



Agent Modeling Guide

Miles Parker

Copyright 2010 Metascape, LLC

1. Introduction	1
1.1. Quick Start	1
1.1.1. Install	1
1.1.2. Welcome!	1
1.1.3. Cheat!	1
1.1.4. Follow the Tutorial	1
1.1.5. Follow your Interest	1
1.2. Agent-Based Modeling	2
1.3. Other Domains	2
1.4. The Agent Modeling Platform (AMP)	2
1.4.1. Agent Modeling Framework (AMF)	3
1.4.2. Agent Execution Framework (AXF)	3
1.4.3. Agent Graphics Framework (AGF)	3
1.4.4. Escape	3
1.5. Credits	4
2. Modeler Guide	5
2.1. Overview	5
2.1.1. Agent Modeling Framework	5
2.2. Structure	5
2.2.1. Overview	5
2.2.2. Details	6
2.2.3. Reference	11
2.3. Actions	12
2.3.1. Overview	12
2.3.2. Concepts	13
2.3.3. Details	21
2.3.4. Reference	32
2.3.5. Example	33
2.4. Functions	35
2.4.1. Overview	35
2.4.2. Details	35
2.4.3. Examples	45
2.4.4. Reference	47
3. User Guide	48
3.1. Overview	48
3.2. Modeling	48
3.2.1. Perspective	48
3.2.2. Creating Projects and Models	48
3.2.3. Menus, Popups and Toolbar	48
3.2.4. Views	49
3.2.5. Modeling Tree Editor	50
3.3. Building	56
3.3.1. Building Models	56
3.3.2. Generating Specialized Model Artifacts	57
3.4. Executing	58
3.4.1. Launching a Model (AMF)	58
3.4.2. Executing a Model (Java / 3D)	60
3.4.3. Controlling Models	60
3.4.4. Views	62
3.4.5. Visualization	64
3.4.6. Launching Other Targets	70
3.4.7. Model Parameterization	71
3.4.8. Model Testing	75
4. Tutorials	77

4.1. Designing a Model	77
4.1.1. Model Goals	77
4.2. Model Implementation	77
4.2.1. Setup	77
4.2.2. Structure	80
4.2.3. Actions	84
4.2.4. Styles	88
4.2.5. Actions 2 Movement Rule	90
5. Programmer Guide	96
5.1. Overview	96
5.2. Documentation	96
5.2.1. Online	96
5.2.2. Update Site	96
5.3. Installation	97
5.4. Example Java Models	97
5.4.1. Exploring Example ABM Models	97
5.5. Developing Models	98
5.5.1. Cheatsheet	98
5.5.2. Steps	98
5.6. Executing Models	98
5.6.1. Tutorial	98
5.7. Extending and Customizing AMP	98
5.8. Integrating Java and AMF Models	99
5.8.1. Method Action	99
5.8.2. Protected Code Regions	99
5.8.3. Interface and Base Class Generation	100
5.9. Converting Existing Ascape Models	100
5.9.1. Model	100
5.9.2. View	100
6. Installation Guide	102
6.1. Tools	102
6.1.1. Complete IDE	102
6.1.2. Eclipse and the Agent Modeling Tools	102
6.1.3. Extensions	103
6.2. Models	104
6.2.1. Obtaining Example Models	104
7. New and Noteworthy	106
7.1. Release 0.8.0	106
7.1.1. Model Visualization	106
7.1.2. Modeling Framework	107
7.1.3. Model Editing	109
7.1.4. Modeling Tools	110
7.1.5. User Experience	110
8. Resources	113
8.1. Websites	113
8.2. Papers	113
9. Support	114
9.1. Issues	114
9.1.1. General Support, Questions and Discussion	114
9.1.2. View Existing Bugs and Feature Requests	114
9.1.3. File a Bug	114
9.1.4. Discuss AMP Development and Project	114
9.2. Other Contacts	114
9.2.1. Professional Support	114

9.2.2. Media and Private Communications	114
9.3. Get Involved	114

Chapter 1. Introduction

In this manual we provide an in-depth understanding of what goes into an Agent Model, how you can use the Agent Modeling Framework and related tools to design one and present tutorials to get you started. But before going into details, let's take a quick overview at what agent-based modeling is and how the Agent Modeling Framework can help you to develop models for ABM or other problem domains.

We hope you enjoy using the tool, and we look forward to your comments and feedback and most of all participation!

1.1. Quick Start

If you're the kind of person who likes to jump right in, here are the basics.

1.1.1. Install

If you've already installed a complete Agent Modeling IDE, such as one offered by the project contributors, or you have already installed AMP from the Eclipse update sites, you can obviously ignore this step. Otherwise, refer to the Installation Guide at the end of this manual.

1.1.2. Welcome!

When you first open the IDE (or you first install the tools from an update site), you will see a welcome screen. If the screen isn't showing, select the **Help > Welcome** menu. Click on the "Overview", "Samples" or "Tutorials" buttons to begin your exploration.

1.1.3. Cheat!

The Agent Modeling tools come with a number of "Cheat Sheets" to help get you started. You can access them by selecting **Help > Cheat Sheets....** Then under the "Agent Modeling" category, select a cheat sheet, such as "Load Sample Projects".

1.1.4. Follow the Tutorial

The tools also come with extensive tutorials. See the "Tutorials" section for more help on that.

1.1.5. Follow your Interest

AMP has many potential facets and use cases. Check out the following sections of the manual for more information about..

1.1.5.1. ABM Researchers

Run an example model using the "Run an Example Model" cheat sheet.

1.1.5.2. ABM Modelers

Create Agent-Based models using simple visual tools and deploy them to a number of popular ABM platforms, including Escape. Follow the tutorial section of the Modeler's guide.

1.1.5.3. ABM Java Developers

Write "Plain Old Java" models within a full-featured agent-based modeling framework. Read Programmers Guide and then try the "Create an Escape Java Model" cheat sheet.

1.1.5.4. Eclipse Plugin Developers

AXF and AGF provide a number of features based around Eclipse technologies such as GEF, GEF3D, BIRT Charts and Zest. For example, AGF supports dynamic visualization of objects in space, and AXF supports

managed UI and headless execution for models of any kind. There isn't currently any general documentation for these features, but we encourage you to look at the APIs and at Escape's implementation of them. They're straightforward and can be useful for a variety of general science platform needs.

1.2. Agent-Based Modeling

The primary focus of the Agent Modeling Platform tools is "Agent-Based Modeling" (ABM). ABM is an innovative technique used to explore complex phenomenon in many domains, including economics, social sciences, biomedicine, ecology and business operations. ABMs share characteristics with object models, but are:

Spatial	Models have explicit environment(s) in which agents interact. (An environment need not be a physical landscape; other examples of spatial relationships include social networks or positions within a logic system.)
Temporal	Models change over discrete units of time.
Autonomous	Agent behaviors are activated independently from other object requests.
Heterogeneous	Agents may share behavior definitions but have apparent and distinct states and behaviors.
Collective	Models contain large communities of agents which exhibit collaborative and competitive behaviors.
Emergent	Agents have collective macro-behaviors that are non-obvious from agent micro-specifications.

Existing scientific models are very good at representing relatively simple systems, but generally speaking aren't very good at representing complex systems. The world is full of complex systems, and our misunderstanding of these systems has prevented us from addressing many of the key challenges facing the world, including the global financial crisis and climate change -- in fact once could argue that our misunderstanding of these systems has strongly contributed to these crises.

Agent-Based Models (ABMs) seek to represent important real-world dynamics by designing communities of software agents that mimic real entities. Rather than make simplifying assumptions about such systems and then representing them in equation form or as off the shelf algorithmic constructs, the ABM researcher aims to identify key agent state, interaction spaces, and behaviors. Agents are then "let loose" on our computers and we explore what happens next. The computational horsepower exists today to simulate large numbers (e.g. >>10) of interacting, adaptive and autonomous agents but often desktop computers are all we need to explore significant domains. ABMs have been designed to represent all kinds of important natural systems, at scales reaching from cellular mechanics to international trade and are being used to solve truly hard problems in government, business, and academia. ABMs are not a solution to every problem, but they can help us to appreciate and gain unique insight into many systems, and often they can help us to come up with better practical decisions than we might using classic approaches.

1.3. Other Domains

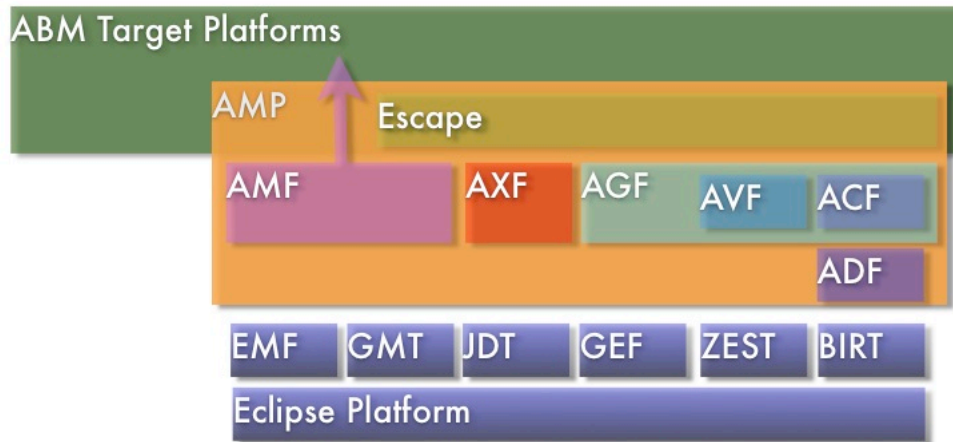
Many kinds of objects share characteristics of ABM agents -- after all, software agents are used in many other contexts. The Agent Modeling Framework meta-modeling support is potentially suitable (or extendible) for a number of approaches outside of ABM; for example business rules, object interactions, systems dynamics models and traditional discrete event models. Similarly, AMP execution and graphic support can be used for modeling natural systems but could also be used to manage other software agents -- for example independent reasoning tasks or dynamic visualization support.

1.4. The Agent Modeling Platform (AMP)

The Eclipse Agent Modeling Project (Incubation) or "AMP" provides the core support for both open source and commercial modeling tools. AMP in turn is built upon Elipse, the most powerful, well-supported and popular Integrated Development Platform (IDE) available anywhere.

AMP provides extensible frameworks and exemplary tools for representing, editing, generating, executing and visualizing ABMs and any other domain requiring spatial, behavioral and functional features. AMP has two main themes that complement but don't depend on one another, modeling of agent systems (AMF) and execution and exploration of those systems (AXF, AGF and Escape).

The overall AMP architecture and project dependencies are summarized in the architectural diagram below:



1.4.1. Agent Modeling Framework (AMF)

AMF provides an ABM meta-model representation, editor, generator and development environment. The AMF Acore meta-model is similar to EMF Ecore and defined in Ecore, but provides high-level support for complex agents. AMF generates complete executable models for Escape, Ascape and Repast Symphony, as well as Java Skeletons and Interfaces, JUnit test cases and documentation and is easily extensible to support additional targets. For more on AMF, see the Modeler Guide.

1.4.2. Agent Execution Framework (AXF)

The execution framework provides services and UI for model management, execution, and views. Arbitrary toolkits can easily integrate with Eclipse and AXF by implementing pluggable providers like engines, agents and view parts. AXF is not just for ABM -- anyone who needs support for executing, managing and visualizing collections of objects may find it useful. AXF user tools are covered extensively in the User Guide, and the forthcoming Platform Developers Guide will provide information about integrating AXF in your own tools.

1.4.3. Agent Graphics Framework (AGF)

The graphics framework extends GEF, GEF3D, Zest, and the BIRT charting engine to support real-time visualization of and interaction with agent models. AGF currently provides support for 2D, 2 1/2 D. and graph structures, and will be extended to 3-D, GIS and others. As with other AMP components, the AGF design focus is to provide an extensible infrastructure so that platform adopters can easily create their own view and editor parts. AGF user tools are covered extensively in the User Guide, and the forthcoming Platform Developers Guide will provide information about integrating and extending AGF in your own tools.

1.4.4. Escape

Escape is an exemplar ABM toolset. It's based on Ascape, which has been in use for more than 10 years. The core API is very stable, and that should give users a way to explore the features of AMP without concerns about keeping in synch with the rapidly evolving AXF /AGF API. It allows modelers to code in Java and/or generate models with AMF and then execute those models within the same development environment. Escape is the primary target for most of the models in this guide. Those users interested in writing code directly to the Escape API can refer to the Programmer Guide.

1.5. Credits

Miles Parker is a consultant and software developer with over ten years of experience in the agent-based modeling field and twenty years developing object-oriented tools and frameworks and is the architect and project lead for the Eclipse Agent Modeling Platform (Incubation).

Metascape, LLC is the primary contributor to AMP and provides Agent-Based Modeling tools, including sophisticated tools based on the AMF platform, and comprehensive consulting services. For more information on Metascape products and services visit <http://metascapeabm.com>.

The AMF meta-model started life as "[score](#)", a component of the Repast Symphony environment. Argonne National Labs supported the initial development in 2007 and agreed to contribute the relevant IP to the Eclipse project in 2009. It wasn't fully utilized within Symphony, and migrated from there to the MetaABM project supported by Metascape. The MetaABM project defined a complete editor, code-generation and model execution suite that formed the initial AMP AMF contribution. Metascape has sponsored the development of AMF from 2007 through the present.

The AMP logo was inspired by the <http://swarm.org> [Swarm logo](#) with their kind permission.

Chapter 2. Modeler Guide

2.1. Overview

In this section we present the design of the Agent Modeling Framework and explain how it can be used to create models that are transparent, composable and adaptable. Fundamentally, an agent-based model, or "ABM", is composed of five pieces: Agents and Context Agents, Attributes, Spaces, and Actions. The first three refer to structural components, whereas Actions define behavior. Agent models also have styles, which are a special kind of Action used to determine how to portray an agent in a visualization. Finally Actions make use of Functions. We'll describe of these components in a separate section.

2.1.1. Agent Modeling Framework

The Eclipse Platform provides many unique features that make it ideal for an ABM platform. AMF provides easy to use and powerful tools and techniques for designing Agent-Based Models, including a common representation, editors, generators and development environment.

The Agent Modeling Framework (AMF) provides high level representations for common ABM constructs, and introduces novel ways of representing agents and their behaviors. As detailed in other documentation sections, the Agent Modeling Framework and related tools have been designed to allow researchers to explore complex models in an intuitive way. One of our major design goals has been to create tools that non-programmers can use to create sophisticated models. It has been our experience that using Model-Driven Software Development (MDS) techniques increase productivity for all developers regardless of skill level.

