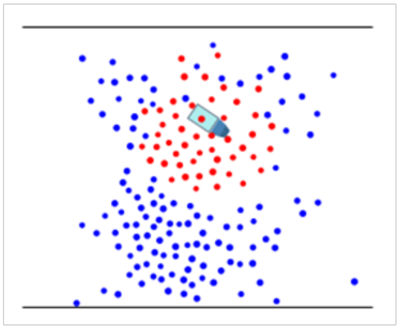
Triggered by: Agent arrival

Action: `get_Main().explosion( getX(), getY() );`

Terrorist car is approaching



Terrorist car exploded at the target point

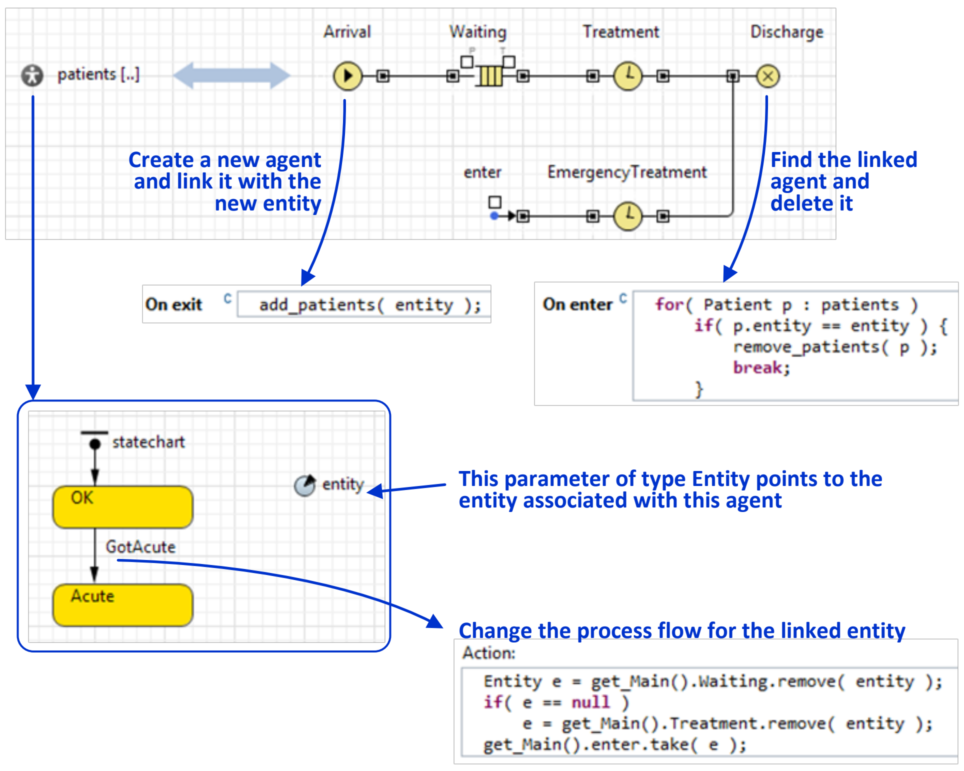


The agent generates event in 2D space that affects pedestrians

The interaction between the agent and the pedestrians is implemented in the function **explosion()**. The function accepts the arguments **x** and **y**, – the coordinates of the explosion. In the **for** loop, we iterate through all pedestrians (the collection returned by the function **getPeds()** of the **pedConfiguration** object). The pedestrian coordinates can be obtained via **getXAnim()** and **getYAnim()**. If the pedestrian is in the death range, we extract him from the regular flow by calling the function **cancel()** of **pedGoTo**. The dead pedestrians exit **pedGoTo** via the "emergency" port and enter the **pedCmdWait** object where their speed is set to 0.

U. An entity in the process is at the same time an agent. Although this architecture is not present in the Figure, it is often used to model process flows that can be changed or interrupted by events generated by entities' individual dynamics. Consider a patient who is in the middle of being treated in a hospital, and whose condition suddenly changes from normal to acute. This patient should be removed from the regular process and inserted into a special emergency treatment process.

U An Entity in DE process is at the same time an agent


patients [..]
Arrival
Waiting
Treatment
Discharge
Create a new agent and link it with the new entity
enter
EmergencyTreatment
Find the linked agent and delete it
statechart
OK
GotAcute
Acute
entity
This parameter of type Entity points to the entity associated with this agent
Change the process flow for the linked entity

On exit
```
add_patients( entity );
```

On enter
```
for( Patient p : patients )
    if( p.entity == entity ) {
        remove_patients( p );
        break;
    }
```

Action:
```
Entity e = get_Main().Waiting.remove( entity );
if( e == null )
    e = get_Main().Treatment.remove( entity );
get_Main().enter.take( e );
```

Each entity in the process has an agent associated with it

In the model shown in the Figure, a population of agents (**patients**) is located on the same canvas as the process flowchart. The population is initially empty. As a new patient-entity is generated by the **Arrivals** source object, a new patient-agent is created and gets linked with the entity: the agent's parameter **entity** points to the newly created patient-entity. When the patient-entity is discharged, the **Discharge** sink object searches for the patient-agent associated with the entity and deletes it from the model.

Each entity in the process, therefore, has an agent linked to it, and vice versa. The agent has individual dynamics: at any time, it can change its state from **OK** to **Acute**. When that transition happens, the agent looks for the linked entity, which can be either in the **Waiting**, or in the **Treatment** stage of the process. The agent removes the entity from the regular process and puts it into the **enter** object, so the patient-entity continues in the emergency treatment process.

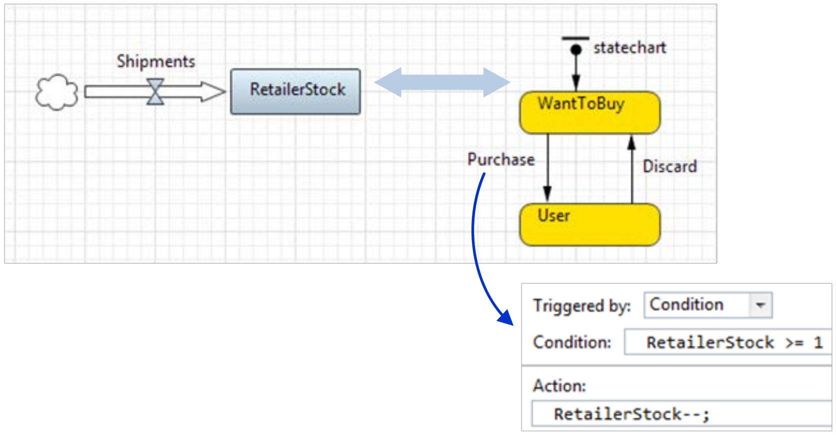
The interaction between the AB and DE parts can be further extended in this model. For example, the regular treatment performed on time can prevent the patient from entering the **Acute** state. On the technical side, one can implement the direct reverse link from the entity to the agent by adding the entity's field **agent** pointing to the associated agent. The procedure of agent deletion will then get a lot simpler.

Referencing model elements located in different active objects

For simplicity, in the previous examples, the statecharts, events, SD variables, and process flowcharts are often located on the same canvas (in the same active object). Inreal models, elements belonging to the different methods are often located in different active objects/agents and on different hierarchy levels. The only change in the model code would be the way the model elements reference one another: prefixes might be needed before their names. The Figure in the chapter Java for AnyLogic users gives a general idea about the prefixes. Here, we will refactor example F to illustrate this.

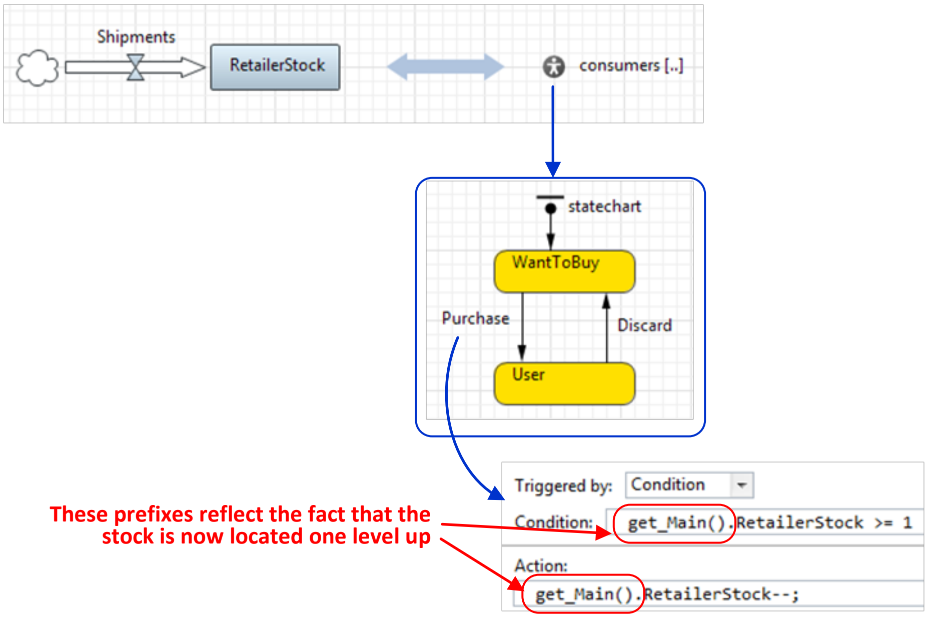
The top case in the Figure is the original example, where the statechart modeling the consumer behavior is located on the same level as the system dynamics stock **RetailerStock** in the same **Main** active object. We want to have multiple consumers in the model; therefore we will place the statechart in the agent **Consumer** and place a population of **consumers** in the **Main** object, replacing the old statechart, see the bottom case in the Figure. Because the **RetailerStock** is now located one level up from the statechart, we need to refactor the code in the **Purchase** transition to reflect this. The prefix "**get_Main().**" brings us to the direct container of a consumer – to the **Main** active object.

F An SD stock triggers a statechart transition, which, in turn, modifies the stock value



Triggered by: Condition
Condition: `RetailerStock >= 1`
Action:
```
RetailerStock--;
```

F [Refactored] Same, but now the statechart is inside the agent


Shipments
RetailerStock
consumers [..]
statechart
WantToBuy
Purchase
Discard
User
These prefixes reflect the fact that the stock is now located one level up

Triggered by: Condition
Condition: `get_Main().RetailerStock >= 1`
Action:
```
get_Main().RetailerStock--;
```

Discrete model elements impact SD (part 1)

The function **get_Main()** was automatically generated by AnyLogic into the **Consumer** active object, when it noticed you have embedded the **Consumer** into **Main**. If the **Consumer** had been embedded into, say, the **Market** object, the function **get_Market()** would have been generated instead. Should you have two populations of consumers, one in **MarketA**, and another in **MarketB**, two functions will be generated, and the function **get_MarketB()** called in the consumer embedded into **MarketA** will return **null**.

The simulation performance of multi-method models

The AnyLogic simulation engine is specifically designed to efficiently execute a mixture of continuous dynamics with discrete events. It is also set up to support large numbers of parallel activities exhibited by agent based models. Two or three methods working together in the same model do not slow down the simulation or increase the memory footprint of the model. There are, however, some things you should keep in mind when designing multi-method models.