The foundation of the Agent Modeling Framework is "Acore". The current version uses an interim version of Acore called "MetaABM". We refer to the AMF models as "meta-models" because they are used to define *how* Agent-Based Models are themselves modeled. For those familiar with Eclipse Model-Driven Development tools, AMF is analogous to EMF but is targeted toward the design and execution of models composed of agents. Acore and MetaABM are defined in Ecore but provide a more direct and high-level ABM representation of agents, including spatial, behavioral and functional features sufficient to generate complete executable models for the target platforms. AMF is fully integrated with the Eclipse IDE platform, but Acore models themselves need have no dependencies on any particular technology beyond XML/XSD.

Models designed in AMF are transparently converted to Java code for leading Agent-Based Modeling tools, such as the Escape tools which are included in AMP and allow direct execution of models within the AMP environment, and Repast Symphony, another popular Java based ABM tool. These tools create Java code that can then be compiled, executed and event modified in these environments just as with any other Java program. AMF's generative capability is designed to be pluggable and modular so that other developers can create AMF generators for their own tools. In fact, targets can be designed that have no inherent dependencies on Eclipse or even on a traditional platform.

The Acore / MetaABM meta-model is made up of three main packages. This is all based on MetaABM, and while names and important details will change for Acore, the core design should be quite similar.

2.2. Structure

2.2.1. Overview

The basic structure of an agent-based model can be quite simple. While there are many subtle complexities -- beyond the scope of this manual -- we can construct most models following some straightforward and elegant design principles. And in fact, one of the main goals of the Agent Modeling Framework is to provide a consistent framework that can support using those principles to support the creation of models that can be easily understood, shared, and that can be used interchangeably as components in other models.

Unlike the approach of a traditional Object Oriented environment, the dominant organizing principal for agents within AMF follows a compositional hierarchical model, not an inheritance model. (Inheritance-like

behavior will be supported in forthcoming versions of Acore, but in a more sophisticated, flexible and dynamic way than is supported by traditional programming languages such as Java.) Contexts -- also referred to as "Swarms" or "Scapes", are simply Agents that are capable of containing other agents. With this basic construct -- known as a Composition in design pattern language -- agents are able to contain other agents.

2.2.2. Details

2.2.2.1. General

Everything represented in an Acore model needs to be referred to in some way. Just as software classes have names, Acore provides a way to label and describe every entity, but in a richer and more maintainable way. All entities in Acore -- including Actions and Functions which we describe in the next two sections -- have a number of shared values.

Named Entities

Label

A reasonably short, human readable name for the agent. For example, "Timber Wolf", "Exchange Trader" or "Movement Probability". These should be defined so that they fit in well with auto-generated documentation. Note that Labels must be unique throughout the model. (This may change in future releases for Action names.) If you try to provide an object with a name that is already in use, "Copy" will be appended to the end of the name.

ID

An ID is an identifier that can be used to represent the object in a software program. This means that it must follow certain rules such as no-spaces or non alpha-numeric characters. The editing tools will help make sure that this value is legal. Note that when you enter a label, a legal ID is automatically created for you! Usually you won't need to change this, but you might if for example you want the value to match up with some database or other external representation. So reasonable values here might be "timberWolf" or perhaps "MVMNT_PRB" if say you're trying to match up to some old statistics records you have. (Note that currently IDs by default use "camel case", i.e. "thisIsAnExampleOfCamelCase" to match with Java variable naming conventions, but this is likely to change.) Like labels, IDs need to be unique across the model, and the editing tools will assign a different id if you attempt to give two entities the same id.

And most entities also define:

Description

A complete textual description of the object. Don't overlook this -- it is the most important part of your model. The description will show up in your auto-generated html documentation, software documentation and even in your running model. You should include enough information that model users will understand what the entity is for and what it does without referring elsewhere. This is also where any attributions and references should go. You can put html tags in here -- such as href's to external papers, but keep those to a minimum as they won't be rendered as html in all contexts.

Plural Label

The plural representation of an entity. This can be a surprisingly useful thing to have in generated documentation and software code, so it's worth maintaining. The editor will automatically add an "s" at the end of the label you've entered above, but you change it to whatever is appropriate. For example "Person", or "Timber Wolves".

2.2.2.2. Agents

Simple Agents

An Agent is simply a software object that has autonomous behavior and (generally speaking) exists within some set of spaces. By autonomous, we mean that agents make the choice about when and how to execute

a behavior, as opposed to have that controlled from "above", so to speak. Like any software objects, agents have attributes (fields) and actions (methods) associated with them.

Attributes

As described in the attribute sections above.

Actions

Described in the "Actions" section.

Styles

Special actions that are used to define how to draw an agent graphically as also described in the "Actions" section and detailed in the "Functions" section.

Context Agents (Contexts)

As detailed above, agents also form the basic structural component of an agent-based model. To get an idea for how this works, have a look at the "EpidemicRegional.metaabm" model. Note that in future releases, we will probably refer to Contexts as "Scapes".

Agents

The agents that are contained within this context. For example, a context representing a city might contain an Individual Agent for defining all individuals in the model, and a Vehicle agent defining all vehicles. Note that when we refer to an agent in this context, we mean a general type or class of agent, not the agent itself.

Spaces

The set of all spaces contained or subsumed by the agent. For example, a context representing a city might contain a geographical space and a transportation network space.

2.2.2.3. Attributes

Agents need some way to represent their internal state. For example, an agent in an economic model might have a wealth attribute, and an agent in an ecology model might have a quantity of food and a particular vision range. These states are represented as attributes just as all software objects do. In an agent model, we keep richer information about these attributes and generally represent them at a higher level. For example, rather than specify that a real value is a "double", a "float" or a "big number", we represent them as "Reals" so that they can be implemented and tested in different environments. This might allow us for instance to ensure that a model's behavior is not dependent on a particular machine implementation of floating point arithmetic. Also, note that attributes only represent so-called "primitive" values -- that is, actual measurable features of a particular agent but not an agent's relationship to other agents or objects. See the discussion about networks for more on this key topic.

Here are the basic types of attributes available in Acore models:

Basic Attributes

Attributes are single values that a given agent contains. For example, an agent might have an "Age" attribute. In this section we go over the values you can set for the attributes. For those with a technical bent, note that we are technically describing the meta-attributes for the meta-class "SAttribute". But it is far too confusing to refer to the attributes of attributes! So we'll just refer to the attributes that any of our model components as "values".

Type

These can be anyone of the following:

Boolean

A value that is simply true or false. Note that unless this value really is a simple binary value, you should consider using state instead.

(See details below.) For example, rather than representing gender as a 'Female' boolean value, define a 'Gender' state with values 'Male' and 'Female'. Generated artifacts and documentation will be much clearer, and you'll be able to easily modify the model later if for example you discover that there are more than two potential gender categories that are relevant to your model.

Integer

A discrete whole number value, such as "100", "-1", "10029920". It's generally a good idea to represent any value that can never have a decimal value as an integer.

Real

A continuous number value. While these are typically represented in software as floating point numbers, they can conceivably represent numbers at any arbitrary precision and in any scheme. Note that while technically speaking we should be representing irrational numbers, this is not currently supported for default values and users should simply use the closest decimal approximation.

Symbol

A string representing some state. More precisely, a computationally arbitrary value with contextual meaning. This could be any kind of identifier. For example, you might use it to store some kind of input coding from data that is then converted into an object state. Or it could simply be an agent's name. But theoretically (though this is not currently supported) one could imagine a symbol using an idiogram, an icon or even a sound that identifies or represents the agent in question.

(Undefined and Numeric types should not be used within a well-defined model.)

Default Value

The value that should be assigned to the attribute at the beginning of any model run. The attribute may of course be assigned a different value in an Initialize rule, but its a good idea to specify one here. It's OK to leave it blank, in which case a sensible 'empty' value will be assigned, i.e. false for Boolean, 0 for Integers and Reals, and an empty string for Symbol.

Gather Data

Here you can specify whether executing models should collect aggregate values for the data. For example, if you select this value as true for a 'Wealth' attribute, Escape will automatically keep track of minimum, maximum, average, sum and optionally standard deviation and variance across all agents for each model execution period. All of these statistics will then be selectable with a mouse click to appear in your model charts at runtime.

Immutable

This value indicates whether you expect the model value to change. If you know that it won't or shouldn't, this value should be true.

Derived

A derived attribute is one whose value is detemrined solely based on other agent attributes. A derived value is always associated with a Derive root action which is created automatically by the editor. See the documentation on the Derive action for more details. An attribute cannot be both derived and immutable.

Units

Specifies what the attribute is actually measuring. For example, if you're defining an attribute for "Wealth" in the context of a model of the world economy, you might specify "USD". If you're defining "Age", you might specify "Years", for "Mass" "Kg". Like description, this value is often overlooked, but can be critically important to allowing yourself and others to understand and correctly calibrate a model. Note that this will also allow you to simplify variable names -- instead of using "Age in Years", you can simply specify "Age"

and the appropriate unit. It may be obvious to you that your model is concerned with age in years, but a user who needs to specify a different granularity will be grateful for more clarity.

Arrays

Arrays are simply attributes with zero or more entries. For example, you might have an array of three Real numbers representing the Red Green and Blue color components for an object. Note that if you find yourself defining very complex sets of arrays, its likely that what you really want to define is a new agent with attributes for each array. In addition to what is defined above, arrays specify:

Size

The number of values that the array attribute will contain.

States

States represent any agent quality that may take on one of a number of well defined values. For example, an "Ice Cream Consumer" Agent might contain a state of "Ice Cream Preference" with options of "Chocolate", "Vanilla" and "Ginger".

State Options

Create new options for states by adding them to the state node. States are simple described items. Don't forget to provide a description! States also have

Default Option

unlike for regular attributes, this option is not optional! Simply pick a state option to be assigned that the agent will take if no other option has been assigned.

2.2.2.4. Spaces

All contexts can contain spaces. Spaces provide an environment in which agents can have a physical or notional location and upon which they can interact with one another and their environment. Agents can exist in more than one space at a time, and a given agent need not exist in every space. (Note that this is different from the Repast Symphony notion of projections, though the two representational approaches are generally compatible.) Agents need not represent explicit, "real" spatial structures such as a landscape, though this is of course the most common use. They can also represent relational and state information, such as a belief space or a social network. There are four kinds of space represented:

Space (Continuous)

In the modeling tools, we simply refer to this as a "Space" as it represents the general concept of a space. A space is simply something that contains objects with certain locations and extents and has a certain number of dimensions. The space is continuous, in the sense that objects can be placed anywhere with arbitrary position. Spaces hold attributes, which simply define their dimensions-- see below.

Border Rule

A value representing what happens to an agent when that agent is asked to move beyond its extent.

Periodic

When encountering an edge, the agent will treat the space as wrapping around to the other side of the space. For example, if the agent at location {1,2} (0-based) within a Moore space (see grid discussion below) of size {10,10} is asked to find some other agent within distance 3, the agent look in the square defined between {8,9} and {4,5}. An agent asked to move beyond the confines of the space will simply stop when it reaches the edge. You can imagine this as taking a piece of graph paper and connecting the opposite edges. You can't actually do that with paper, but if you could you would have a toroidal (donut) shape in three dimensions defining the shape in two.

APeriodic

When encountering an edge, the agent treats it as the edge of the space. For example, if the agent at location {1,2} is asked to find some other agent within distance 3, the agent look between {0,0} and {4,5}. An agent asked to move beyond the confines of the space will simply stop when it reaches the edge.

The "Strict" and "Bouncy" values are obsolete and should not be used.

Dimensionality

The number of dimensions that the space has. After selecting a dimensionality, attributes will be added to represent each dimension. For example, if you enter 3 here, you will have an attribute for X, Y, and Z. Be sure to enter default values here, as they will be used to specify the actual size of the space.

Grid

A grid is technically a regular lattice structure. Currently, only rectilinear structures are supported, i.e. a One-dimensional vector, a two-dimensional grid, a three-dimensional cube and so on. (Though none of the current target platforms support n-d spaces yet.)

Like continuous spaces, a grid has a border rule and dimensionality. A grid has a couple of other important values:

Multi-Occupant

Does the grid allow more than one agent to occupy it at a time? This value may be replaced with another mechanism in future releases.

Neighborhood

This value determines what constitutes a region within a particular distance from the agent. The value for this is often critical in obtaining particular behavior from a model, and shouldn't be overlooked. There are three possible values:

Euclidean

The distance between any two cells is taken to be the "real" distance. For example, if an agent was within a chess board, and we wanted to find all agents within distance three of it, we could determine that by taking a string of length 3, tacking it to the center of the source square, and including all cells whose centers we can reach with the other string end. Note that although Euclidean space may seem the most reasonable neighborhood configuration to choose, this really isn't the case. Euclidean space is continuous whereas grid space is discrete, and mapping the two to each other can create unexpected issues. Still, this is a good choice for models representing notional real spaces.

Moore

Here, the distance between any two cells is defined by the number of edge *or* corner adjacent cells crossed to get between them. To continue the chess board analogy, this is the set of moves that a king can make. Note that this does not map well to real space at all, as a cell at distance 1 in a moore space is at distance $\sqrt{2}$ in "real" space.

Von-Neumann

Here, the distance between any two cells is defined by the number of edge adjacent cells crossed to get between them. This is the set of moves that a rook might make on a chess board -- if a rook could only move one square at a time. It is also often referred to as a Manhattan distance for the self-evident reason.

Network

A network represents a set of relationships between agents, in a graph structure. The concept is pretty simple, but note that a network is actually a critical part of many AMF models. This is because we use networks to

A network has only one value to specify:

This indicates whether connections between agents are one-way or two-way. If this value is false, then if a connection is made between an agent A and an agent B, and agent B searches within distance 1, agent B will find agent A. If this value is true, agent A can find agent B, but agent B can not find agent A. (Unless of course some other path leads B to A.)

A geography represents a physical landscape. Here we assume that that landscape is going to be defined by an external data source and representational scheme -- typically a Geographical Information System. We'll describe how to work with GIS in more detail when we discuss builder actions.

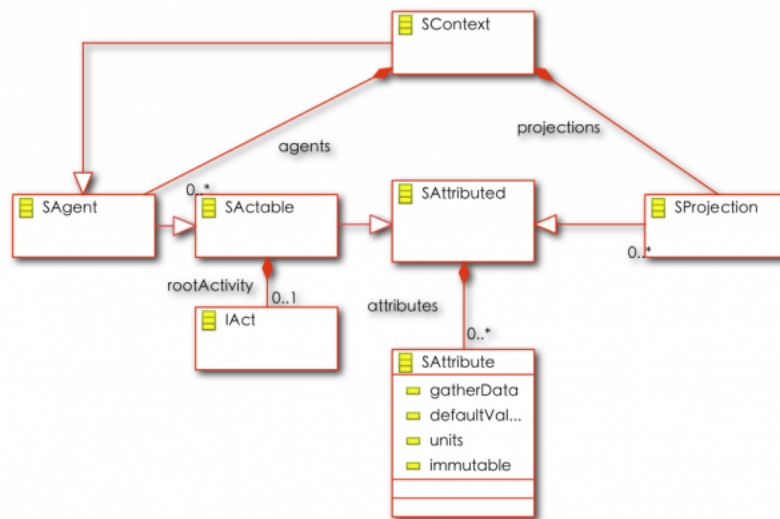
2.2.3.1. Diagrams

Meta-Classes

[illegible]

AMP 0.7.0

Key Collaborations



Core interactions are in Red. The meta-model structure is essentially a Composite pattern.