SD is typically simulated more slowly than other methods. The frequency of micro-steps of the numeric solver is typically higher than the frequency of the discrete events in the model. Moreover, the frequency of the numeric steps is constant, whereas the intervals between discrete events are irregular. In the case of a purely discrete model, the engine would just jump to the next event, no matter how far it is on the time axis. The numeric solver settings can be adjusted on the **Advanced** page of the experiment properties.

Each SD formula is evaluated multiple times at each numeric step. Therefore, the time complexity of formula evaluation affects a lot of the overall simulation performance. You should avoid including complex calculations (for example, loops traversing populations of agents or collections of entities) directly into the formulas. Consider the cases H and I in the example set previously considered.

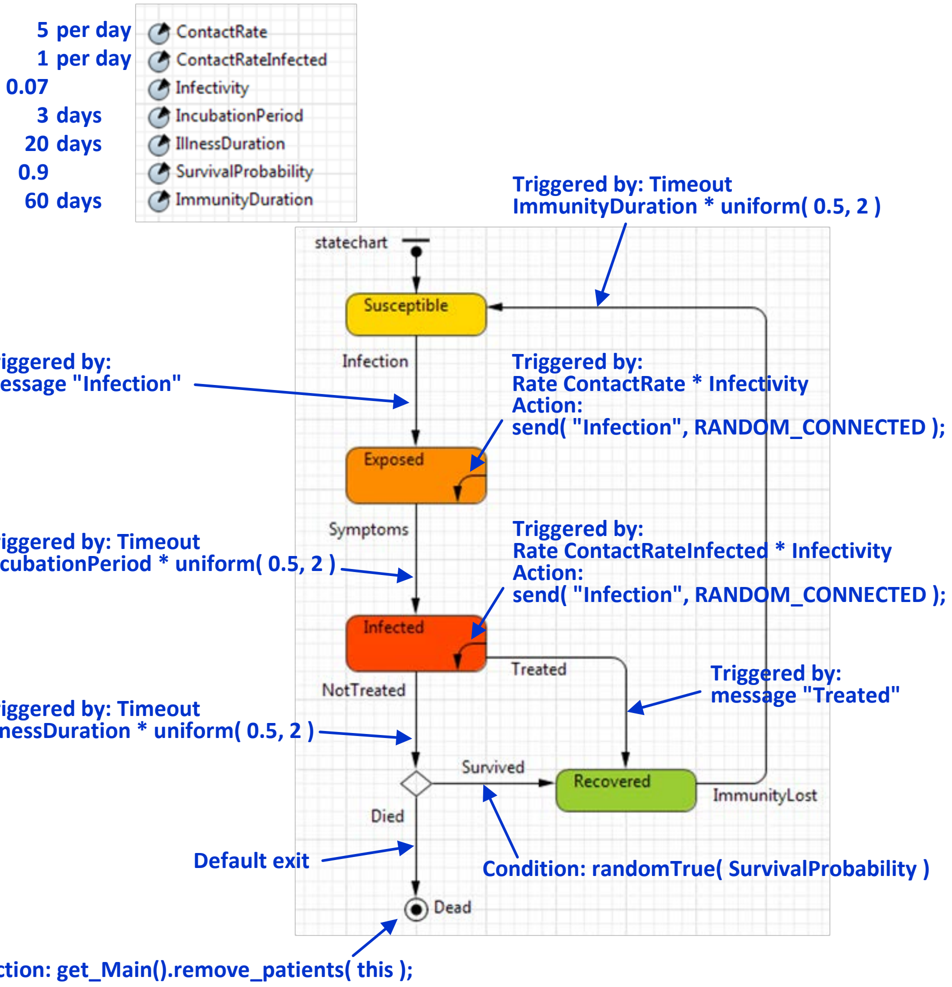
If SD is present in the model, the conditions of events and transitions are also evaluated at each numeric step. This is done to ensure that the events and transition will be triggered exactly when their conditions turn true. The computational complexity of those conditions should also be kept low.

If you embed an SD stock and flow diagram into an agent, at runtime there will be as many independent copies of that diagram as there are agents. If, for example, the diagram contains 100 dynamic variables, in a population of 1,000 agents, there will be a system of 100,000 dynamic variables submitted to the numeric solver, and you should expect the corresponding impact on the simulation performance.

Examples

Example: Epidemic and clinic

We will create a simple agent based epidemic model and link it to a simple discrete event clinic model. When a patient discovers symptoms, he will ask for treatment in the clinic, which has limited capacity. We will explore how the capacity of the clinic affects the disease dynamics. This model was suggested in 2012 by Scott Hebert, a consultant at AnyLogic North America.



5 per day ContactRate
1 per day ContactRateInfected
0.07 Infectivity
3 days IncubationPeriod
20 days IllnessDuration
0.9 SurvivalProbability
60 days ImmunityDuration
statechart
Susceptible
Infection
Exposed
Symptoms
Infected
Treated
NotTreated
Survived
Recovered
ImmunityLost
Died
Dead
Triggered by: Timeout ImmunityDuration * uniform(0.5, 2)
Triggered by: message "Infection"
Triggered by: Rate ContactRate * Infectivity Action: send("Infection", RANDOM_CONNECTED);
Triggered by: Timeout IncubationPeriod * uniform(0.5, 2)
Triggered by: Rate ContactRateInfected * Infectivity Action: send("Infection", RANDOM_CONNECTED);
Triggered by: message "Treated"
Triggered by: Timeout IllnessDuration * uniform(0.5, 2)
Default exit
Condition: randomTrue(SurvivalProbability)
Action: get_Main().remove_patients(this);

The patient behavior statechart

Create the agent population:

1. Create a new model.
2. Drag the **Agent Population** from the **General** palette onto the graphical editor of the **Main** object.
   In the first page of the wizard, change the **Agent class name** to **Patient**, and the **Population name** to **patients.** Set the initial number of agents to 2000.

In the next page, of the wizard set the space **Width** to 650 and **Height** to 200.Set **Network** type to **Distance based** with range 30. Click **Finish.**

3. Run the model. The agents are randomly distributed in the rectangular space and, so far, no activity is going on.

We have created a population of 2,000 agents. There is a network in the population: if the distance between two agents is less than 30, they are linked. The network and layout settings can be changed in the settings of the **environment** object. The next step is to define the behavior of our patient. We will use a statechart for that.

The statechart is similar to the classical SEIR statechart. The patient is initially in the **Susceptible** state, where he can be infected. Disease transmission is modeled by the message **"Infection"** sent from one patient to another. Having received such a message, the patient transitions to the state **Exposed**, where he is already infectious, but does not have symptoms. After a random incubation period, the patient discovers symptoms and proceeds to the **Infected** state. We distinguish between the **Exposed** and **Infected** states because the contact behavior of the patient is different before and after the patient discovers symptoms: the contact rate in the **Infected** state is 1 per day, as opposed to 5 in the **Exposed** state. The internal transitions in both states model contacts. We model only those contacts that result in disease transmission; therefore, we multiply the base contact rate by **Infectivity**, which, in our case, is 7%.

There are two possible exits from the Infected state. The patient can be treated in a clinic (and then, he is guaranteed to recover), or the illness may progress naturally without intervention. In the latter case, the patient can still recover with a high probability (90%), or die. If the patient dies, it deletes himself from the model, see the **Action** of the **Dead** state. The completion of treatment is modeled by the message **"Treated"** sent to the agent. So far, this message is never received, because we have not created the clinic model yet.

The recovered patient acquired a temporary immunity to the disease. We reflect this in the model by having the state **Recovered**, where the patient does not react to the message **"Infection"** that may possibly arrive. At the end of the immunity period the transition **ImmunityLost** takes the patient back to the **Susceptible** state.

Note that as long as we have defined the parameters *inside the agent*, we can make their values different for different agents. In our model, however, for simplicity, they are the same throughout the whole population.

▶ **Define the patient behavior:**

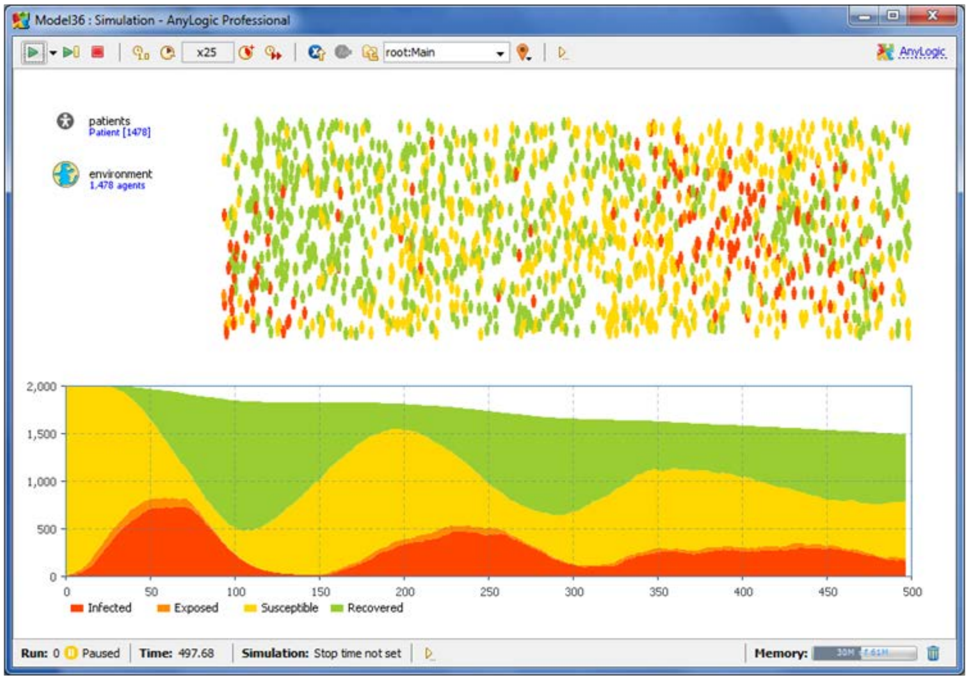
4. Open the editor of the **Patient** object.

5. Add seven parameters with the default values shown at the top left of the Figure.
6. Draw the statechart as shown in the Figure.
7. In the **Entry action** filed of each color states, type the code that changes the animation of the patient animation into the state color, for example, in the **Entry action** of the state **Exposed** type: **person.setFillColor(darkOrange);** (this is not shown in the Figure).
8. Open the properties of the **Main** object and type the following code in the **Startup code** field:

```
for( int i=0; i<5; i++ )
  patients.random().receive( "Infection" );
```

   This will infect five randomly chosen people in the beginning of the simulation.
9. Run the model for a while.

The contagious disease spreads around the initially infected agents (remember that our network of contacts is based on distance). The epidemic does not end after the first wave, because the immunity period is not long enough. We will now add a chart to view the type of the system dynamics.