Details

1. Every model has at its root a Context (Scape). Contexts are Agents that are capable of containing other Agents (including other contexts, naturally).
2. (Meta-level) Contexts contain (meta-level) Agents at the model (design-time) level. At runtime, (model-level) Context instances defined in a (meta-level) SContext will contain (model-level) agent instances of the defined (meta-level) SAgent. This sounds more complicated than it is, so let's look at a simple example. Suppose we create a new Context, and give it a label of "Wiki Example". Within that Context, we create an Agent and give it a label of "Individual" and another Agent with the label "Block". At runtime when we are actually executing the model we will have one WikiExample model instance which contains a number of Individuals.
3. Agents contain Attributes, such as Vision and Age. Context attributes often represent input parameters for contained agents. For example, our Wiki Agent Context might contain an Individual Count as well as a Minimum Age and Maximum Age.
4. Contexts can contain Projections (Spaces), which represent some kind of spatial or structural interaction space for the agents; either a grid, a continuous (euclidean) space, or a network (graph) or geographic space of some kind. For example, we might want to have a City that contains Blocks and that an Individual can move around within.
5. Agents are Actable and thus can contain any number of behaviors called "Actions", described in detail in the next section. Actions can describe individual behavior, and at the Context (Scape) level can define how member Agents and Projections are created.
6. Styles provide a mechanism for defining generic visualization behavior for Agents and so are also Actable. For example, an Agent might have an Action that says effectively "draw a red circle shaded red for the wealth of the agent".

2.3. Actions

Actions are the most important part of an agent model. While Agents, Attributes and Spaces define what we're modeling, it is Actions that give the models life.

2.3.1. Overview

Actions allow the definition of behavior for agents at a very high level. You can think of actions as being analogous to methods in a traditional object-oriented model but that analogy only goes so far. In the same way

that methods are defined as part of objects, actions belong to particular agents. (Though even more expressive ways of defining actions are contemplated in future releases.) In the next section we go into detail about what Actions are and how they can be used to define all agent behavior. They are also conceptually more challenging as unlike with structure they have no direct analogies to past agent representations.

An action provides simple, well-defined details that a model can use to determine what steps to take in model execution. That definition seems general enough to be almost useless, but its important to understand that an action is not equivalent to an instruction, a method or a query. In fact, actions can have aspects of all of these. But an action is not in itself an instruction specifying exactly *how* the modeling engine should do something -- instead an action represents *what* the modeler intends for the agents to do. (Technically, we might say that Actions takes a "declarative" approach instead of an 'imperative' approach, but that's not quite true, because actions do allow us to define behavior in a richer way than most declarative approaches, and many action constructs map directly to imperative approaches.)

Actions are connected together in a series of sources and targets. (Technically, an acyclic directed graph.) In an abstract sense that is similar to the way any programming language is defined although that structure isn't usually obvious because of the constraints of textual representations. But unlike a typical programming language, in Actions it is not the execution thread (the processor) that moves from one instruction to the next, but the result of the previous action. In this way, action results "flow" from the output of one action into the next action.

Why are these distinctions between traditional Object-Oriented and the Action approaches important? They give us the advantages of simplicity, clarity and flexibility that [data-flow approaches](#) like spreadsheets and some query languages have, but with less restrictions. At the same time, they can bring us much of the power and expressiveness of [functional languages](#) like Lisp or [logical languages](#) like Prolog, but without the level of complexity and obscurity that such languages can have.

We can get a better idea for how Actions work by thinking about how a spreadsheet works. In a spreadsheet, we might define a cell A that adds up a row of data, say "Income". We might define another cell C ("Profit") that takes A and adds it to another cell B that adds up another row of data ("Expenses"). Now, if we change a value in any of the rows, all of the other rows are added up and we get the results in A and B updated automatically. We never had to write code that said something like "for each cell in row X, where...". In fact, we don't really care *how* our Spreadsheet program adds up the numbers -- it could have added them all up at once but in backward order, or stored a running total somewhere and updated just the difference in value for the cell we changed -- what we care about is *what* the result is, and whether it is correct.

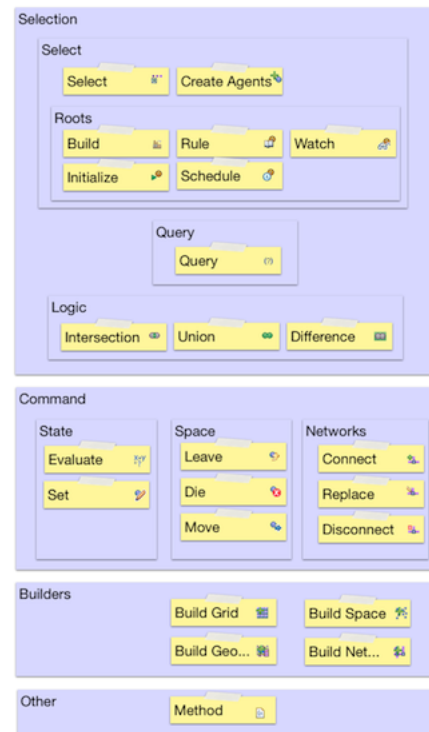
But Actions are much more powerful than a spreadsheet, because what is flowing from Action A to Action B is not just a number, but any model component such as a space or a set of agents that we need to use in target actions.

2.3.2. Concepts

In this section, we'll describe how modelers can assemble actions into sets of behavior that accomplish complex tasks on interrelated agents and spaces over time.

2.3.2.1. Kinds of Actions

Before getting into the details of how each Actions work together, or the various kinds of Actions, it will be helpful to take a broad overview of how they all fit together. As discussed above, actions are strung together in a sequence or flow. They're always composed of two parts, though those parts can be assembled and intermixed in many different ways. First, we search for a collection of agents, and then we do something with that selection. We refer to these two parts as Selections and Commands. (For the technically minded, another useful way of looking at the Actions approach is as a Query Transformation language, as with SQL and Stored Procedures. Except again, the results of the queries along with the transformations flow through from one query to the next.) Selections find the agents we want to do something with, and the commands do it. We need some way to start the whole chain of actions off, and so we have a kind of Selection called a Root Selection, or simply a Root. Secondly, we need some way to actually make the agents exist in the model in the first place, so we have a Create Agents action. Finally, we have special commands called builders that allow us to create the spaces that the agents will occupy. The various actions are discussed in depth in the Details section. The diagram at the right depicts how actions relate to one another -- it does not include some actions that have been added more recently.

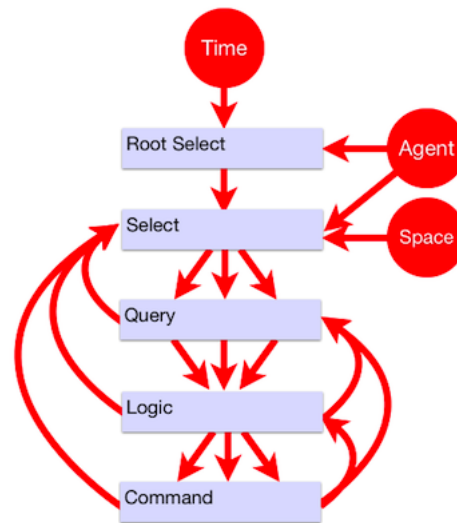


2.3.2.2. Flow

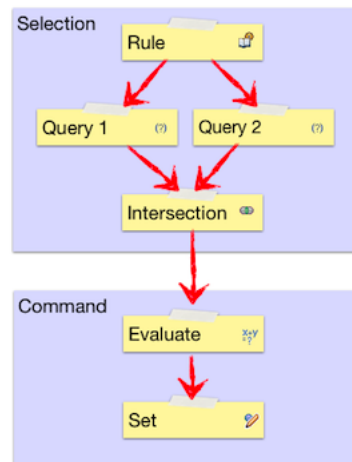
First, let's look at how actions define the basic path that agents take during a model run. As with any programming language, the path we take through the program specification is what determines our state when we get there. In a pure object oriented program, the path just defines the control flow -- what we are doing. The actual state of our model is defined within the object itself. If we call a method B from another method A, we'll be relying on method A to set the values that we need into the object state itself. In a purely functional program the path defines how we are going to deal with whatever has been explicitly passed in to a function that has been called, that is the function parameters. In fact, most languages such as Java combine aspects of both approaches.

In Actions, the path itself implicitly carries all of the context of prior execution with it. This means that we don't have to worry about storing context in the object -- as we would in an object-oriented language -- or passing the correct values from one method call to the next as we would in a functional language. Instead, Actions can use the implicit context of the path of flow to determine what the current state of execution is. An important aspect of the Actions design is that loop structures are not allowed -- that is, flows are acyclic. An action can never have an ancestor target (that is targets, targets of targets, etc..) that has as one of its ancestors that same action. As you'll see, actions don't typically *need* loop structures. By far the most common use of loops in conventional programming languages is to loop through collections of objects. As selections (see below) refer to the entire collection of agents, any actions on a selection apply to all members of that collection. Recursive structures are needed for some particular usages and will be supported in future releases, but not through an explicit looping construct.

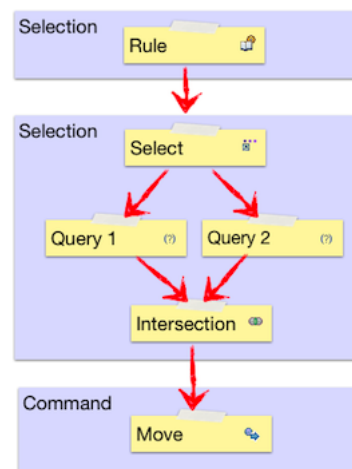
Again, behaviors in Actions are always defined by a set of *selections* and *queries*. In the diagram to the right, we can see the pattern. First, we define a Root Selection for a Rule, Schedule or other triggering event. Then, we might add a series of Query and Logic Actions to define the specific agents that we are interested in. These are all part of the Selection. Next, we might define a series of Commands to determine what to do with those agents. Or, we could use the result of that selection to immediately define another selection, for example if we are searching for an agent that is near another agent.



The diagram to the right depicts a simple example. Here, we create a rule, and then check the results of two queries. For any agents that meet those criteria, we'll evaluate some function based on their state, and then set some value on them.

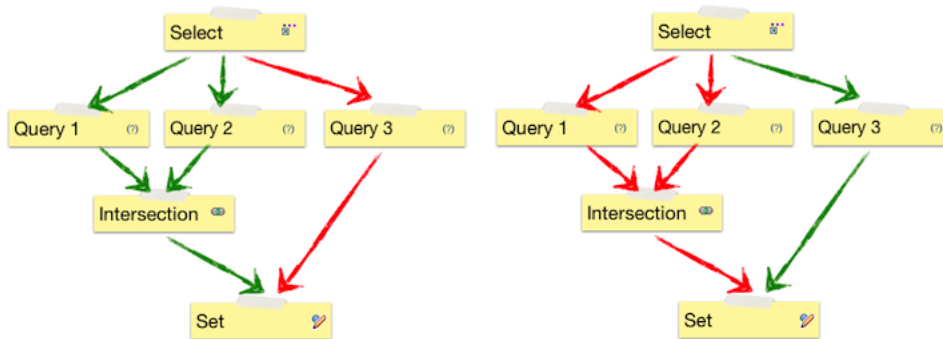


In this next example, we'll first create the rule, and then create a new selection with a set of criteria. Finally, we'll do a move based on those queries.

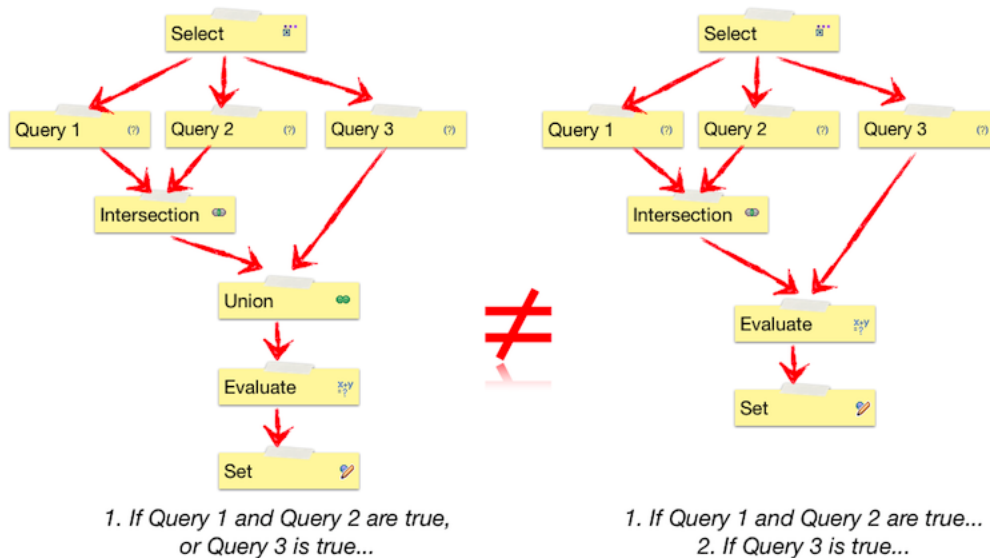


In the following example, we've defined a set of actions and their relationships. We have a selection, a few queries and a couple of logic operators leading to a Set Action. We'll describe in detail below how Logic

Actions are used in conjunction with other actions to assemble any kind of query structure needed. But for now, we'll focus on the control flow itself.



As you have probably already guessed, the agents that have the Set Action applied to them could take one of two paths through the Action flow. Readers with experience with programming or formal logic will note that this looks just like a parse tree, and while that's basically what it is, there are important differences. For example, if we looked at the following structure as a definition of control flow for a single agent we'd take them to be equivalent. Both would evaluate the statement (Query 1 AND Query 2) OR Query 3 for each agent.