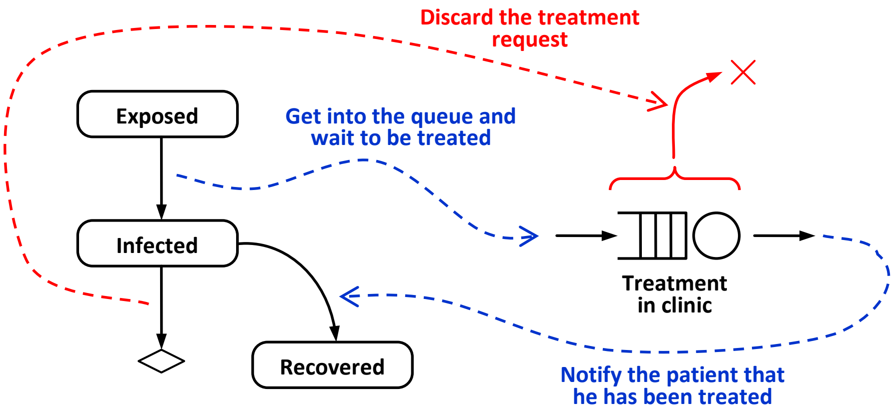


Oscillation in the epidemic model

▶ **Collect statistics and display graphs:**

10. Open the editor of **Main**, select the **patients** population and open the **Statistics** page of its properties. Add statistics with the name **NSusceptible** of type **count** and with the condition **item.statechart.isStateActive(item.Susceptible)**. This will generate the function **NSusceptible()** of the population that will count and return the number of patients who are in the **Susceptible** state.
11. Add three other statistics for the states **Exposed**, **Infected**, and **Recovered**, respectively.
12. Drag the **Time stack chart** from the **Analysis** palette onto the **Main** object, and place it below the area occupied by the agents. Set the **Time window** of the chart to 500, and **Display up to** 500 latest samples.
13. Add four data items to the chart corresponding to the four statistics functions you have created. The first item, for example, has the title Infected, **orangeRed** color, and value **patients.NInfected().**
14. Select the **Simulation** experiment in the **Projects** tree, and open the **Presentation** page of its properties. Set **Execution mode** to **Real time with scale** 25. This will make things happen faster.
15. Run the model. Now you can see oscillation and the gradual decrease of the total population due to deaths, see the Figure.

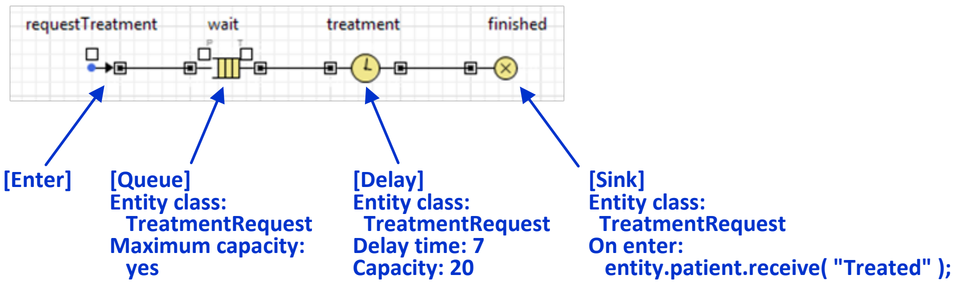
The next step is to add the clinic, and let the patients be treated there. Our clinic will be modeled via a very simple discrete event model: **Queue** for the patients waiting to be treated and **Delay** modeling the actual treatment. However, unlike in pure discrete event models, the entities in this process will not be generated by the **Source** object, but injected by the agents. The communication scheme between the patients-agents and the clinic process is shown in the Figure.


Discard the treatment request
Exposed
Get into the queue and wait to be treated
Infected
Treatment in clinic
Recovered
Notify the patient that he has been treated

Interface between the agent based and the discrete event parts of the model

Once the patient discovers symptoms, he will create an entity – let's call it "treatment request" – and inject the entity into the clinic process. Once the treatment is completed, the entity will notify the patient by sending him a message "Treated" that will cause the patient to transition to the **Recovered** state. If, however, the patent is cured or dies before the treatment is completed, he will discard his treatment request by removing it from whatever stage in the process it is. On the technical side, we need:

- The entity that will carry the reference to the patient
- The ability to remove the entity originated by a particular patient from the process.

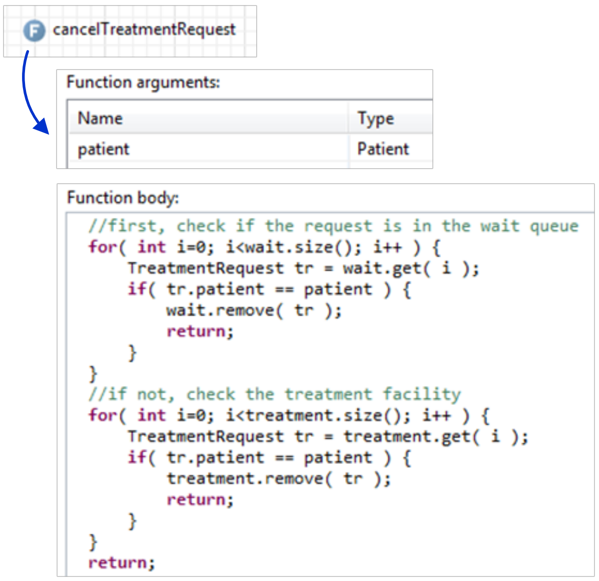


[Enter]

[Queue]
Entity class:
TreatmentRequest
Maximum capacity:
yes

[Delay]
Entity class:
TreatmentRequest
Delay time: 7
Capacity: 20

[Sink]
Entity class:
TreatmentRequest
On enter:
entity.patient.receive("Treated");



Function arguments:

Name	Type
patient	Patient

Function body:

```
//first, check if the request is in the wait queue
for( int i=0; i<wait.size(); i++ ) {
    TreatmentRequest tr = wait.get( i );
    if( tr.patient == patient ) {
        wait.remove( tr );
        return;
    }
}
//if not, check the treatment facility
for( int i=0; i<treatment.size(); i++ ) {
    TreatmentRequest tr = treatment.get( i );
    if( tr.patient == patient ) {
        treatment.remove( tr );
        return;
    }
}
return;
```

The discrete event part – the model of clinic

Create the new entity type TreatmentRequest:

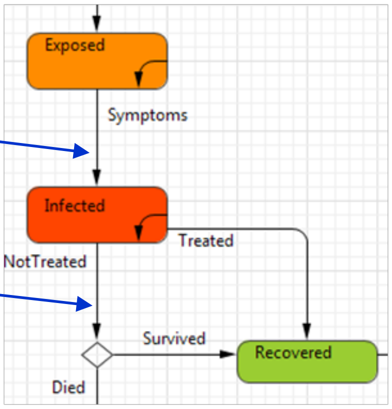
16. Right-click the model item in the **Projects** tree, and select **New | Java class** from the context menu. In the wizard, name the class **TreatmentRequest** and set its **Superclass** to **Entity**. Click **Next.**
17. On the next wizard page add, one field with the name **patient** and type **Patient**. Click **Finish**, and close the Java editor that opens – you will not need it.
18. Open the editor of **Main** and put together the process flowchart as shown in the Figure. Use objects from the **Enterprise library** palette.
19. Remaining in **Main**, create the function **cancelTreatmentRequest()** with the parameters and the body code as shown in the Figure.

We created a custom entity class **TreatmentRequest** that has a field **patient** – this will be the reference to the patient who originated the treatment request. In the process flowchart, we identified that entities passing through the **wait**, **treatment**, and **finished** objects are not of the generic **Entity** class, but of its subclass **TreatmentRequest.** This is necessary because we plan to use the **patient** field of those entities. For example, when the treatment is finished, the finished object sends a message "Treated" to the patient referenced by the entity before disposing of the entity.

We also specified that the queue has infinite capacity, that the treatment takes exactly seven days, and that there are only 20 beds in the clinic, so only 20 patients can be treated simultaneously.

Action:
get_Main().requestTreatment.
take(new TreatmentRequest(this));

Action:
get_Main().cancelTreatmentRequest(this);



The patient behavior modified to communicate with the clinic process

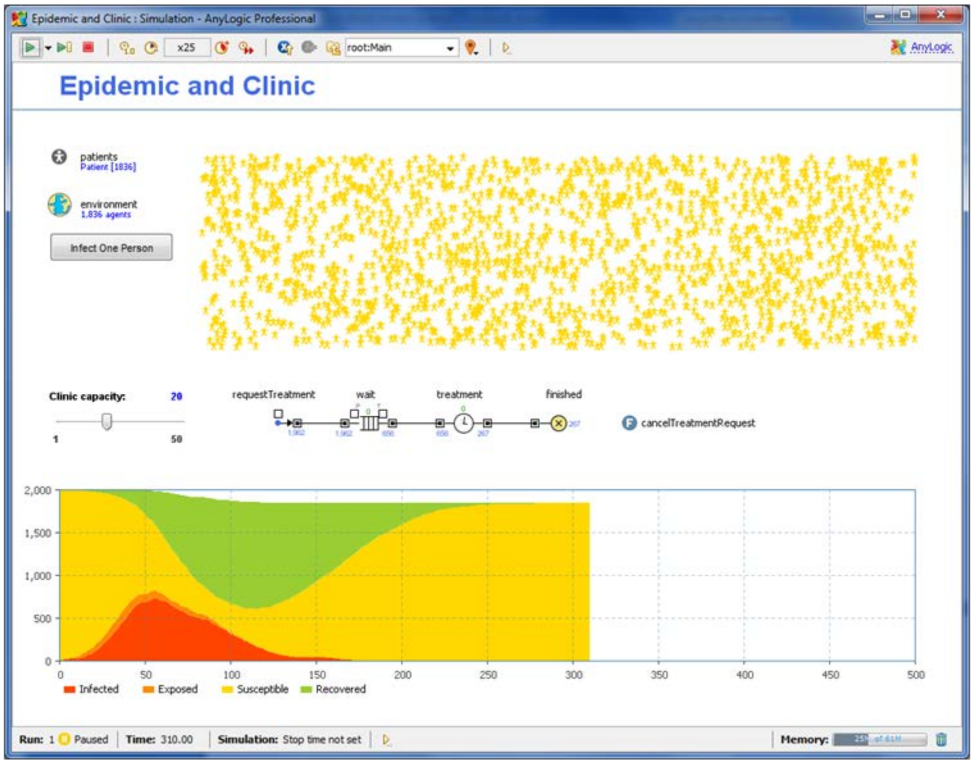
Finally, we prepared the function **cancelTreatmentRequest()** that will be called by patients who got well on their own or died before getting chance to be treated. That function uses the API of the **Queue** and **Delay** objects to search for a particular entity in them and remove it.

The remaining task is to modify the behavior of **Patient** to link it to the model of clinic. Remember that since the clinic process is located one level above the patient's statechart, in the **Main** object, the clinic objects and functions should be preceded by the prefix **get_Main()**. The Java word "**this**" references the object to which the code belongs, in this case, the patient.

Incorporate treatment in the clinic into the patient behavior:

20. Add the actions to the statechart transitions **Symptoms** and **NotTreated** as shown in the Figure.
21. The model is complete. Run the model.

Now, the model shows a different dynamic, or, to be more precise, a different range of dynamics. The oscillations are still possible, but a possibility also exists that the epidemic will end after the first wave, as shown in the Figure. You may experiment with different clinic capacities to figure out the number of beds needed in order to treat everybody on time and prevent the further waves of the epidemic.



The patient behavior modified to communicate with the clinic process

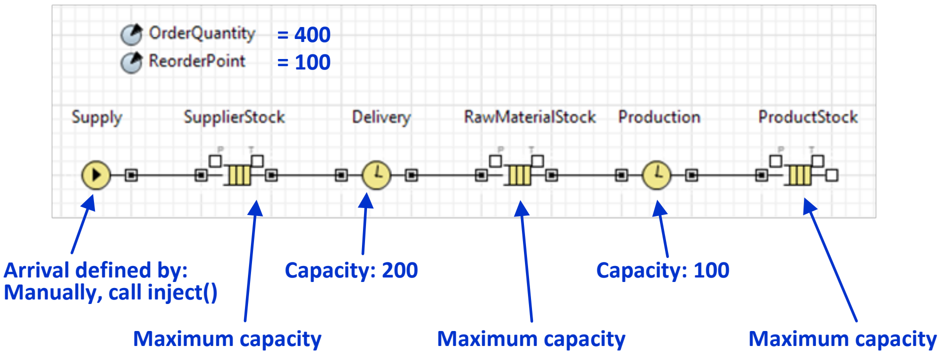
Example: Consumer market and supply chain

We will model the supply chain and sales of a new product in a consumer market in the absence of competition. The supply chain will include the delivery of the raw product to the production facility, production, and the stock of the finished products. The QR inventory policy will be used. Consumers are initially unaware of the product; advertizing and word of mouth will drive the purchase decisions. The product has a limited lifetime, and 100% of users will be willing to buy a new product to replace the old one.

We will use discrete event methodology to model the supply chain, and system dynamics methodology, namely, a slightly modified Bass diffusion model, to model the market. We will link the two models through the sales events.

▶ Create the supply chain model:

1. Create a new model, and put together the process flowchart as shown in the Figure. All parameters not mentioned there should be left at their default values.
2. Open the properties of the **Main** object, and type this code in the **Startup code** field: **Supply.inject(OrderQuantity);**. This will load the supply chain with some initial product quantity.
3. Run the model.



The discrete event model of the supply chain

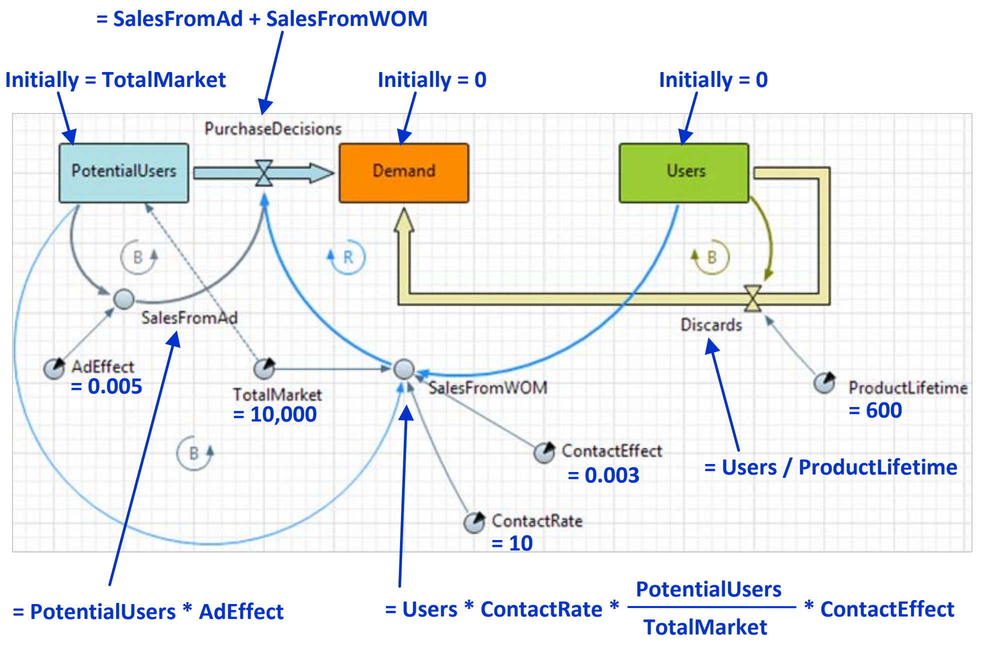
Four hundred items of the product are produced and accumulate in the **ProductStock**. Nothing else happens in the model (the **Supply** object is set up to not generate any new entities, unless explicitly asked to do so). The inventory policy is not yet present in our model.

Create the market model:

4. Create the stock and flow diagram as shown in the Figure. You can place the diagram below the supply chain flowchart.
5. Run the model.

The supply chain still produces the 400 items and stops, and the potential clients gradually make their purchase decisions, building up the **Demand** stock.

The difference of our market model from the classical Bass diffusion model with discards is that the users, or adopters, stock of the classical model is split into two: the **Demand** stock and the actual **Users** stock. The adoption rate in this model is called **PurchaseDecisions**. It brings **PotentialUsers** not directly into the **Users** stock, but into the intermediate stock **Demand**, where they wait for the product to be available. The actual event of sale, i.e., "meeting" of the product and the customer who wants to buy it, will be modeled outside the system dynamics paradigm.



The system dynamics model of the market (the modified Bass model)

Before linking the two models we will put a couple of charts on the model UI so we can better view the dynamics.

▶ **Create the charts to view the supply chain and the market dynamics:**

6. Add the time stack chart and the time plot of the data items as shown in the Figure. Set the **Time window** of both charts to 200 and let the charts **Display up to** 200 items collected every time unit.
7. Run the model again.

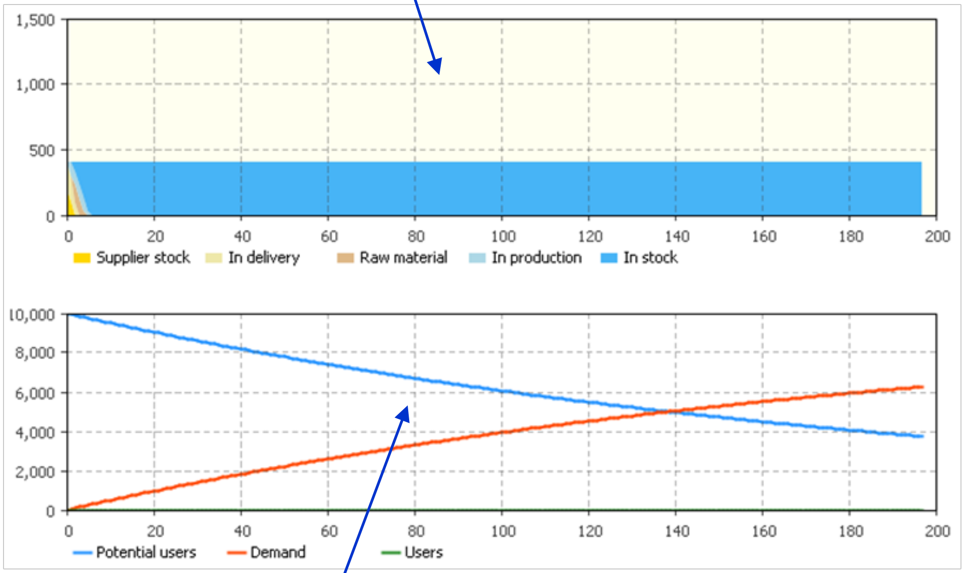
Note that in the current version of the model, the only reason for the potential users to make a purchase decision is advertizing. The word of mouth effect is not yet working because nobody has actually purchased a single product item.

How do we link the supply chain and the market? We want to achieve the following:

- If there is at least one product item in stock and there is at least one client who wants to buy it, the product item should be removed from the **ProductStock** queue, the value of **Demand** should be decremented, and the value of **Users** should be incremented, see the Figure A.

Time stack chart with four data items:
SupplierStock.size()
Delivery.size()
Production.size()
ProductStock.size()

Time window: 200
Dipspay up to: 200 latest items

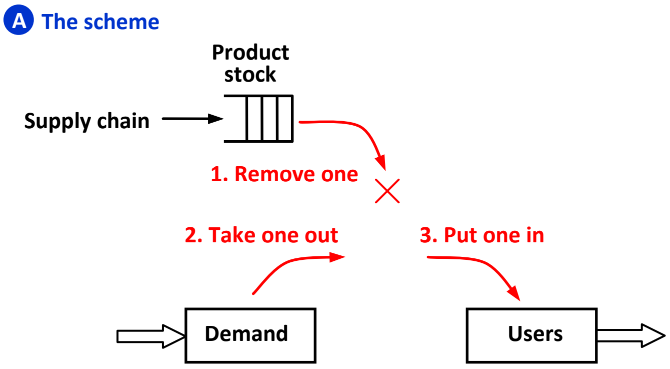


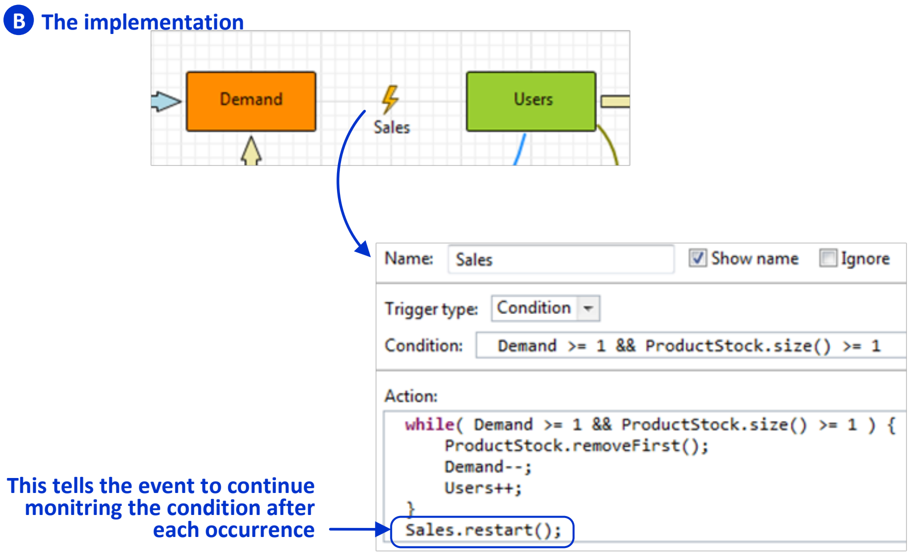
Time plot with three data items:
PotentialUsers
Demand
Users

Time window: 200
Dipspay up to: 200 latest items

The charts in the supply chain and market model

We, therefore, have a condition and an action that should be executed when the condition is true. The AnyLogic construct that does exactly that is the condition-triggered event.