Within Actions in many cases these two approaches will also act equivalently. If we are simply setting a value, it doesn't matter how an agent gets to that Set Action, as long as it gets there. All sources that flow into a given target Action act like a logical union since any of the paths might reach that target. But note that we have two flows moving in parallel in the flow on the right. What happens when the conditions for *both* branches are true? As the set of agents flow through each branch the Set Action on the left will be evaluated once, while the one on the right will be evaluated twice. Again, this often ends up with the same behavior, but not always. If for example, the evaluate Action uses the value of the attribute that we are setting as input, we can get different results. Of course, you can write code in any language that accomplishes the same thing, but the code will look quite different. For example, if we wrote the same basic logic in Java, in the first case we'd have something like:

```
if ((query1.evaluate() && query2.evaluate()) || query3.evaluate()) {
    doSomething();
}
```

In the second we'd have:

```

if (query1.evaluate() && query2.evaluate()) {
    doSomething();
}
if (query3.evaluate()) {
    doSomething();
}

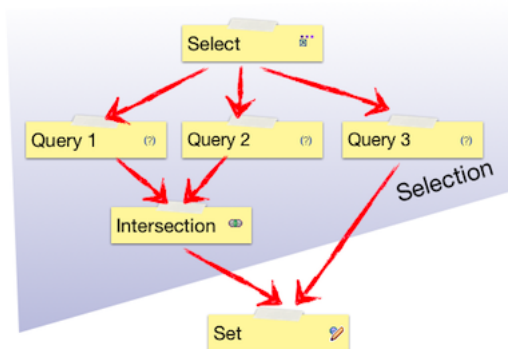
```

This is a simple example, but with multiple branches such code design issues can quickly grow complex. The flow approach allows us to express things in a way that is often more natural and expressive. The important thing to keep in mind when designing action flows is to see the flow as representing a selection of agents moving through streams independently. In the Actions example we expressed both approaches in nearly the same way, except in the case on the left we used a Union Action to bring the two branches of flow back together.

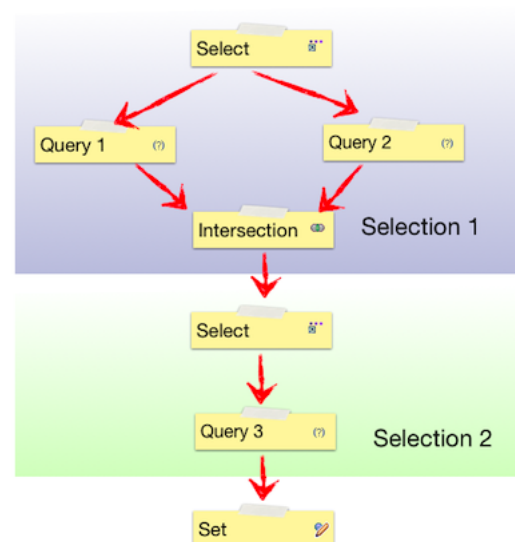
2.3.2.3. Selections

Selections are a key concept in Actions. Put simply, selections define what we are searching for and where. They are defined by a combination of Select, Query and Logic Actions. Each time we create a new Select Action, we define a new selection. Queries can be used to further refine selections either immediately after or later in the Action flow, as described in the next section. Logic Actions are used to combine and organize the Action flow defined by Query Actions. In order to understand how these three pieces work together, we need to understand the idea of selection boundaries.

A selection boundary determines the set of selection actions that are used to determine what agents to apply target actions to. For example, in the following diagram, we can see the extent of the boundary for a straightforward selection.



Each time we create a new selection, we define a new set of boundaries. In the diagram to the right, Selection 1 and Selection 2 each start with a new Select Action.

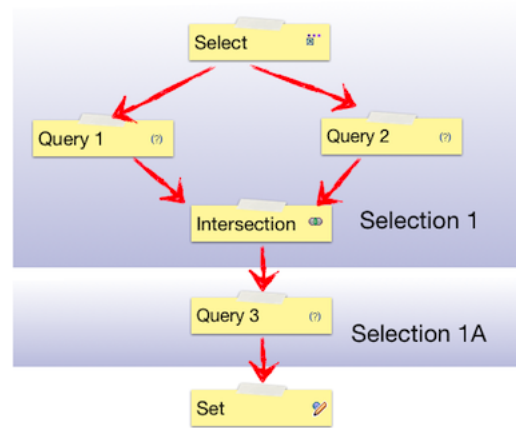


But boundaries can be defined for a group of actions by a Query Actions as well. This is because Query Actions can be directly part of a selection definition, but they can also refine selections. We'll see how that

works below. So where does one selection boundary end and the next one begin? The simple rule is that the end of the boundary is defined for a given Action by the place where:

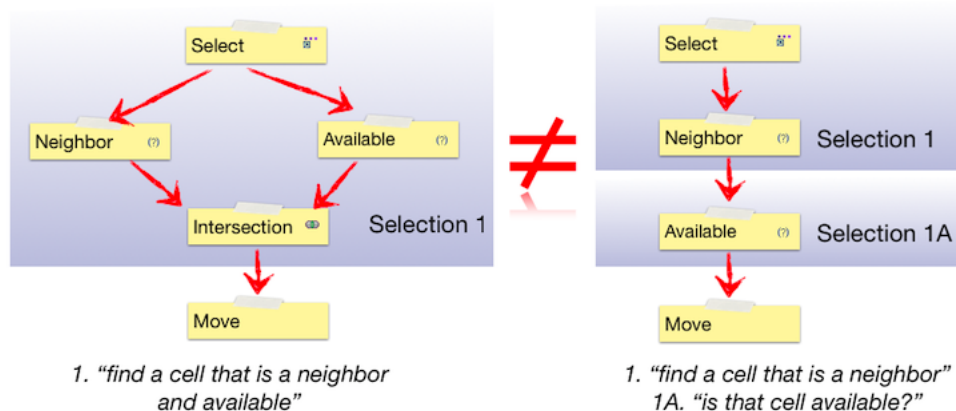
1. A Query Action is not followed by a Logic Action, or
2. A Logic Action is not followed by another Logic Action

In other words, as soon as a Logic Action occurs in a path leading to an Action, any following Query will define a new boundary, as shown in the example to the right.



Note that we refer to "Selection 1" and Selection 1A". This is because Selection 1A is a refinement of Selection 1 along its particular path of flow. When a query appears for the same selection but past a particular boundary, you can think of it as a sort of filter on the selection contents. We don't have a "Selection 2" here because any Actions that refer to "Selection 1" along the current path of flow will be acting on the selection defined by Selection 1 and Selection 1A.

These rules allow actions to be defined in the simplest possible way, but it is important to understand their implication as they result in behavior that can be different from what someone used to and imperative programming environment such as Java might expect. In a simple case the distinction might not matter. For example, if we are using a Query 1 to test whether an agent's attribute $a == x$ and attribute $b == y$, we would get the same outcome if we placed intersected the queries as if we simply put them in sequence. Internally we would actually be searching for agents with $a == x$, and *then* taking those agents and choosing those agents with $b == y$, but the outcome would be the same. But consider a more sophisticated case, where we are searching for neighboring available cells.

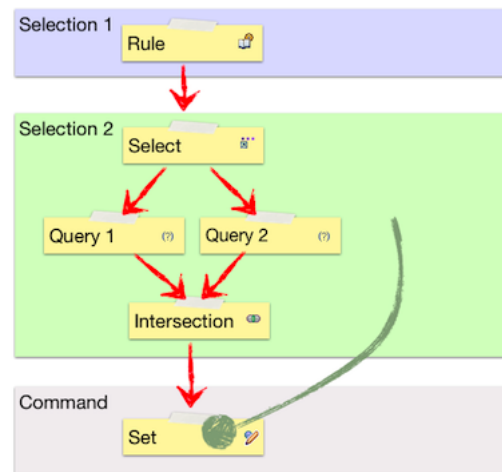


In the first case, we execute a search for all agents that meet the two criteria. This means that if there are *any* neighboring cells which are available, we're guaranteed to find one (random) cell. In the second case, we first search for all cells that are neighbors. This will match any agents that include both available and non available agents. Now, at this point since our search returns one agent (in the current AMF design -- richer

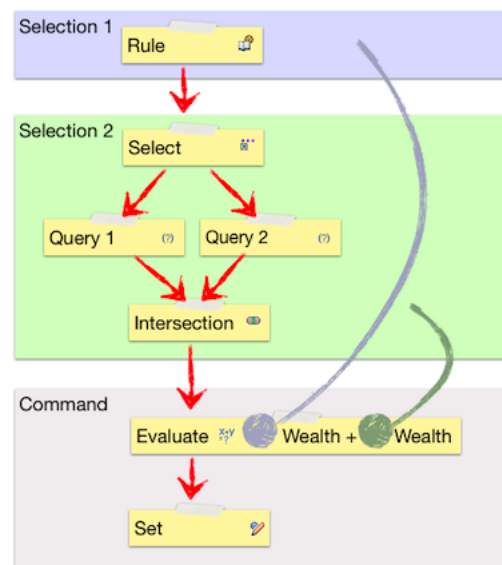
behavior will be supported in the future) the randomly selected agent could be either available or not. So in the second case, we might end up with no cell to move to, and thus make no move at all. This then becomes an important aspect of model design. For example, if one were defining a model where neighbors played a game with each other, one might want to instruct agents to play the game only with neighbors that have a certain wealth threshold. In the real-world situation that we are modeling, we might simply search for neighbors who are over a given wealth threshold and then play the game with them. This would imply that information about other agent's wealth is open knowledge. Or, we might simply select a random neighbor, and ask that neighbor to play a game with us. Upon discovering that our neighbor does not meet our wealth criteria, we would then choose not to play with them. Here we are modeling a cost in time to obtain information about another agent's wealth, because we might miss an opportunity to play the game with another agent on that round.

2.3.2.4. Weaving

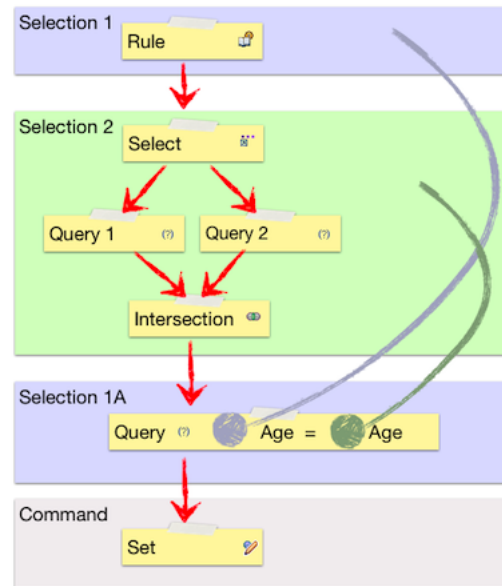
Now, let's put the concepts of Actions sequences and boundaries together to see how we can easily define complex interactions between multiple selections. When we define a Select, the state of its selection flows through and with any subsequent selections. So for example, if we have a Root Action rule, and then do a selection based on it, we'll have access to the agent from the original context as well as all of the subsequent selections. We can refer to any previous selection for any subsequent action. For example, instead of setting the value for the rule agent, we might instead set a value for an agent we've found in a target selection.



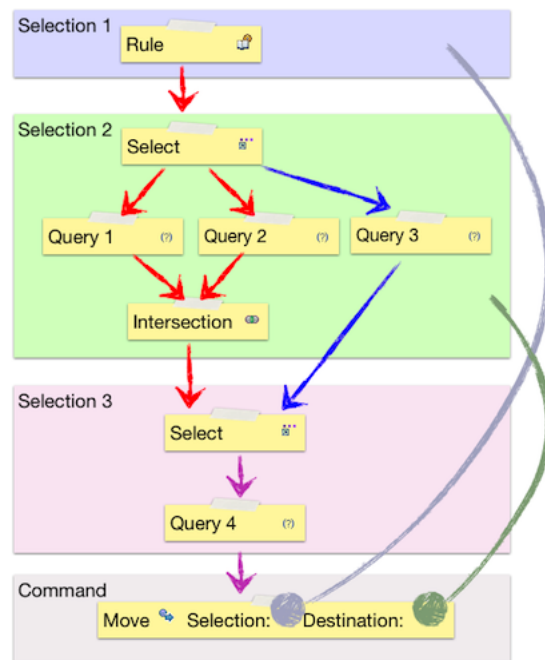
Inputs to functions also use selections. (We'll discuss more details in the functions section.) In the following example, we're adding the wealth of the Selection 1 agent to the wealth of the Selection 2 agent and using that value to set some other value. (Here, perhaps we are modeling an agent in a winner takes all game, in which case we'd also add a Set Action on Selection 2 and set the second agent's wealth to 0.)



But we can also use selections in defining Query Actions themselves. So in the following example, we select a neighbor agent and then compare the age of our Rule agent with the age of the Selection 2 agent. If and only if those ages are the same will we execute the target Set Action. This example also demonstrates why we refer to the data flow as weaving. Query Actions can be used to refine selections at any point in the data flow. Selections and their uses are interwoven throughout an action sequence.

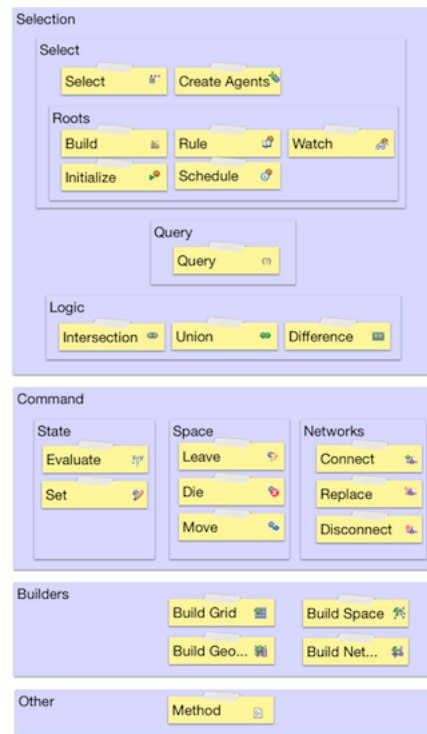


Finally, we can put all of these concepts together by weaving selections together with flows. As we discussed in the flow section, if we use multiple paths in the Query, the agents that flow through from any prior Query can follow multiple paths at once. And as we discussed in the selection section, the selection and its boundaries determine what agents we will be working with at any given evaluation point in the flow. Consider the example to the right. As we'll see in the detailed explanation of each Action below, Transformation Actions such as Move or Connect take multiple selections. The first selection defines the set of agents that will be performing the action. In the case of a Move agent, this refers to the mover. The second selection, which for Move we call "destination", refers to the selection that will be receiving the action. In the case of movement this is the agent or location that the Rule agent will be moving to. If we follow the flows through, we can note two important outcomes of our model design -- a Rule agent might move twice if it meets the criteria for both the blue path and the red path and that it might move to a different location each time.