B The implementation
Demand
Sales
Users
This tells the event to continue monitring the condition after each occurrence

Name: Sales ☑ Show name ☐ Ignore

Trigger type: Condition

Condition: `Demand >= 1 && ProductStock.size() >= 1`

Action:

```
while( Demand >= 1 && ProductStock.size() >= 1 ) {
    ProductStock.removeFirst();
    Demand--;
    Users++;
}
Sales.restart();
```

Linking the supply chain and the market: the scheme and the implementation

▶ **Link the supply chain and the market models with a condition event:**

8. Drag the **Event** object from the **General** palette, and place it in between the **Demand** and **Users** stocks (the location of the event object does not matter, but this way we can show that the event implements the missing Sales rate). Change the event name to **Sales**.
9. Specify the trigger and the action of the event, as shown in the Figure B.
10. Run the model.

In the presence of continuous dynamics in the model the condition of the event is evaluated at each numeric micro-step. Once the condition evaluates to true, the event's action is executed.

We put the sale into a **while** loop, because a possibility exists that two or more product items may become available simultaneously, or the **Demand** stock may grow by more than one unit per numeric step. Therefore, more than one sale can potentially be executed per event occurrence.

By default, the condition event disables itself after execution. As we want it to continue monitoring the condition, we explicitly call **restart()** at the end of the event's action.

The sales start to happen. The 400 items produced in the beginning of the simulation disappear in about a week. The **Users** stock increases up to almost 400; it then slowly starts to decrease according to our limited lifetime assumption. And, since we have not implemented our inventory policy yet, no new items are produced. This is the last missing piece of the model. We will include the inventory policy in the same **Sales** event; the inventory level will be checked after each sale.

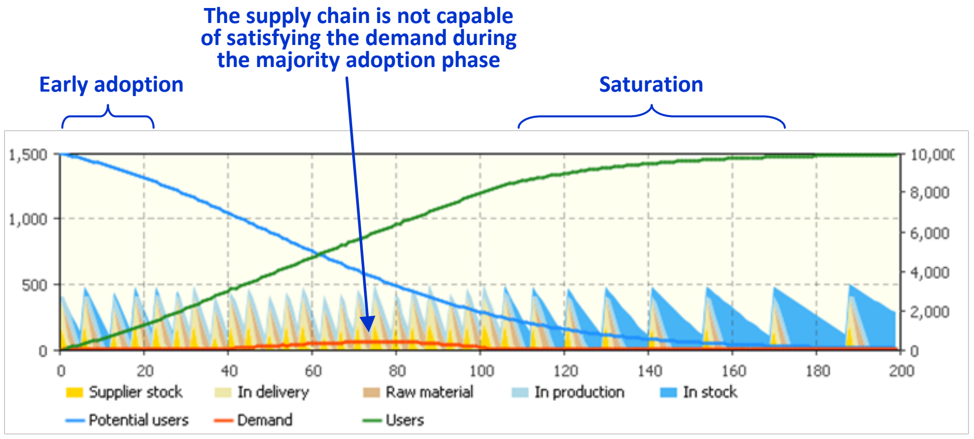
▶ **Implement the QR inventory policy:**

11. Modify the Action code of the Sales event as shown below:

```
while( Demand >= 1 && ProductStock.size() >= 1 ) {
  //execute sale
  ProductStock.removeFirst(); //remoove a product unit from the stock (DE)
  Demand--; //remove a waiting client from Demand stock (SD)
  Users++; //add a (happy) client to the users stock (SD)
}
//apply inventory policy
int inventory = //calculate inventory
    ProductStock.size() + //in stock
    Production.size() + //in production
    RawMaterialStock.size() + //raw product inventory
    Delivery.size() + //raw product being delievered
    SupplierStock.size(); //supplier's stock
if( inventory < ReorderPoint ) //QR policy
  Supply.inject( OrderQuantity );
//continue monitoring
Sales.restart();
```

12. Run the model.

Now, the supply chain starts to work as planned, see the Figure. (Here the inventory chart and the market charts are combined: the one was dragged on top of the other, and the labels were put on different sides.)



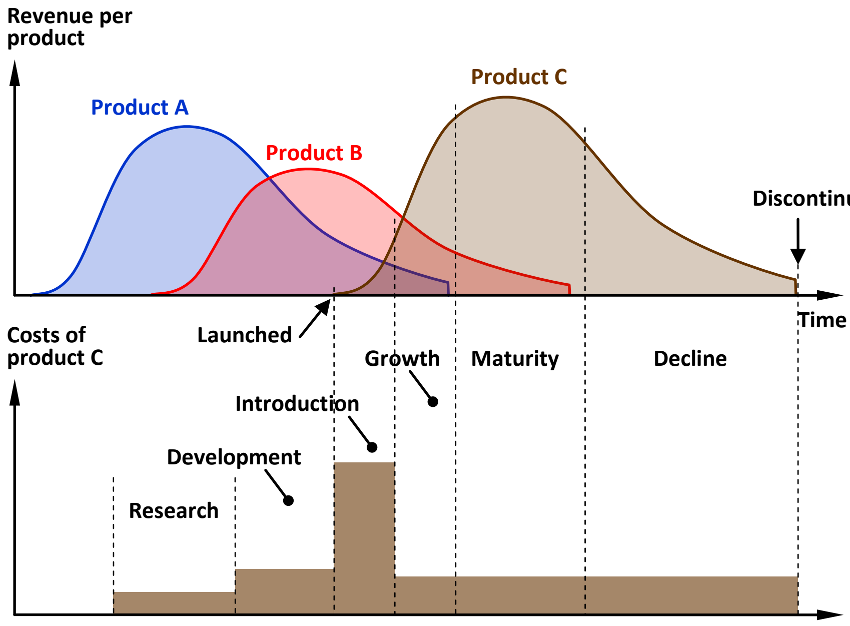
The supply chain dynamics pattern changes as the market gets saturated

During the early adoption phase the supply chain performs adequately, but as the majority of the market starts buying, the supply chain cannot keep up with the market. In the middle of the new product adoption (days 40-100), even though the supply chain works at its maximum throughput, still the number of waiting clients remains high. As the market becomes saturated, the sales rate reduces to the replacement purchases rate, which equals the **Discard** rate in the completely saturated market, namely **TotalMarket / ProductLifetime** = 16.7 sales per day. The supply chain handles that easily.

? Make the supply chain adaptive. Try to minimize the order backlog and at the same time minimize the inventory by adding feedback from the market model to the supply chain model.

Example: Product portfolio and investment policy

A company develops and sells consumer products with fairly short lifecycle. After the product has been successfully launched, its revenue peaks, and then falls, as shown in the Figure. To keep the business going the company has to maintain a continuous process of new product research and development. Part of the company revenue is therefore reinvested in R&D, and another significant part is spent on introducing new products. We will investigate how the investment policy affects the business.

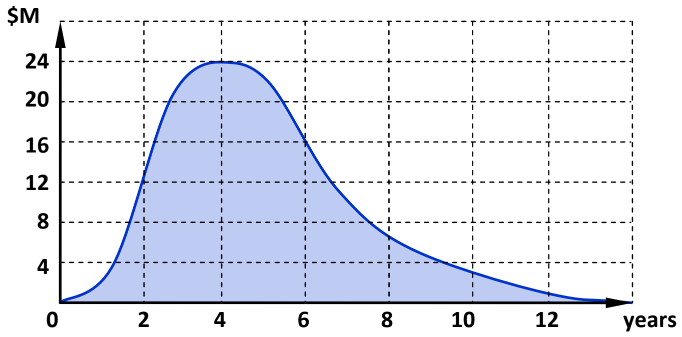


The product lifecycle and associated costs

We will make some simplifying assumptions:

The product lifecycle:

- The duration of the research phase of a product is uniformly distributed between 1.5 and 6 years. During this phase, each product is assigned a random "success factor", which is uniformly distributed between 0 and 1. At the end of the research phase, the product is killed if its success factor is less than 0.5; otherwise, it proceeds to the development phase.
- The duration of the development phase is also uniformly distributed between 1 and 3 years. During the development phase, the initial success factor is modified by adding a random number uniformly distributed between -0.3 and 0.3. At the end of the development phase, the project is killed if its success factor is less than 0.5; otherwise, it is released to the market.
- When the product is launched, its success factor is once again modified by adding a random number between -0.3 and 0.3. This value of the success factor then stays the same. As you can see, the value is between 0.2 and 1.6.



The base revenue curve of a product

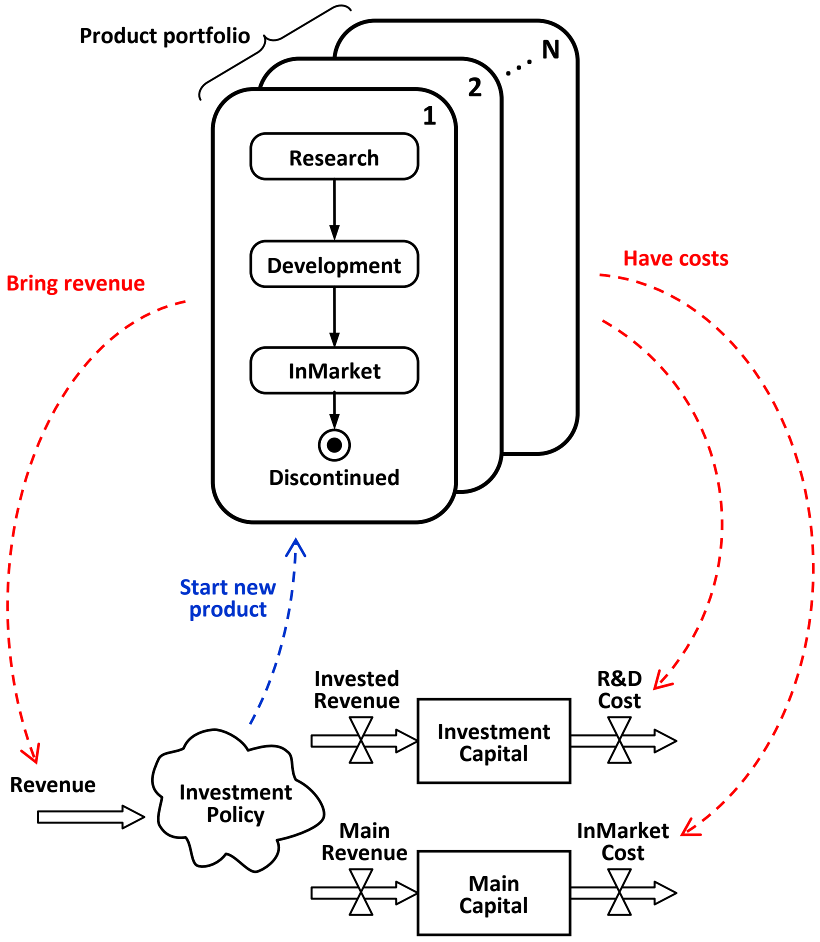
Revenue and cost:

- While in market, all products have the same base revenue over time curve (see the Figure), and the actual revenue equals base times the "success factor".
- The first two years of the product in market are considered as "introduction period". Any time after the introduction period the product is discontinued if its annual revenue falls below $5M.
- The annual cost of research is $0.5M per product and is same for all products. The annual cost of development is $1M per year. The cost of introducing a new product is $10M, which is spent evenly during the two years. In addition there is a one-time fixed cost of $0.5M for starting a new R&D project.
- After the introduction period we will assume no cost per product in market, in other words, we will treat the revenue as revenue after production and distribution costs.

Investment policy:

- A fraction of the company revenue goes into "investment capital". All R&D costs are paid from the investment capital.
- The company has a limited R&D capacity and cannot perform more than 100 projects concurrently.
- Once the company determines that the accumulated investment capital is greater than (the number of ongoing R&D projects + 1) times "average project cost" ($3M is assumed), a new project is started.
- The invested fraction of the revenue is determined as follows. If the accumulated investment capital is greater than the R&D capacity times "average project cost", no money is invested. Otherwise, 20% of the revenue goes into the investment capital stock.
- The remaining part of the revenue goes into the "main capital" stock, and product introduction costs are paid from there.

Assumptions, as you can see, are quite strong. For example, R&D projects may be killed at only two points: at the end of the research phase and at the end of the development phase, but not halfway through. The money spent on introducing the new product does not vary from product to product, the product lifecycles are similar, and there are no complete market failures and no great, long-lasting successes. All these things can be incorporated into the simulation model, but for the purpose of demonstrating the interaction of different modeling methods a simpler model will work just fine. Of course, all numeric values previously given are not fixed and will become the model parameters.



Architecture of the product portfolio and investment policy model

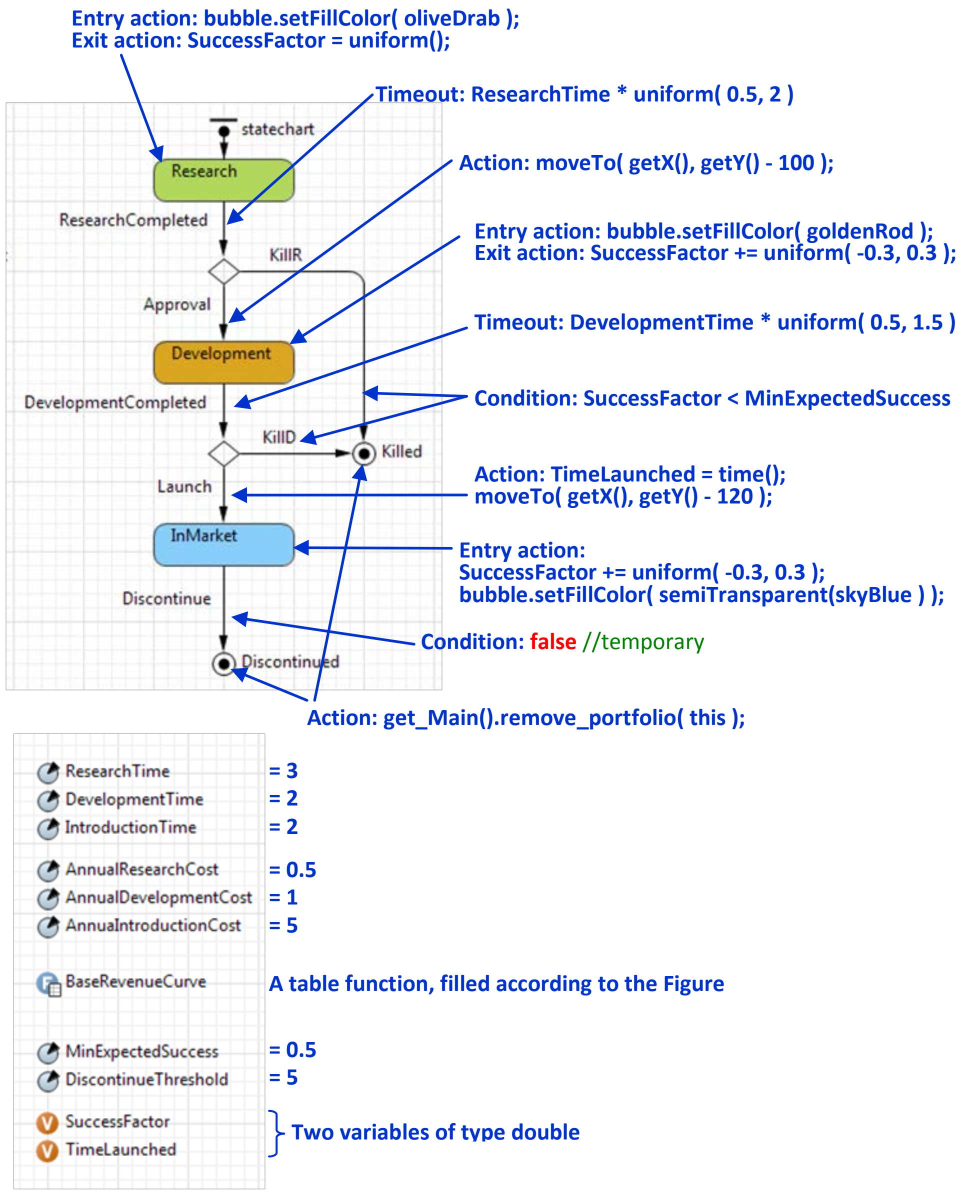
We will model each product individually as an agent (as you know, agents in agent based model are not necessarily people – they can be anything), so the product portfolio will be a population of agents. The company finances and investment policy will be a system dynamics component. The overall architecture of the model is shown in the Figure. The product lifecycle is naturally represented as a statechart that starts in the Research state and is deleted from the model when it is discontinued or killed. The implementation of the interface between the products-agents and the system dynamics part will be clear from the step by step description that follows.

Create the Product agent and the portfolio population:

1. Create a new model. Drag the **Agent Population** to **Main**, and specify the following parameters: **Agent class name**: **Product**, **Population name**: **portfolio**, **Initial population size**: 100, **Environment**: **Do not use.**
2. Open the editor of **Product**. Delete the default animation shape, and add the **Oval** from the **Presentation** palette instead. Call it **bubble**. We will animate the product portfolio in the form of a bubble chart.
3. Open the **Agent** page of the **Product** properties. Uncheck the checkbox **Environment defines initial location**, and set the **X** initial coordinate to **uniform(600)** and **Y** to **uniform(-60)**. This will distribute the bubbles randomly across a certain area.
4. In the same property page, set the **Velocity** of the agent to **100**. As the product will progress through the lifecycle phases, its bubble will move.
5. Open the editor of **Main**, and move the animation of the product portfolio (the bubble) to, say, (200,350). We need some space above, as the bubbles will move up.
6. Return to the editor of **Product**, and create parameters , variables, and the statechart as shown in the Figure. As you can see, the numeric values in the problem statement have become parameters in the model.

For example, the duration of the development phase was originally specified as uniformly distributed between 1 and 3 years. In the model, the parameter **DevelopmentTime** has initial value of 2, and in the timeout expression in the **DevelopmentCompleted** transition, it is multiplied by a random coefficient taking values between 0.5 and 1.5, which gives us the distribution we need. Should we decide to change the average value of the development duration, the resulting interval will correctly follow it.

7. Run the model.



Entry action: bubble.setFillColor(oliveDrab);
Exit action: SuccessFactor = uniform();

Timeout: ResearchTime * uniform(0.5, 2)

Action: moveTo(getX(), getY() - 100);

Entry action: bubble.setFillColor(goldenRod);
Exit action: SuccessFactor += uniform(-0.3, 0.3);

Timeout: DevelopmentTime * uniform(0.5, 1.5)

Condition: SuccessFactor < MinExpectedSuccess

Action: TimeLaunched = time();
moveTo(getX(), getY() - 120);

Entry action:
SuccessFactor += uniform(-0.3, 0.3);
bubble.setFillColor(semiTransparent(skyBlue));

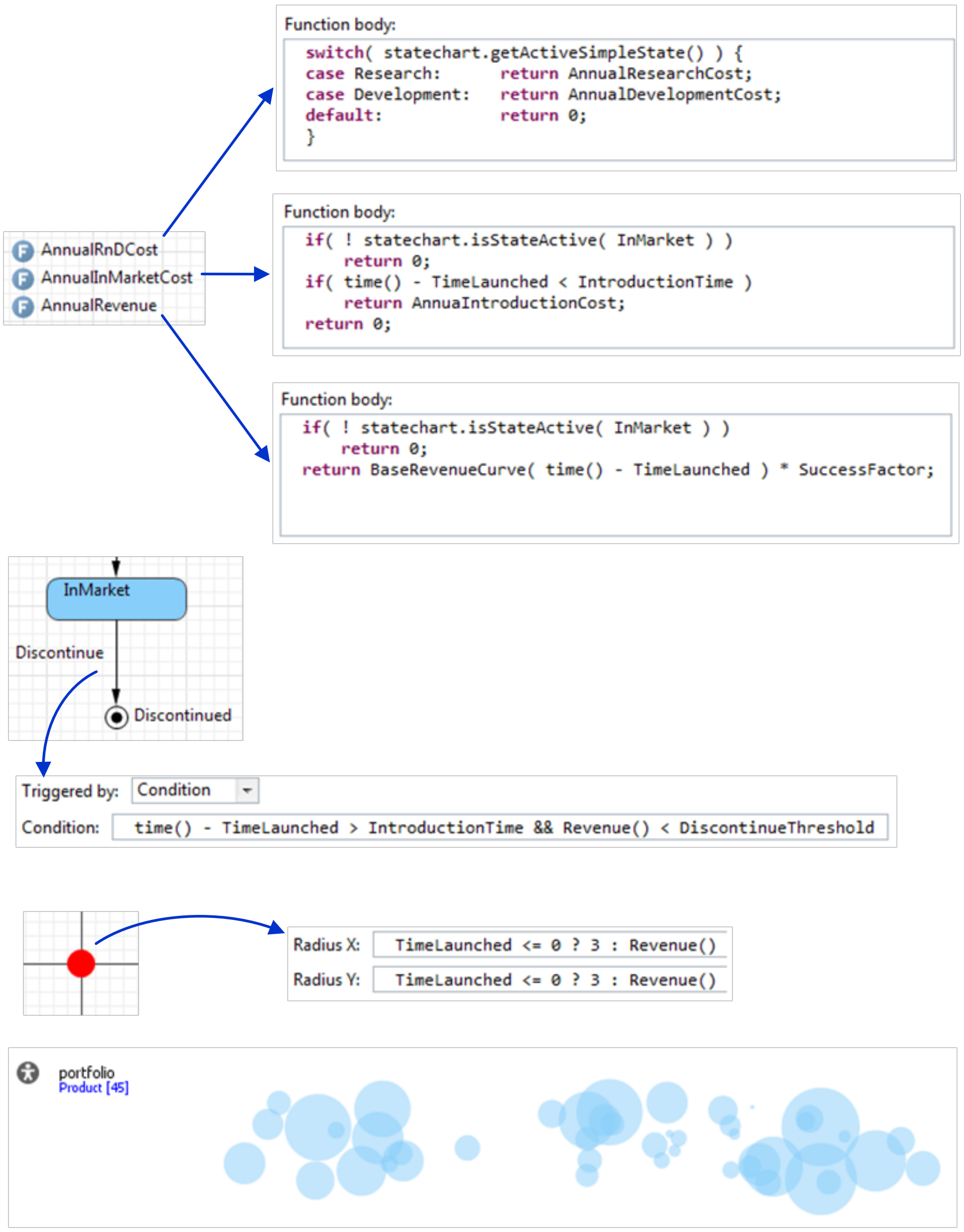
Condition: false //temporary

Action: get_Main().remove_portfolio(this);

ResearchTime	= 3
DevelopmentTime	= 2
IntroductionTime	= 2
AnnualResearchCost	= 0.5
AnnualDevelopmentCost	= 1
AnnualIntroductionCost	= 5
BaseRevenueCurve	A table function, filled according to the Figure
MinExpectedSuccess	= 0.5
DiscontinueThreshold	= 5
SuccessFactor	Two variables of type double
TimeLaunched	

Statechart, variables, and parameters of the agent Product

In the beginning of the simulation, you can see 100 olive-colored bubbles scattered randomly. After a while, some bubbles turn brown and move up, and some disappear; these are the projects that were killed after the research phase. Then, more bubbles disappear, and the rest turn blue and move further up – these are the products that go to market. As the condition of the **Discontinue** transition is at the moment set to false, the products will remain in the market forever. We will now add the cost and revenue calculation to the Product agent, fill in the missing condition, and enhance the animation.