2.3.3. Details

In this section, we'll dig into the specific role of each of the Actions. From the design discussion we hopefully have some sense of how these all fit together in general. Again, the block diagram to the right provides an overview of how the various actions are related, but it is missing some of the more recent actions such as Diffusion, Perform, Derive and Cause. You might want to take a look at the meta-class diagrams in the reference section as well.



2.3.3.1. Selections

A selection defines a particular set of agents that we want to do something with. Selections are made up of the Select action itself, along with Query and Logic actions. When we refer to a selection in any target command, we are referring to the selection in the context of where we have defined the behavior.

Select

As we discussed above, when we refer to a Select, we're actually referring to the selection as a whole leading up to the current action. The Select Action itself is used to define what we are searching for (Agent), where we are searching for it (Space), and from whose perspective we are doing it (Selection). Create Actions are a special kind of Select Action that are used to create agents. See the description in the Builders section below for more information.

Selection

The selection that we are searching "from". This seems to be the concept that new users have the most difficulty with, but it is key to understanding how Actions works. Just as with any other Action -- a Move command for example -- a selection needs to know what set of agents that it is working with. For example, if we want to define a selection B that finds the cells neighboring a rule B's agent, we would set selection B's selection to A.

Agent

Here we define what agent we will be looking for. In order for this to make sense, the agent has to be related in some meaningful way to the agent whose behavior we are defining. For example, it might be a partner agent or a location that we might want to move to. An agent must be specified unless we are searching within a continuous space, in which case this value should be empty, and the result of the selection will represent some location in that space. In the current version of the framework, we treat destination cell locations as

agents, and require that location to be specified as a target agent, but in a future version we'll allow searches without defining an agent within grid spaces as well.

Space

The space that we want to search within. Of course, this must be specified if we use any spatial query terms (see below), but if we simply want to search across all agents it should not be specified.

Represent Queries (Controls) on agents and Transformations, or Commands, on the result of those queries. Queries, Transformations and other in which each child target carries the execution and data context for it's particular path.

For

This value is obsolete and will be replaced with a different mechanism in the next version of the modeling environment.

Query

A Query represents a concrete criteria for our search. The name is a bit confusing because of potential for confusion with a generic query. Queries -- along with their cousin Evaluators -- define a function that is evaluated and that can take Agent attributes and the results of other Actions as input. Queries are combined with each other and with the logic actions to determine the results of a selection for their direct target actions.

Selection

As with all other actions, evaluations specify a selection, and just as with the other actions, this determines the set of agents that the evaluation occurs for, but the input selections determine what agent is used for the calculation itself.

Function

A query function is evaluated to determine the results of a particular selection. Functions can represent very simple search criteria such as "My Age == Your Age", but they can also represent complex and inter-related concepts such as spatial relationships. They must return logical values. See the functions section for more information on specific functions.

Inputs

The set of values that will be used to determine the result, in the order specified by the function prototype. Inputs can specify any source evaluation and any agent state or agent parent context state. They can also be literal values -- see the section on literals below. The selection determines which agent's will be used to determine the value, and different inputs can specify different selections.

Logic

These Actions provide us with the ability to combine queries with one another, and follow the basic rules of set logic. But as we've seen above, it is important to understand that there are important differences between Logic Actions and typical programming logic. Most importantly, they apply not to individual agents per se, but to the set of agents that move through them. Also, there is not necessarily short circuit execution (it's not needed) and much richer criteria can be joined together because of the action flow design.

Intersection

An intersection contains only those agents that match all of its source actions. This is essentially equivalent to a logical AND statement and has similarities to an && operator in a java "if" statement. An agent must be able to flow through all incoming actions in order to flow out of an Intersection Action.

Union

A union contains all agents that match any of its source actions. This shares similarities to a logical OR statement and the || operator in a java "if" statement. It does more than that however, as it acts to join multiple flows of action. That is, as set logic implies, an agent will never appear in the result of a union more than once.

Difference

A difference contains all agents that do not match any of its source actions. This is essentially equivalent to a logical NOT statement, and has similarities to the Java else statement. Like the Union Action, difference implies that a given agent will only appear once in any subsequent targets. No agents that reach a Difference Action will flow through to the next action(s), and all agents (that meet the definition of the Select Action) that cannot reach that action will.

2.3.3.2. Root Actions

Root actions are a special case of a selection. These represent behaviors that are defined for all members of agents; these are the highest granularity of an agent's behavior, such as "Find Partners" or "Eat" or "Reproduce". When you want to create a new set of Actions, you have the following choices.

Build

The Build Action is a specialized action that allows the construction of member agents and spaces within a parent context. A Build Action executes once for each context before any initialization actions occur for any children agents of that context. Currently it is undefined whether a context's own Initialize Action is executed before the Build Action occurs, so implementors should not rely on any initialized values being available at build time.

Initialize

An Initialize action is executed once and only once for every agent when the model is first started -- at time 0. Initialize Actions are guaranteed to execute before any other non-builder action occurs.

Rule

A Rule executes once for every agent for every iteration of the model. An important note is that the actual sequence of rules is technically undefined. An implementor should not rely on the fact that a rule occurs before another rule in the list of agent actions though typically the order in which the rules were actually created is respected.

Schedule

A schedule is executed on a recurring basis, according to the values detailed below. Note that schedules are often overused. In most agent-based models it makes sense to have any behaviors occur at the same granularity using a Rule. Please note that Schedules are not currently supported in the Escap target, but that support should be available soon. In the following descriptions we refer to period as the current iteration of the model, that is where `time == t`.

Start

The period that the schedule will first be invoked. For example, if this value is 100, and interval is 1, the schedule will be executed at times 100,101,102..

Interval

How often the schedule is invoked. For example, if this value is 3 and start is 1, the schedule will be executed at times 1,4,7..

Priority

Where the rule will be placed in the execution queue for a given time period. For example, if Schedule A's priority is set to 3 and Schedule B's is set to 2, Schedule B will be executed for all agents before Schedule A is executed for any agents.

Pick

Controls how many agents to execute the schedule against. While this value will have an effect on the Repast target, it is not recommended to be used for general models and is likely to be replaced by another approach.

Watch

A Watch is executed any time the watched value is set for any agent. Note that the Action will be triggered even if the state is simply set back to the value that it already has. It is important to be careful about creating Watches that might set the values of other Watch actions which might in turn trigger this watch. To clarify, if a modeler creates a Watch A for attribute a, and creates a target Set for it for attribute b, and Watch B is watching attribute b, then if Watch B has a target Set for attribute A, a circular execution could occur. This would cause the model to get stuck in its current iteration. To help model developers avoid this case, a warning will be provided if such a set of circular watch dependencies is created.

Attribute

The attribute that will be monitored for change.

Derive

A Derive action is a unique kind of root that is used to determine that value of a derived attribute. There can be one and only one Derive action for each such attribute. One of the benefits of a derived action is that unlike a standard attribute, it only needs to be calculated when the value is actually needed and its inputs have also been changes -- allowing significant performance optimizations to be made. Derived actions can also make model behavior much more clear. Derive actions are especially useful for calculating dependent measures for model agents. These in turn can be used directly in charting and data output tools -- leaving no need to configure your chart view calculations separately. Developers need not worry about overhead and should feel free to create as many derived attributes as might be necessary. The "gather data" value of the derived attribute can always be set to false to prevent data collection overhead when they aren't needed.

The derived value will always simply be the value of the last evaluation in a particular flow. You can mix queries and even selections on separate agents into a derived value. The only restriction is that the type of the last Evaluate action(s) must match the type of the value. If there is no path leading to an Evaluate action for a particular agent state, the attribute's default value will be used.

Attribute

The attribute that will be derived.

Diffuse

The diffuse action provides high level support for the common behavior of diffusing some value across some lattice space. For example, heat may spread over time from one grid cell to the next. (There are actually significant issues involved in implementing this through lower level actions.) To specify that a value should diffuse through a space you simply need to provide the following values. This action does not need and shouldn't include target actions. The "Heatbugs" model in the `org.eclipse.amp.amf.examples.escape` project provides a good example for how diffusion can be easily implemented into any grid model.

Diffused

The attribute whose value is to be diffused.

Diffusion Rate=

The rate at which any given cell's attribute value is transferred to surrounding cells.

Evaporation Rate

An optional rate by which each cell's value is reduced for each period. This is useful for a model where agents are creating energy within the environment.

Perform

A Perform root action simply defines a set of actions that have no independent trigger. The only way Perform actions can occur is if a Cause action specifies one as a result. Performs then are similar to sub-calls or helper methods within a traditional procedural or OO language, and can be very helpful in organizing and simplifying

code. Note that whenever you use a Perform instead of directly specifying a set of targets you lose the context of the selection. You couldn't use a Perform to directly trigger a move as the selection source for the move would not be available.

2.3.3.3. Builders

Builders are a special category of actions that are used to create spaces. They should not be confused with the Build Action itself which is a root selection that defines the time at which builder actions should occur. Generally speaking, specific Builder Actions for spaces and Create Actions for agents are targets of Build Actions.

Create Agents

Agents are created using the Create Agent action. The Create Agent Action is actually a special kind of Select Action that is used to actually create agents rather than simply search for them. Other than this capability, a Create Agent action can be used just like any other action except that Query and Logic actions aren't needed or generally appropriate as this action defines a set number of agents to perform an action against. Create Agent actions have the special feature of being usable from the containing context, so that contexts can create initial agent populations before any agents exist within the context to perform actions against. But they can also be used directly within agent rules, for example to create a child agent as the result of a reproduction rule. Note that Initialize Actions are *not* performed on agents that have been created within a model during regular (non-initialization time) execution. If the model creates actions during the regular model run, any state initialization should be handled by the targets of this action directly. (Naturally, if the enclosing context is used to create agents at the beginning of a model run, then the Initialize action is used as part of the normal model life-cycle.)

Agent

The kind of agent to create.

Selection

The selection to use as the basis for this selection. This is generally not important except to define control flow.

Space

Not generally relevant for agent actions, as agents are not placed into a space unless explicitly moved to that space. Potential associations between agents and spaces are defined by the space building actions.

Agent Count

The number of agents to create. If used as part of an enclosing context's Initialize Action(s) an Attribute parameter will automatically be created. If not, then an evaluation or agent attribute should be used to define this number.

For

Deprecated. Should not be used.

Create Shaped Agent

Creates an agent that has a particular shape extent within some continuous (i.e. vector as opposed to raster) space. This action performs just like a Create Action except for the following attribute:

Shape

Determines how each agent's spatial extent will be defined. The actual form of that shape is implementation specific. For example, in a GIS space (the immediate rationale for this Action) a polygon represents a region on a map and the extent of that shape might be determined by a .shape file. The available shape types are:

- | | |
|-------|---|
| Point | A simple point in space, fully equivalent to a standard continuous space location. |
| Line | A single line within a continuous space. Technically this represents a line segment as it is expected to have a beginning and ending point. In the future, this might refer more generically to |

	planes in three dimensional space and hypersurfaces in n-dimensional spaces, but this is currently not supported by any AMF target implementations.
Polygon	A region within a space defined by an arbitrarily large set of line segments. Potentially this could be used to refer to polyhedrons in three-dimensional space, or even more generally as polytopes, but this is currently not supported by any AMF target implementations.

Load Agents

Imports and creates a set of agents from some input file. The actual form and manner of the import is implementation specific but should be inferrable from any source file based on the URL provided, assuming that the target platform or the AMF platform supports the appropriate input type. For example, a Tab-CR delimited file might be used to populate a set of agents with various attributes. Additional meta-data could be provided by the URL, but we will likely add additional extensible meta-data to this action to better support the definition of expected input types and routines from within an AMF model itself. This action is equivalent to the Create Agents action except for:

Source URL

The location of the input file or set of meta-data used to determine the location and type of the input file.

Load Shaped Agents

Combines the properties of the Create Shaped Agent and Load Agent Actions. The source url should of course specify an input format that supports the shape type specified. See the descriptions of the related actions for more details.

Build Network

Creates a network, i.e. a graph structure, supporting the establishment of edges between arbitrary nodes.

Agents

Specifies the set of agents that can exist within this network. Agents must be included here in order for them to make connections within the network but agents included here are not required to have connections within the network, nor are such connections created for them. (See important note regarding network types below.)

Attributes

Not currently used for networks.

Network Type

Deprecated. This feature is only currently supported for Repast targets and is likely to be removed from future versions of the AMF meta-model. Future AMF implementations are likely will provide a different mechanism for instantiating and importing network structures either within the network definition or through other Action definitions. Instead of using this feature, modelers should create specific networks by building them up with Connect Actions for individual agents. For example to create a small world network, a modeler might create random links between agents and then replace or augment those connections.

Selection

Not relevant for network builders except as part of normal control flow.

Build Grid

Creates a grid space.

Agents

The set of agents that might exist as occupants of this grid; that is, members of individual cells within a given grid. Agents must be included here in order for instances to exist within the space, but agents included here do not actually have to exist within the space. (In the Repast implementation, all agents technically are members

of the spatial projection, but are not required to have valid coordinates within that space.) For example, in an agriculture model these might represent agents moving about and harvesting plots of land.

Fill Agent

The agent that will be used to populate the grid itself. A grid is guaranteed to contain one and only one fill agent within each grid location. The grid will be populated with instances of the specified agent and these agents cannot move, leave or die within this space. This value need not be specified -- if left blank a default cell without any state will be used. For example, in an agriculture model, this agent might represent a single plot of land.

Space Type

Deprecated. Should not be used.

Selection

Not relevant for builders except as part of normal control flow.

Attributes

Not currently used for spaces.

Build Space

Creates a continuous space. The actual dimensionality and other qualities of the space are currently defined in the space itself though this might change in future versions. All other values are used in the same way as for the grid and other builder actions.

Agents

The set of agents that might be a part of this space. Agents must be included here in order for instances to exist within the space, but agents included here do not actually have to exist within the space. (In the Repast implementation, all agents technically are members of the spatial projection, but are not required to have valid coordinates within that space.)

Build Geography

Constructs a geographical space. All details of this space are specified by the implementation, i.e. a specific geographical imported space. Generally these would be defined by a Create Agents action; that is a set of imported agents representing US states would also represent the overall space of interest.

2.3.3.4. Commands

Evaluate

Evaluate Actions define some calculation on a function based on the model state and a set of input(s). The inputs that an Evaluate Action can take is determined by its functions and can be either agent attributes, prior evaluations or literals. The result is then determined based on those inputs. In some cases Evaluate functions can be used to determine some action indirectly, such as with a graphics fill, but they can never be used to directly change model state.

Selection

As with all other actions, evaluations specify a selection, and just as with the other actions, this determines the set of agents that the evaluation occurs for, but the input selections determine what agent is used for the calculation itself.

Function

As with queries, a function is evaluated against its input set. Functions can represent simple operators as well as complex functions. See the functions section for more information on specific functions.

Inputs

The set of values that will be used to determine the result, in the order of the function prototype. Inputs can specify any source evaluation and any agent state or agent parent context state. They can also be literal values -- see the discussion in the Tools section. The selection determines which agent's will be used to determine the value, and different inputs can specify different selections.

Set

The Set Action assigns some value to another value.

Selection

Here the selection refers to the agent that we want to change. This does not have to be the immediately preceding selection but can be any accessible selection.

Attribute

The attribute to modify. It must be a member of this action's agent or of that agent's parent context.

Parameter

The value to assign to the attribute. Here, we can use either another agent attribute, or the results of a source evaluation.

Cause

A Cause Action "causes" some Root Action to occur upon the specified selection. This action can be extremely useful for organizing model behavior and preventing the need for creating duplicate action definitions.

Cause actions also support recursive functionality and "WHILE" behavior. These can be used to mimic loop structures. For example, you might want an agent to execute some behavior as long as that agent has energy remaining. To accomplish this, you could create a query for "energy > 0" and then create a Cause target with the root action as its results.

Note that you should be cautious and thoughtful when using the Cause action. Remember that selections represent sets of agents and thus get rid of the need for collection loop structures -- such as Java "for each" -- common in traditional programming languages. You should be able to do almost anything that might require a loop structure using the selection mechanism itself. Also, just as with a traditional language, you should be careful about defining cause actions that trigger their own root actions or that cause other actions to in turn trigger the original root action. You can easily end up defining infinite loops in this way. (In the current implementation this will eventually trigger a stack overflow error as cause invocations are recursive, but future implementations will be able to infer target language loop structures to prevent stack depth issues.)

Result

The root action that should be triggered for every member of the current selection. This is typically a Perform action but it can be any kind of root action except for "Derived". (Which doesn't make any sense to do.)

Move

The Move Action causes an agent to change its location in some space or network. The agent will leave whatever location it was in before within the selection space, and move to its new destination.

Selection

As in any other action, the selection determines what agent is affected -- in this case the agent that is being moved.

Destination

Specifies the target agent or location for the movement.

Leave

Causes the agent to leave a particular space.

Selection

The selection determines what agent will be leaving and what space the agent will be leaving. If the agent doesn't exist in that space nothing will happen.

Destination

The destination is irrelevant for a leave action and should not be specified.

Die

Causes the agent to cease to exist within the model as a whole.

Selection

The selection determines what space the agent to remove.

Destination

The destination is irrelevant in this case and will probably be removed.

Connect

Connects two agents within a network space. This Action is not applicable for any other kinds of spaces. Note that unlike with other transformational commands, we do not use the destination space to determine the space that will be impacted by the Action. This provides a more efficient representation without any loss in generality, because it allows us to search for a source and target agent within other spaces and then create a connection without creating a separate selection. As the important structural feature of networks are the relationships themselves, not the nodes this provides a more direct way to specify these relationships.

Selection

The selection determines the agent that will be connected to another agent. In the case of a directed graph, this is the source node.

=Destination

The destination determines the agent that the selection agent will be connected to. In the case of a directed graph, this is the target node.

Within

Specifies the network that the connection will be created within.

Directed

Determines whether the connection made is directed or not. If true, selections from source agents will include the target, but target agent selections will not include the source agents (unless they are connected through some other path).

Disconnect

Removes the connection between agents within a network space. See the description of the Connect Action for important details.

Selection

The selection determines one side of the agent relationship that will be disconnected.

Destination

The selection determines one other side of the agent relationship that will be disconnected.

Within

Specifies the network that the connection will be created within.

Replace

Functions in the same way as a Connect Action except that all other connections to other agents will first be removed.

Selection

The selection determines the agent that will be connected to another agent. In the case of a directed graph, this is the source node.

Destination

The destination determines the agent that the selection agent will be connected to. In the case of a directed graph, this is the target node.

Within

Specifies the network that the connection will be created within.

Directed

Determines whether the connection made is directed or not. See the Connect description for more details.

2.3.3.5. Other**Method**

The Method action supports inclusion of arbitrary code within a generated method. Generally, this will be Java code as all of the current target platforms are Java-based but there is no technical requirement that it must be. For example, if a target has been developed to produce code for Swarm running on an iPad (and no, there are no current plans to support such a thing, though it would certainly be cool!) then the modeler could define Objective C code for the method.

Please note that the Method Action should be avoided whenever possible. You should consider using it only in the case where there doesn't appear to be a way to construct equivalent functionality using the native Actions framework, such as when interfacing with third party APIs. The aim of Actions is to provide the most general support for Agent Modeling possible without compromising the core design. Any use of native Java code strongly limits the set of platforms that your model will be able to generate code for and prevents you from using the AMF edit tools to explore the model behavior. In the case where you wish to construct a model feature and believe that it isn't possible or practical to do it with Actions, please contact us (see support section) so that we can suggest how you can accomplish what you want within Actions. If that is not possible, we'll consider developing new features that will support what you want to do.

On the other hand, you may simply wish to use the Agent Modeling Framework to build a scaffolding for your model -- perhaps using your own custom Java framework for example -- and Method would be a good way to accomplish that. Also, note there are a number of other approaches to mixing hand-crafted Java together with AMF generated code. Please see the Programmer's Guide section "Integrating Java and AMF Models" for more on that.

If you do decide to use the Method Action, keep in mind the following design practice recommendations:

1. Keep your usage of external API references to a minimum. If you can use only code provided by the core Java classes as well as the Apache Collections library, your code should work on every current Java target. On the other hand, if you make use of a specific ABM platform APIs your code will obviously only compile and run against that target.
2. Code should be in the form of a method body, excluding the signature. A single Java method is created using this code body. There is no support for input parameters -- if you need access to evaluated values

from source actions, create agent attributes for them, set their values for the selected agents, and use them as sources for your Method Action.

3. All Java class references should be fully qualified. For example, if you wish to use the eclipse Draw2D Graphics class, you should refer to "org.eclipse.draw2d.Graphics", not simply "Graphics". If classes are not fully qualified, you will receive compile errors. You can usually easily fix these by selecting your source code directory and choosing **Source > Organize Imports..** but it prevents automatic code generation.
4. The method interface has no support for code completion, syntax checking or other common Java development environment features. You can avoid code maintenance and even support targetting multiple APIs by using the following technique. From your method body, call a helper method in a separate class. The referred class could use a static method call, or you could instantiate the class and call a method against it, passing in the agent class so that the helper class can reference the agent's state. For example, if you wanted to use some custom java code to import agents from a specialized input file, you could put the following code in the Method Action body for the root Context:

```
(new org.me.SpecialFileLoader()).load(this);
```

Then create a new source directory in your project called "src" (**New > Source Folder...**) and create the class for the specialized file loader including the following method:

```
public void load(MyRootModel model) {...}
```

This approach allows you to a) maintain the working code using the Java Development Environment, b) avoid changes to the Method Action within the model, and most importantly, c) allow other implementations of the code using multiple APIs. For example, if you need to create a specialized graphics routine, you could create separate implementations for your Escape (Eclipse Draw 2D), Ascape (Java Swing), and Repast (Simphony API) and place them in the appropriate projects. As long as the different Java source files have the same names and signatures, they will all compile correctly and execute the appropriate behavior.

Selection

The selection determines what Agent class the code will be created within and the set of agents the method will be called upon.

Body

The actual code to insert in the method body. See the detailed recommendations for code use above.

Generate

Determines whether the code is actually inserted. If this is false, a bare method body will be constructed instead. This can be useful if you wish to turn off method generation in certain model implementations without removing the actual code.

2.3.3.6. Query and Evaluation Inputs

Query and Evaluation Actions are both "Sinks" which means that they are capable of containing inputs. When you select a function, the appropriate number of inputs will be created. After selecting a function, you can view and select the inputs. The choices for the inputs will be constrained by the type of the function and the other operands you've selected.

Input Literals

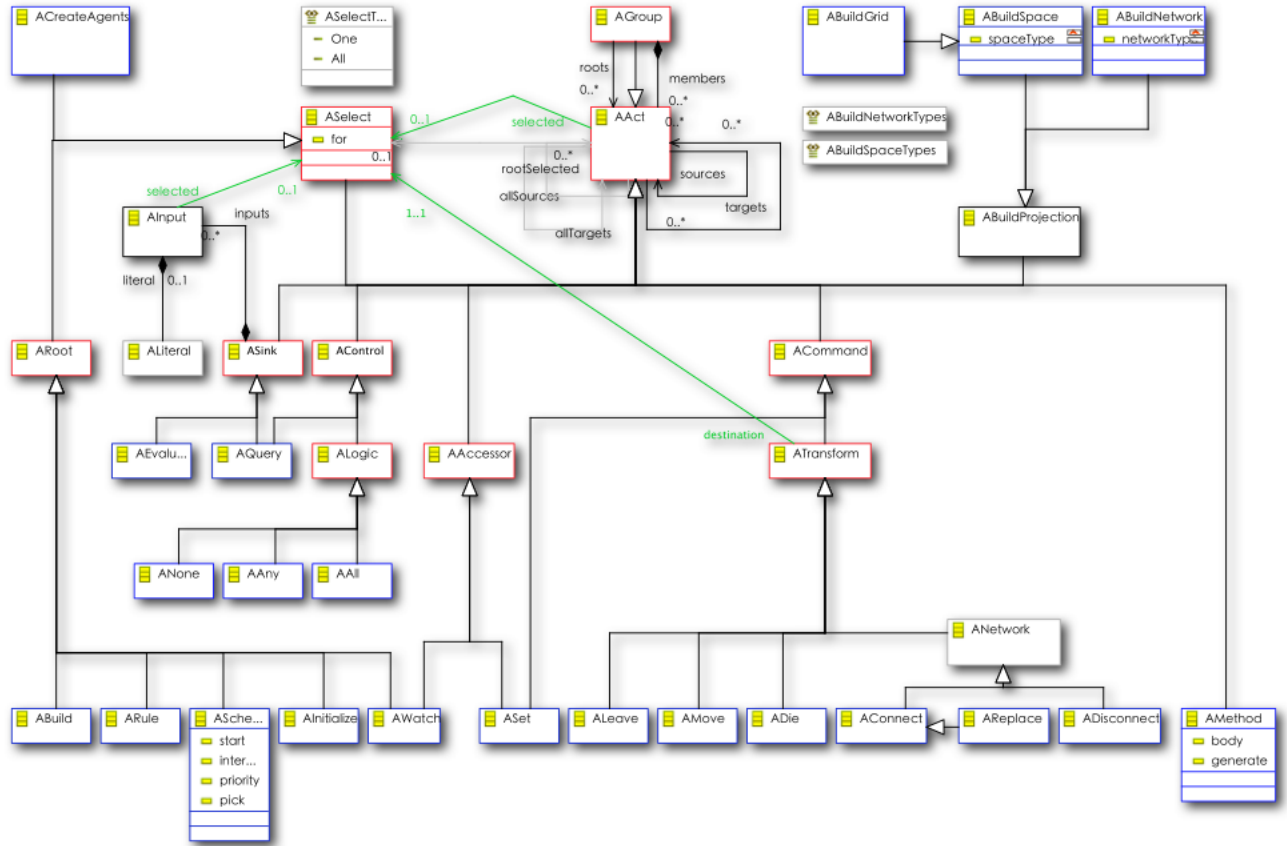
Inputs can take literal values; that is values that you specify simply by entering them directly into the query. In general it is useful to think of literals as similar to local variables in a conventional programming language, whereas attributes are analogous to member variables. (And this is how they are represented in the generated Java code.) As with local variables in model code, literals are not recommended for any values that can change model behavior. The value cannot be easily accessed or changed by other model users. For greater transparency, you should instead create an Attribute with an appropriate default value, setting the "immutable" value to true. Still, literals can be useful for values that are special cases related to the evaluation or query, such as an input code, and for quickly prototyping functionality.

2.3.4. Reference

2.3.4.1. Diagrams

The following diagram may be helpful to readers familiar with UML and Meta-modeling:

Meta-Classes



Details

In the diagram above, all meta-objects except for Input, Literal, and the enumerators (lists of options) are Actions. Blue meta-classes are concrete (you can create and use them directly). Red meta-classes are key collaborations.

1. An Act is anything that might happen during the execution of an Agent-Based Model.
2. All Actions have as their root-most source action a Root. These are added first to any agent behavior and act as triggers for all target behavior. For example, a Watch will execute any time the watched attribute is modified. (As the diagrams do not refer to elements outside of the current package, we cannot see here that Accessor includes a reference to some Attribute, but it does. To see these kinds of relationships you will want to refer to the metaabm.ecore file itself.)#Actions are targets and sources of one another, but an Act can never have itself as a source. (That is, Actions are acyclic, but branches can re-converge. When we refer to an Act source or target, we typically mean to include all ancestors or descendants, not just the immediately connected Act.)
3. All Actions (except for root Actions) reference a Select, referred to as the "selected" relation. An ASelect represents the model aspects that the Act is working within; that is, the spatial, temporal and type (agent) "world" that is currently being selected.
4. Commands trigger some model state change (Set) or spatial transformation (Transform).

5. Controls determine whether target Actions are executed and against what agents. They are in some sense query terms and include Query actions and Logic Actions.
6. Transforms also specify a "destination" Select. This represents aspects that the selected agent(s) will transform to. For example, a Move may use a Rule to select all SugarAgents (type) in the SugarGrid (space) every period (time) and move them to a destination of a neighboring SugarCell (type) in the SugarGrid (space, with time implicit).
7. Sinks are Actions which use some Function (see next section) to interpret state in the form of Inputs. Inputs can come from selected agent attributes, other Actions, or literal values.

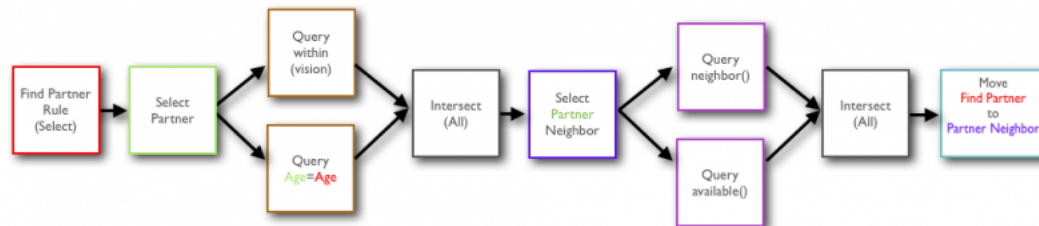
2.3.5. Example

In this section, we'll look at an example that should make clear how the basic Actions approach works in a model. Say we want to define a behavior like:

"Search for a random agent within my vision that is the same age as I am. Find a location next to that agent that is not already occupied and Here, we create a sequence of actions like so:

1. Select every agent for every period of the model. ("Find Partner" Rule)
2. For every member of that selection, search for other agents of the same age within vision distance. ("Partner" selection.)
3. From "Partners" find a random member and search for a neighboring locations. ("Partner Neighbor" selection.)
4. Finally, move the agent in "Find Partner" to the "Partner Neighbor" location.