Function body:

```
switch( statechart.getActiveSimpleState() ) {
case Research:      return AnnualResearchCost;
case Development:   return AnnualDevelopmentCost;
default:            return 0;
}
```

Function body:

```
if( ! statechart.isStateActive( InMarket ) )
    return 0;
if( time() - TimeLaunched < IntroductionTime )
    return AnnuaIntroductionCost;
return 0;
```

Function body:

```
if( ! statechart.isStateActive( InMarket ) )
    return 0;
return BaseRevenueCurve( time() - TimeLaunched ) * SuccessFactor;
```

Triggered by: Condition

Condition: `time() - TimeLaunched > IntroductionTime && Revenue() < DiscontinueThreshold`

Radius X: `TimeLaunched <= 0 ? 3 : Revenue()`

Radius Y: `TimeLaunched <= 0 ? 3 : Revenue()`

Functions calculating cost and revenue. Updated statechart and animation

Add the cost and revenue calculations, add the discontinue condition:

8. In the editor of **Product**, add three functions with return type **double** as shown in the Figure.

These functions calculate the cost and revenue based on the current state of the product. They use the statechart functions **isStateActive()** and **getActiveSimpleState()** to find out the state. However, while the product is in market, there is one special state that is not reflected in the statechart, namely this is the introduction phase that lasts for two years, according to our specification. To find out whether the product is in the introduction phase, we compare the time from the product launch (**time() – TimeLaunched**) and **IntroductionTime**. The time from the launch is also provided as an argument to the table function **BaseRevenueCurve()**.

9. Change the condition triggering the transition **Discontinue** as shown in the Figure. The condition reflects the fact that the product can be discontinued only after the introduction phase.
10. Select the **bubble** (the animation of the product), and open the **Dynamic** page of its properties. Provide the expressions for the bubble radius as shown in the Figure.
11. Run the model. Now you can see that bubbles of the products in market change their sizes dynamically as their revenue rises and then falls. Eventually, the bubbles seem to disappear.

The bubbles should disappear, because, according to the problem statement, the products have a limited market lifetime. However, you may notice that, although there are no bubbles at the end, the **portfolio** population still contains products. If you go inside any of the remaining products, you can see they are still in the **InMarket** state, despite their bringing in no revenue (and their bubbles have zero size!). The **Discontinue** transition does not seem to work: no products proceed to the **Discontinued** state, where it deletes itself from the model.

Why is this happening? To understand it, we need to recall how the trigger of condition type works. The condition is tested once when the statechart enters the **InMarket** state (at this time, it obviously evaluates to false). Then *it will be re-evaluated if something happens within the agent*, for example, an event occurs, or somebody calls the agent's function **onChange()**. Also, *the condition is constantly monitored if the model contains dynamic variables*. Our model does not have an SD component yet, and nobody is telling the products to check if their revenue has fallen below the threshold.

12. To finish the current phase and to see the product deleting itself from the model, open the editor of **Main** and add any dynamic variable from the **System Dynamics** palette, say, **Stock**. This is, of course, a temporary solution.
13. Run the model again. Now you can see the **portfolio** is depopulated and then disappears.

The next step is to model the company investment policy. This will be done at the **Main** level. We will create statistics items in the **portfolio** agent population calculating the total revenue and costs, and use the statistics in the system dynamics model of investment. After that, we will model the start of new R&D projects, which depends on the money accumulated in one of the SD stocks.

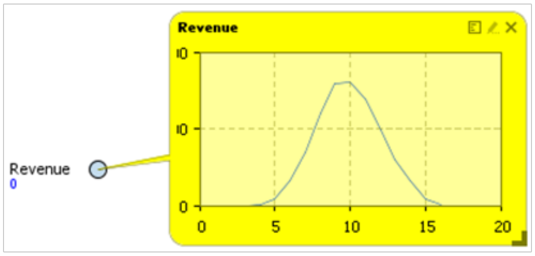
Add statistics items to the product portfolio:

14. Open the editor of **Main**, and select the **portfolio** population. Open the **Statistics** page of the population properties.
15. Create three statistics items (all three are of type **Sum** and have no condition):
    AnnualRevenue – the sum of **item.AnnualRevenue()**
    AnnualRnDCost – the sum of **item. AnnualRnDCost()**
    AnnualInMarketCost – the sum of **item.AnnualInMarketCost()**
16. Create the fourth statistics item that will count the number of products that are in R&D phase:
    NinRnD – the count of **! item.statechart.isStateActive(item.InMarket)**
    (it is easier to write that the product is not in market than to write that it is either in the research or in the development phase).
17. Delete the dynamic variable that you created in step 12 as a temporary solution. It is no longer needed.
18. Drag a regular dynamic variable from the **System Dynamics** palette, and call it **Revenue**. In the formula of the variable, type: **portfolio.AnnualRevenue()**.

You may remember that we did not recommend using the agent statistics functions directly in the formulas of dynamic variables, because the calculation may be time consuming. In this particular case, however, the number of products is not high, and we can afford it.

19. Run the model and watch how the value of **Revenue** changes over time, see the Figure.

As one can expect, the total revenue of several products launched approximately at the same time is similar to the base revenue curve of a single product. The company is not investing in new product research and development, and in about 15 years it goes out of business.



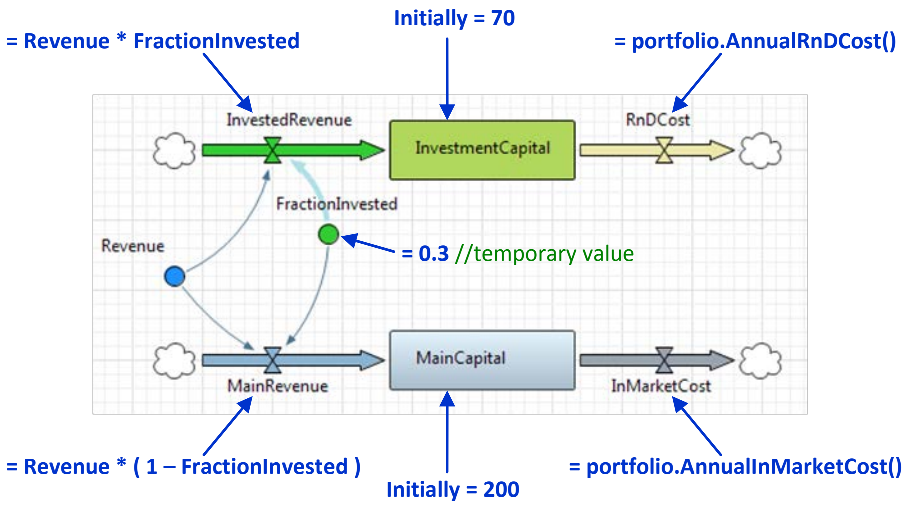
Total revenue of several products launched at approximately same time

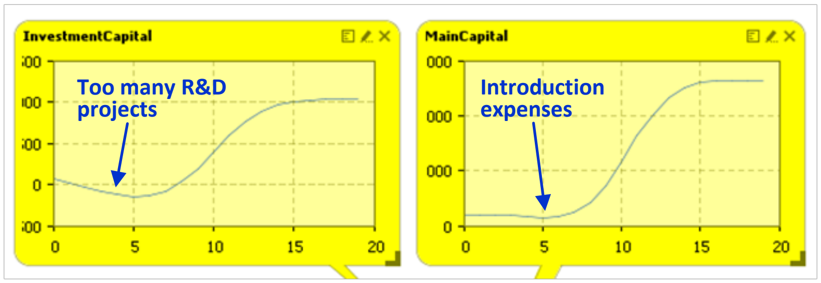
Create the first draft of the stock and flow diagram:

20. Starting with the **Revenue** variable created in the previous step, draw the stock and flow diagram as shown in the Figure.
21. Run the model.

As you can see, shortly after the model starts, the **InvestmentCapital** stock falls down below zero, see the Figure. It happens because at the beginning of the simulation the company starts too many (100) R&D projects simultaneously; they consume money, but bring in no revenue. As the products are launched in the market, the stock goes back up and then follows the S-shaped curve typical for systems with saturation. The **MainCapital** stock has a slight depression during the products' introduction period and then follows the same S-shaped curve.

In the next step, we will close the loop and implement the rule for starting new products, depending on the available investment money.



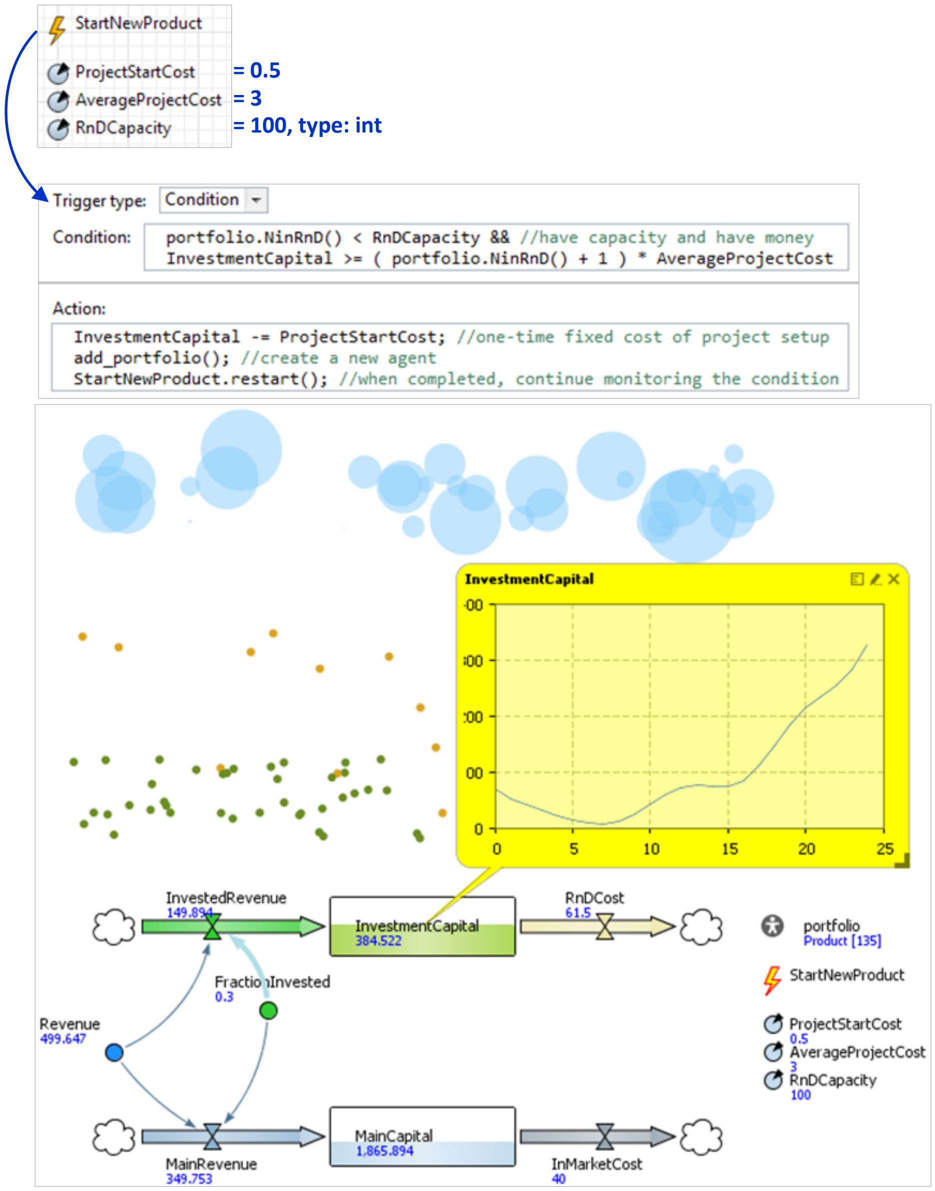


The first draft of the system dynamics model of investment policy

▶ **Create a condition event that will start new products:**

22. In the editor of **Main**, create three parameters and a condition-triggered event **StartNewProject** as shown in the Figure.
23. Select the **portfolio** population, and set its **Initial number of objects** to 0.
24. Run the model.

The parameter **AverageProjectCost** is an estimation of how much money will be required (per project) to finish all the ongoing projects plus a new one. The event **StartNewProject** is constantly monitoring the **InvestmentCapital** stock and, when it detects room for one more project, starts it. The one-time project setup cost is immediately subtracted from the stock, and a new **Product** agent is created in the **portfolio** population. The last statement in the event action (the call of **restart()** function) tells the event to resume monitoring the condition after each occurrence. Because new products are now created automatically, we can set the initial number of products to 0.


StartNewProduct
ProjectStartCost = 0.5
AveragePojectCost = 3
RnDCapacity = 100, type: int

Trigger type: Condition

Condition:
```
portfolio.NinRnD() < RnDCapacity && //have capacity and have money
InvestmentCapital >= ( portfolio.NinRnD() + 1 ) * AverageProjectCost
```

Action:
```
InvestmentCapital -= ProjectStartCost; //one-time fixed cost of project setup
add_portfolio(); //create a new agent
StartNewProduct.restart(); //when completed, continue monitoring the condition
```

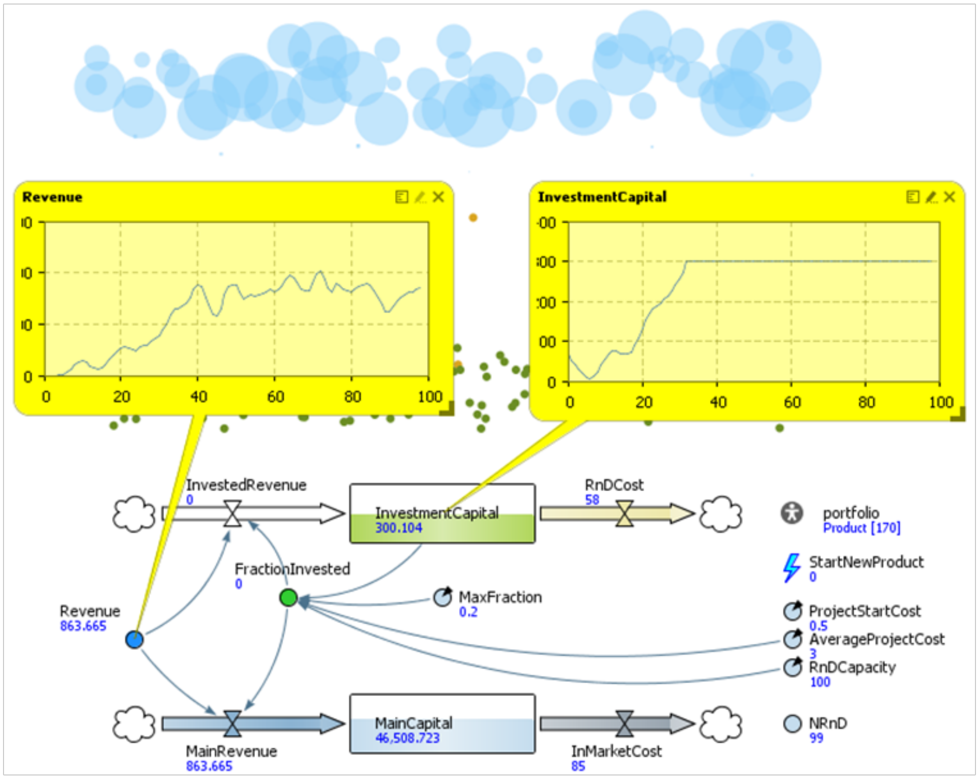
Condition event that creates new products. The new dynamics of the model

Now, company acts safely and is profitable. The continuous R&D process ensures that the company always has new products to offer. The revenue stream grows during the first three decades up to approximately $800M/year and then starts oscillating irregularly around that value. Further revenue growth is limited by the R&D capacity of the company.

Under the current model setup, 30% of the company gross revenue always goes into the investment capital stock, which continues to build up and remains largely unused. In the last step, we will implement the remaining part of the investment policy, namely, we will make the invested fraction of the revenue a variable that depends on the accumulated resources. This is done purely in the system dynamics part of the model.

▶ **Modify the formula for the invested fraction of the revenue:**

25. Add one more parameter: **MaxFraction** = 0.2.
26. Change the formula of **FractionInvested** to
    InvestmentCapital > RnDCapacity * AverageProjectCost ? 0 : MaxFraction
27. Run the model.



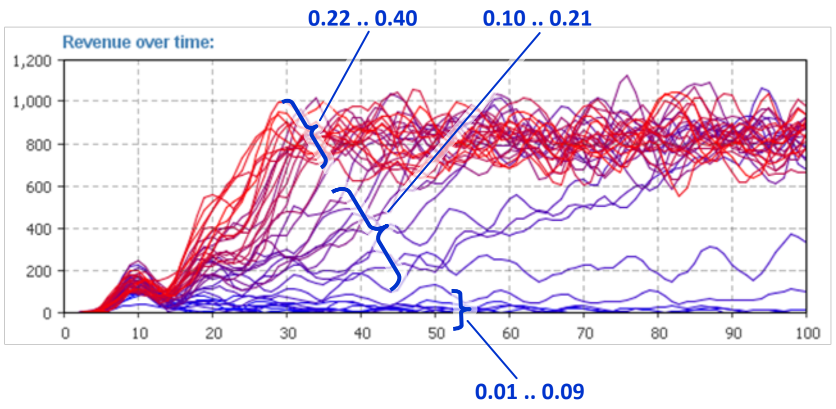
Total company dynamics under the fully implemented investment policy

The picture is different. The **InvestmentCapital** stock reaches the value of $300M and stops growing. The revenue oscillates around $800M/year, which, as we know already, is the upper limit with the given R&D capacity.

We can use this model to optimize the investment policy. For example, we can investigate how sensitive the company dynamics are to the parameters of the investment policy, say, to **MaxFraction**. We will compare the revenue over time curves obtained in different simulation runs.

▶ **Create a sensitivity analysis experiment:**

28. In the editor of **Main**, right-click the **Revenue** variable and choose **Create data set** from the context menu. A dataset **RevenueDS** is created. In the dataset properties, select **Update data automatically.**
29. Right-click the model (topmost) item in the **Projects** tree, and choose **New | Experiment** from the context menu. The experiment wizard opens.
30. In the first page of the wizard, select **Sensitivity analysis.**
31. In the **Parameters** page of the wizard, select **MaxFraction** as the varied parameter and specify **Min**: 0.01, **Max**: 0.4, **Step**: 0.01.
32. In the **Charts** page of the wizard, fill in one row of the table: **Title**: Revenue over time, **Type**: **dataset**, **Expression**: **root.RevenueDS.** Click **Finish.**
33. A sensitivity analysis experiment is created. Right-click it in the **Projects** tree, and choose **Run.**



Sensitivity analysis: revenue dynamics under different invested fraction values

The results are interesting. The revenue over time curves cluster into three groups, see the Figure. In the first one the company goes out business; this corresponds to the values of **MaxFraction** from 0.01 to 0.09. When the parameter is in the range 0.10..0.21, the revenue climbs up – the higher the **MaxFraction**, the faster it reaches the maximum value of $800M. Further increase of the invested revenue fraction does not affect the growth.

Discussion

When developing a discrete event model of a supply chain, IT infrastructure, or a contact center, the modeler would typically ask the client to provide the arrival rates of the orders, transactions, or phone calls. He would then be happy to get some constant values, periodical patterns, or trends, and treat arrival rates as variables exogenous to the model. In reality, however, those variables are outputs of another dynamic system, such as a market, a user base. Moreover, that other system can, in turn, be affected by the system being modeled. For example, the supply chain cycle time, which depends on the order rate, can affect the satisfaction level of the clients, which impacts repeated orders and, through the word of mouth, new orders from other customers. The *choice of the model boundary* therefore is very important.

The only methodology that explicitly talks about the problem of model boundary is system dynamics. However, the system dynamics modeling language is limited by its high level of abstraction, and many problems cannot be modeled with the necessary accuracy. With multi-method modeling you can choose the best-fitting method and language for each component of your model and combine those components while staying on one platform.

? A telecom company is about to introduce a new type of service, say, HDTV or high-speed Internet access and is planning the additional network infrastructure, the tariff policy, and the marketing campaign. Model adoption of the new technology by the users in the loop with the network infrastructure performance. Consider potential dissatisfactions effects, incremental growth of the infrastructure capacity, and ROI.