Now, notice that although it's convenient to speak as if there is only one "Find Partner" and one "Partner Neighbor" in step 4 above, in fact selections are flowing through for each of the results of each of the previous action sequences, and we can refer to each of the directly. We could represent these behaviors in many different ways. For example, we might want to specify the model in a (hand-drawn) graphical language or in a (made-up) textual language:



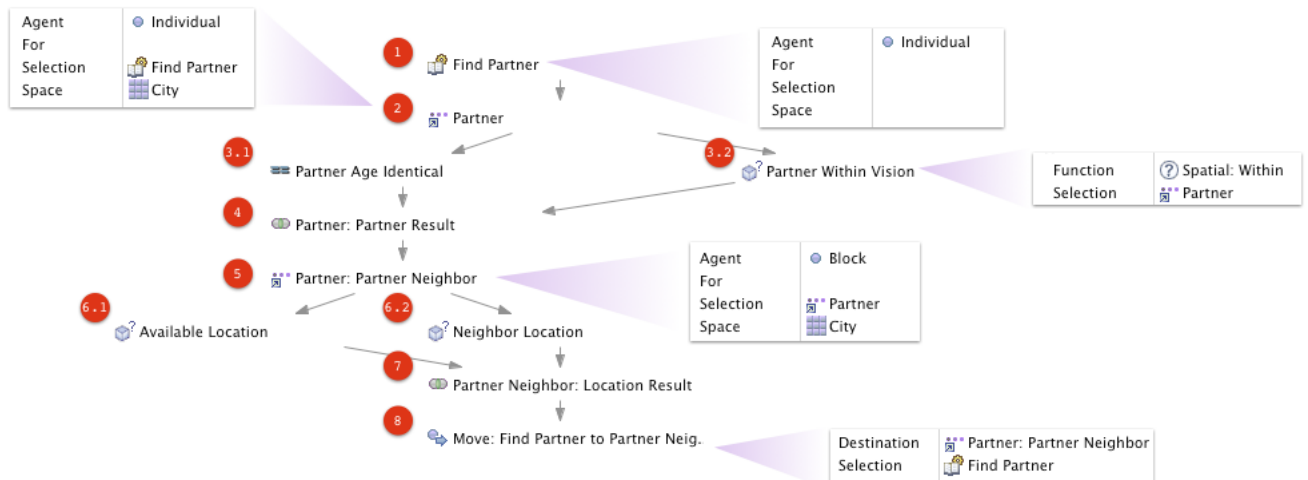
"Find Partner": RULE (AGENT: Individual, SPACE: City)

"Partner": SELECT (Agent: Individual, SPACE: City) Find Partner WHERE within(vision) AND Age = Partner.Age

"Partner Neighbor": SELECT (Agent: Block, SPACE: City) Partner WHERE neighbor() and available()

"Move to Partner": MOVE Find Partner to Partner Neighbor

This is how it looks in an actual model:



And here is how this works in detail:

1. Create a Rule that will trigger the behavior. In this case, we want this rule to apply to all "Individual" agents within the model. (Space isn't relevant in this case).
2. Create a child Select Action that will find our partner. Two important things to note here:
 - a. The selection occurs based on the "Find Partner" selection. This means that for each Individual in the model, we'll be searching from the point of view of that agent to some other selection of agents.
 - b. We also need to define what type of agent we want and in this case the space does matter. We want to find an agent that is nearby within the City space. If instead we wanted to find a partner in a social network, we'd specify that instead.
3. Create two child Query Actions:
 - a. We want to search for someone who is the same age as us. This highlights the importance of the idea of the Selection in the Actions design. We're qualifying Age by the rule agent's partner for the first input and by the rule agent for the second. The selection carries throughout the flow of execution and this context is an explicit part of the entire structure. Note that this is very different from the way control flow works in a traditional imperative language such as Java.
 - b. We also want to search using a function for nearness, "within", which takes a parameter of vision. Note that the spatial functions are all polymorphic -- if we decided later on that we would rather search within say "Kevin Bacon space", that is a graph structure representation of space, we would only need to change the space we've defined in Select Partner.
4. Intersect the results of these two query components. This delineates the end of the selection definition for any target Actions.
5. Select a neighbor. Again, we can see the importance of Selections. Here, we are selecting from the point of view of the partner, not the initial agent that the current Rule is being executed for. Note that this time our target agent is a "Block", that is, a location within the city.
6. As above, define some queries. This time we want only those agents that are:
 - a. available, and
 - b. neighbors of our partner.
7. And another intersection..
8. Finally, we move to the location we've found. All that's required at this point is to specify:
 - a. The movement selection, or those agents that are moving, which in this case is the original agent we're executing the rule for, and

- b. The destination, which is the cell that we've found. Note that the framework infers from the space definition that the Block agent is capable of hosting the Individual.

2.4. Functions

2.4.1. Overview

Functions are relatively simple in terms of model design, but we need to understand how particular functions work in order to develop models. Functions are divided in two ways. By type:

1. Operators are simple calculations sharing the same type.
2. Functions that can represent any general function that takes some well-defined input(s) and returns some well-defined output(s).

And by usage:

1. Generics can return any value and are used in Evaluate actions.
2. Logicals return some boolean result and are used by Query actions to decide whether target Actions apply to a particular selection, and by Evaluate actions just as with any other functions. Input types should be defined as generally as possible.

These overlap, so we have operators, logical operators, functions and logical functions. Functions are also divided into categories. We'll go through each of these in depth below.

The most important thing to point out about functions is that - as we've seen with other Acore concepts -- they provide richer sets of functionality than traditional approaches. Many functions are designed to collaborate with one another as we'll see when looking at Spatial and Graphical functions. Functions can also trigger the creation of related model artifacts as we'll see with the Distribution functions.

A technical note: conceptually, functions can return multi-values, but that is not currently implemented in the reference targets because of limitations of the target language Java.

2.4.2. Details

2.4.2.1. General Functions

Naturally, the Modeling tools provide general functions for performing calculations, comparisons and more complex mathematical expressions. The function library can be easily extended, and we'll be adding additional capabilities over time. As always, we welcome feedback and we'd like to hear what functions you need that aren't covered here.

2.4.2.2. Logical Operators

Logical operators allow comparison between values. They are typically used in Query terms but may be used in Evaluate actions as well.

Not

The result of the expression !X.

Equal

The result of the expression X==Y.

Identical

The result of the expression X==Y.

Greater

The result of the expression X>Y.

Lesser

The result of the expression $X < Y$.

Greater or Equal

The result of the expression $X \geq Y$.

Lesser or Equal

The result of the expression $X \leq Y$.

True

The result of the expression true.

False

The result of the expression false.

Identity

The result of the expression X .

Different

The result of the expression $X \neq Y$.

2.4.2.3. Numeric Operators

Numeric operators support all of the basic operators used in evaluations.

Negative Value

The result of the expression $-X$.

Add

The result of the expression $X + Y$.

Subtract

The result of the expression $X - Y$.

Multiply

The result of the expression $X * Y$.

Divide

The result of the expression X / Y .

Power

The result of the expression X^Y .

Modulo

The result of the expression $X \% Y$.

Increment

The result of the expression $++X$.

Decrement

The result of the expression --X.

Unit Value

The result of the expression 1.

Zero Value

The result of the expression 0.

Original Value

The result of the expression o.

2.4.2.4. Spatial

Spatial functions provide the core functionality for Agent Models. Spatial functions are polymorphic, which basically means that they don't care what space they are operating on as long as that space is suitable for them. Spatial functions are designed to collaborate with one another. For example, by intersecting the "Neighbor", "Available" and "Toward" functions, we can design a rule that causes the agent to move to the next neighboring cell that get's it closer to some target agent. See the function details for more information.

Nearest

Represents the nearest agents (including grid cells) or locations to this agent. If more than one agent or location is the same distance away they will all be considered. Note that while this function is defined for the selection of an agent, the result of this function is defined by the context within which it is used. If the selection specifies another agent within a space, this function will represent the nearest agent in that space. If the selection specifies a Cell within a grid space, this function will represent that cell.

Toward

Represents a location that is on the shortest path to a particular agent or location from the source agent (that is, the selection's selection's agent). This function collaborates with the within and neighbor functions to allow the agent to move in a particular direction towards some objective.

Within

Represents a limit to the distance of a spatial search. When used in combination with other spatial functions such as "nearest" requires that all agents or locations must be within the distance specified by the input value.

Inputs:
[Numeral]

Neighbor

Represents any agents that are nearest neighbors to the agent, that is nominally of distance 1. This function is only relevant in discrete spaces -- grids and networks -- where there are immediate neighboring cells as defined by the geometry of the selection's space.

Include Self

Specifies whether the agent that we are searching from -- that is, the agent of the selection for this Query Action's selection -- is included in the results of the search.

Within 2D Boundary

Represents agents or locations that exist within the boundary specified by the inputs.

Inputs:
[Numeral]

Here

Represents the location of the searching agent. For example, if a selection is defined for an agent cell, and that selection's selection's agent is an occupant of a cell, the cell that the agent is occupying will be used.

Available

Represents cells which are not currently occupied. This function is only relevant for grids which are not multi-occupant.

Distance

The distance between the source agent and an agent represented by this selection. If more than one agent is represented by the other functions in the selection, this function will the distance to an arbitrary (randomly selected) agent as defined by those other functions.

Outputs:
[Real]

Away

Represents a location that is on the path that will take the source agent (that is, the selection's selection's agent) the farthest distance from the agent(s) represented by the search. This function collaborates with the within and neighbor functions to allow the agent to move in a particular direction away from some location or agent.

Minimize

Finds the agent with the lowest value for the specified input. For example, if we created a Select for HeatCell, created a Minimize Query Term with Heat as the input Query Term, created Neighbor and Available Query Terms and set an Intersect as the target for all of those Queries, the result would be the the neighboring available cell with the lowest heat level.

Inputs:
[Real] The value we will minimize for.

Maximize

Finds the agent with the highest value for the specified input. For example, if we created a Select for HeatCell, created a Maximize Query Term with Heat as the input Query Term, created Neighbor and Available Query Terms and set an Intersect as the target for all of those Queries, the result would be the the neighboring available cell with the highest heat level.

Inputs:
[Real] The value we will maximize for.

Location 2D

Represents the location of the current agent for use in subsequent selections.

Inputs:
[Real]
[Real]

Boundary 2D

Represents a two-dimensional boundary within a space. (Not currently relevant for any general usages.)

Outputs:
[Real]
[Real]

All

Causes all agents that meet the other query terms to be included in a selection. Without this query term, a single random agent is picked out of all agents matching the query terms.

2.4.2.5. Random

Random functions are especially significant for agent models. Of particular interest are the weighted membership and random state and boolean value functions. You should be familiar with these functions so that you don't have to create more complex Action flows to accomplish the same thing.

Note that we only have support for uniform distributions as of this release. We're working on a collaborative design for evaluations that allow easy mixing and matching of random functions and distributions.

Random In Range

A pseudo-random value within that numeric range specified as drawn from a uniform distribution. The minimum values are inclusive. The maximum values are inclusive for integer inputs and exclusive for Real inputs.

Inputs:

[Numeral] The minimum value (inclusive).

[Numeral] The maximum value (inclusive).

Outputs:

[Numeral] The random number.

Random To Limit

A pseudo-random value between zero and the value specified by the (non-zero) input and drawn from a uniform range. That value is inclusive for Integers and exclusive for Reals. (Note that as with the random in range function in the context of real numbers the distinction between an exclusive and inclusive limit is essentially meaningless.)

Inputs:

[Numeral] The maximum value (inclusive).

Outputs:

[Numeral] The result.

Random Unit

A pseudo-random Real value between 0 and 1 drawn from a uniform distribution. (The distinction between inclusive and exclusive range is essentially meaningless in this context and we can assume that the result will never be greater or equal to 1.)

Outputs:

[Real]

Random Boolean

A value that is randomly true or false, i.e. a fair coin toss.

Random Weighted

An indexed value weighted against a probability distribution. The total probability must sum to 1.0. For example, an input of { .1,.2,.7 } under a uniform distribution would have 10% probability of producing "0" , 20% for "1" and 70% for "2". This function can then be used with Item to return a biased result from another list.

Inputs:

[Real] A list of values that will determine the resulting weighted index.

Outputs:

[Integer] A resulting indexed value bounded by 0 and the length of the input list - 1.

Random Member

Represents a random value drawn from the set of Real values specified.

Inputs:

[Real] Returns a random member of the supplied list of numbers.

Outputs:

[Generic] The value of the item at a random index.

Random State

A random specified value (option) from the specified state.

Inputs:

[Generic] The state to select items from. All items are included.

Outputs:

[Integer] The resulting option.

2.4.2.6. Graphic

Graphic functions are combined within Style Evaluate Actions to determine how to draw an agent within a visualization. One nice aspect of this approach is that the same style definition can be used in multiple places without changing any code. For example, the same style could be used to draw an agent on a two-dimensional grid within Escape, a three-dimensional shape within Escape, a Java Swing based visualization in Ascape, and an XML configured visualizaiton in Repast Symphony.

To define a graphic style for an agent, design a flow in which you create Evaluate Actions for color and shape, and then create an Evaluate Action with the graphic fill or outline function as a target of these.

Shape Oval

Draw a generic oval.

Shape Rectangle

Draws a rectangular shape.

Shape Inset

Shrinks the current shape by the input amount. (The overall scale is currently unspecified, but in most implementations should be 20.)

Inputs:

[Integer] Number of nominal pixels to inset.

Shape Marker

Draw a marker, that is a graphical indicator that can be used to add an additional que about the object state. For example, in a two-dimensional graphics representation this might be a small shape drawn inset at the corner of the larger shape.

Shape Marker 2

Represents a marker placed in a different location from the other shape markers.

Shape Marker 3

Represents a marker placed in a different location from the other shape markers.

Color RGB

A color specified by the three inputs for Red, Green and Blue color components. Those inputs are expected to be in the range 0..1.

Inputs:

[Real] A value from 0.0 to 1.0.

[Real] A value from 0.0 to 1.0.

[Real] A value from 0.0 to 1.0.

Color Red

The color red.

Color Yellow

The color yellow.

Color Blue

The color blue.

Color Orange

The color orange.

Color Green

The color green.

Color Purple

The color purple.

Color Black

The color black.

Color White

The color white.

Color Gray

The color gray.

Graphic Outline

Draws an outline of the last evaluated shape, using the last specified color or the default color (usually black) if none has been specified.

Graphic Fill

Fills the last evaluated shape with the last specified color or the default color (usually black) if none has been specified.

2.4.2.7. Time

Time functions return values related to model execution time.

Now

The current simulation period, that is the number of iterations that the model has gone through, or in the case of models with calibrated time, the number of iterations added to the model's nominal start time.

Outputs:

[Integer] The current period.

2.4.2.8. Math

The math functions use the extremely well specified and tested routines from the Java Math library. (Because of copyright restrictions, we aren't able to include the exact definitions here. Click on the links to get more details on each function.)

Sine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sin\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sin(double))

Cosine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cos\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cos(double))

Tangent

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tan\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tan(double))

Arc Sine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#asin\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#asin(double))

Arc Cosine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#acos\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#acos(double))

Arc Tangent

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan(double))

Convert Degree to Radians

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toRadians\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toRadians(double))

Convert Radians to Degrees

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toDegrees\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toDegrees(double))

Exp

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#exp\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#exp(double))

Log

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log(double))

Log b10

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log10\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log10(double))

Square Root

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sqrt\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sqrt(double))

Cube Root

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cbrt\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cbrt(double))

Remainder

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#iEEEremainder\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#iEEEremainder(double,%20double))

Ceiling

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ceil\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ceil(double))

Floor

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#floor\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#floor(double))

Rount

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#rint\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#rint(double))

Arc Tangent Rectangular

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan2\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan2(double,%20double))

Power

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#pow\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#pow(double,%20double))

Absolute Value

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs(double))

Maximum

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#max\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#max(double,%20double))

Minimum

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#min\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#min(double,%20double))

ULP

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ulp\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ulp(double))

Sign

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#signum\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#signum(double))

Hyperbolic Sine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sinh\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sinh(double))

Hyperbolic Cosine

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cosh\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cosh(double))

Hyperbolic Tan

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tanh\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tanh(double))

Hypotuneuse

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#hypotuneuse\(double,%20double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#hypotuneuse(double,%20double))

Expn1

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#expm1\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#expm1(double))

Log1p

See JavaDoc: [http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log1p\(double\)](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log1p(double))

2.4.2.9. List

List functions are used for working with arrays and other functions that have lists as output.

Item

Returns the item at the specified index from the list of items provided. Those items will typically be input primitives such as Integer or Real values.

Inputs:
[Generic]
[Integer]
Outputs:
[Generic]

Length

The number of items in the provided list of items.

Inputs:
[Generic]
Outputs:
[Integer]

2.4.2.10. Distribution

One of the most common tasks in the Agent Modeling process is the creation of agents with particular states drawn from a distribution. For example, you might want to create a number of agents with wealth randomly distributed between some minimum and maximum values. The distribution functions greatly ease the process of setting up those initializations and their associated parameters.

Uniform Cross Distribution

A random number taken from a distribution of values as defined by a cross of all values. (See Cross Distribution.) This function then returns a value drawn from the minimum and maximum values as determined by the current agent state. In the cross distribution, each of the values is treated independently so that an input attribute is created for every potential combination of states.

Inputs:
[Generic] The list of states to factor into the distribution. This is a multi-argument, which means that you can specify any number of attributes.
[Real] The set of attributes that will determine the minimum value of the function result based on the current state of the agent. Note that this is a multi-argument.
[Real] The set of attributes that will determine the maximum value of the function result based on the current state of the agent. Note that this is a multi-argument.
Outputs:
[Real] The resulting random number based on the current agent state and the input parameters.

Uniform Additive Distribution

A random number taken from a distribution of values in which each of the minimum and maximum values are added to determine a total minimum and maximum value. (See Additive Distribution.) In the additive distribution, each of the values is treated as dependent on the others so that an input attribute is only created for each separate state.

Inputs:
[Generic] The list of states to factor into the distribution. This is a multi-argument, which means that you can specify any number of attributes.
[Real] The set of attributes that will determine the minimum value of the function result based on the current state of the agent. Note that this is a multi-argument.
[Real] The set of attributes that will determine the maximum value of the function result based on the current state of the agent. Note that this is a multi-argument.
Outputs:
[Real] The resulting random number based on the current agent state and the input parameters.

Cross Distribution

A value taken from a set of (auto-generated) attributes based on the value of each state included. For example, if the multi-values included a state X with values A and B and a state Y with values I and II, this distribution would create separate input attributes for AI, AII, BI and BII. Then for an agent with States A and II this function would return the value specified by the AII input attribute.

Inputs:

[Generic] The list of states to factor into the distribution. This is a multi-argument, which means that you can specify any number of attributes as arguments.

[Real] The set of attributes that when multiplied against each other will determine the value of the function result based on the current state of the agent.

Outputs:

[Real] The resulting value based on the current agent state and the input parameters.

Additive Distribution

A value taken from a set of (auto-generated) attributes based on the combined values of the states provided. For example, if the multi-values included a state X with values A and B and a state Y with values I and II, this distribution would create input attributes for A, B, I and II. Those values would then be added together, so that for an Agent with state A and II this function would return A + II.

Inputs:

[Generic] The states to include in the distribution. This is a multi-argument, which means that you can specify any number of attributes as arguments.

[Real] The set of attributes that when combined with each other determine the value of the function result based on the current state of the agent.

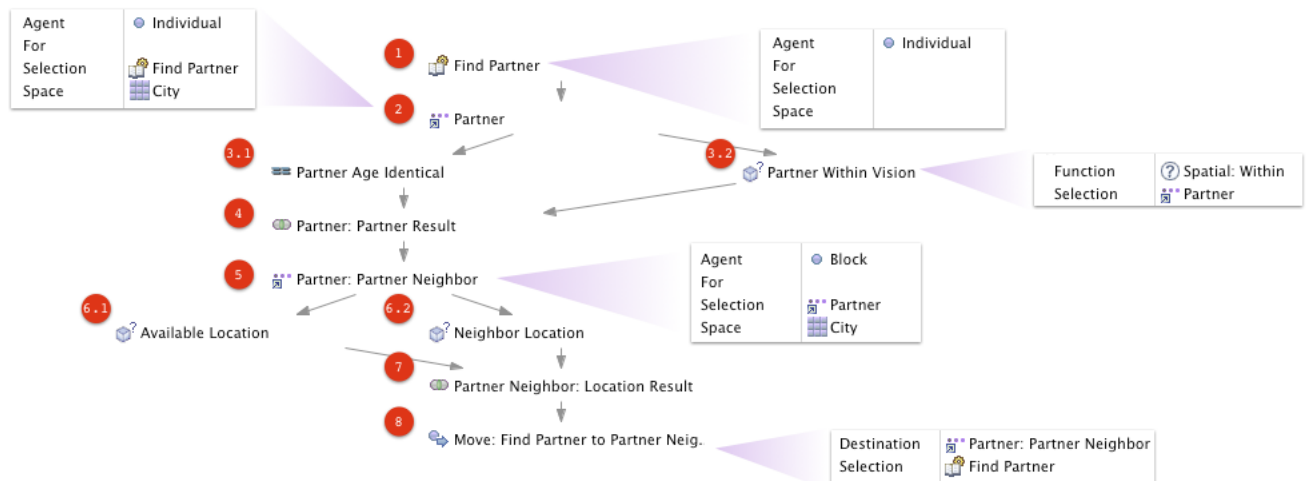
Outputs:

[Real] The resulting value based on the current agent state and the input parameters.

2.4.3. Examples

2.4.3.1. Spatial

For examples of how spatial functions can be used within action flows to provide agents with complex movement behaviors, see the Modelers Guide actions examples. In the following example from that section, we define an action that causes a partner agent to move to an available neighboring space.

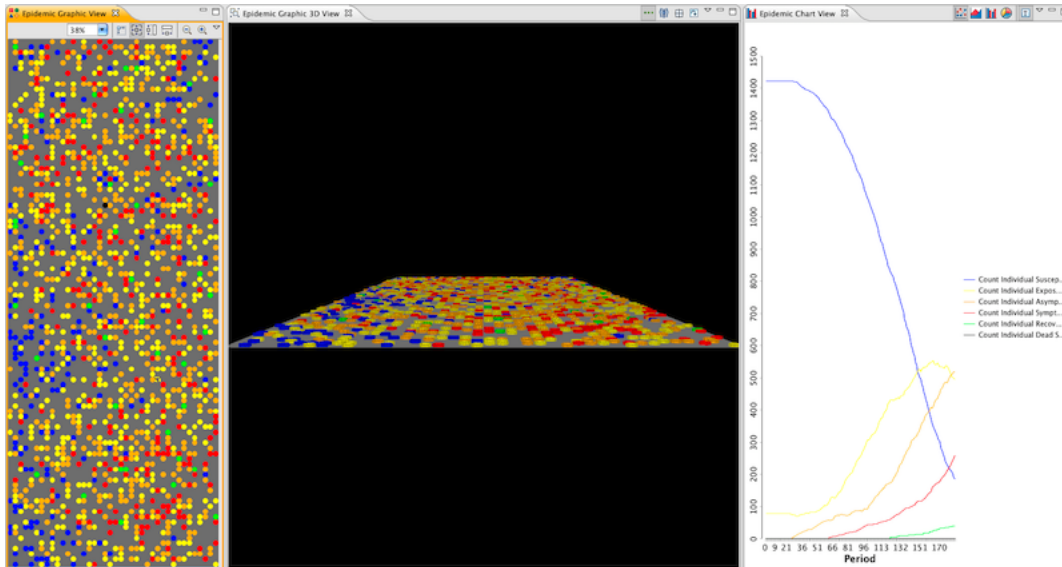


2.4.3.2. Graphic

In the following action flow for the epidemic style, we've create Query Actions to determine each agent's current state, picked a color based on that, and then used a shared target to select a shape for the agent style and fill it:



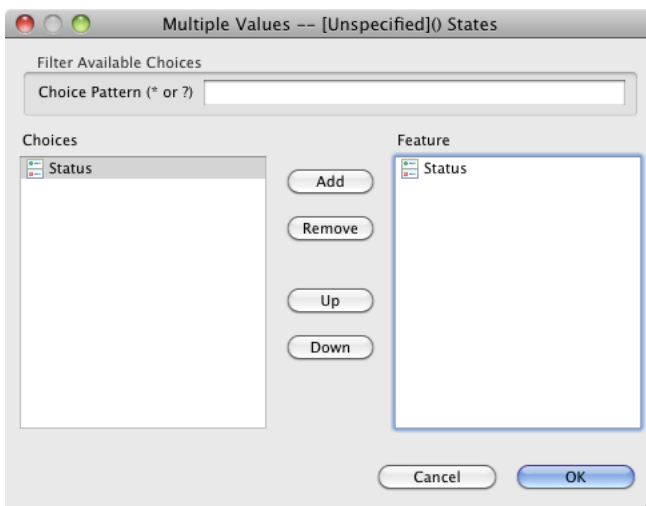
After saving the model we can execute the two and three dimensional visualizations. Note something really nice -- even the charts have used the colors we've defined!



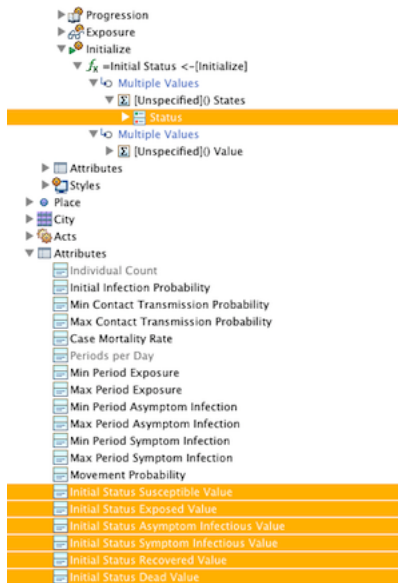
2.4.3.3. Distribution

In the following example, we walk through the process of using a distribution functions, demonstrating how we can easily modify the Epidemic model so that instead of simply setting an initial exposed population, we can define factors that take together will determine an individuals initial exposure state. We simply:

1. Create an Evaluate Action called "Initial Status".
2. Set the function to "Cross Distribution"
3. Opened the "Multiple Value" node in the editor, and clicked the "Multiple Values" item within it.
4. Selected the "Status" attribute.



The appropriate attributes are automatically added to the model, as you can see below.



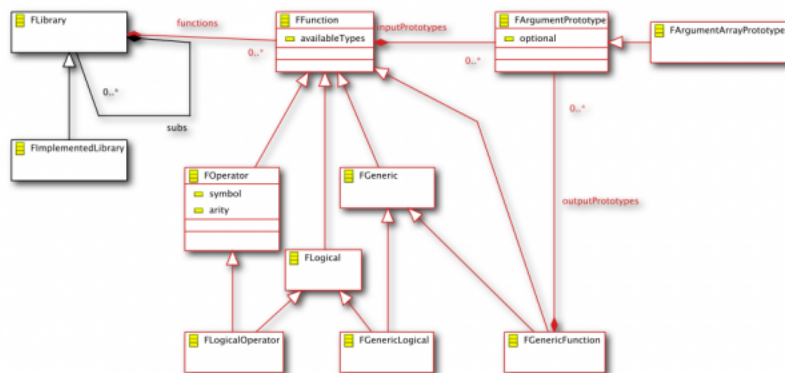
In order to assign these values to the agent, we'd simply need to assign the results of this Evaluate Action to the agent.

2.4.4. Reference

2.4.4.1. Diagrams

The following diagram may be helpful to readers familiar with UML and Meta-modeling:

Meta-Classes

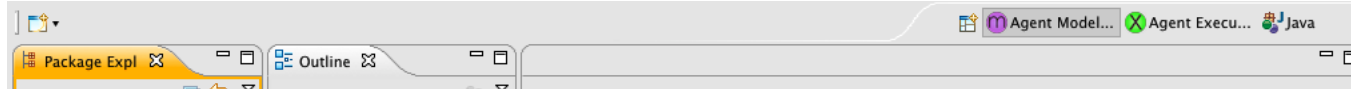


Chapter 3. User Guide

In this section of the guide, we'll discuss specific aspects of the Agent Modeling tools and show you how to use them in your day to day agent development activities. We discuss only tools specific to Agent Modeling itself. For more general information about the tools, such as how to customize editors, views, and perspectives, see the **Workbench User Guide**.

3.1. Overview

The Agent Modeling Platform provides two general modes or "perspectives" for working with Agent Models.



Agent Modeling supports all aspects of working with models, including editing models and automatically generating all the of your code and documentation. **Agent Execution** supports running and exploring those models. A key feature of the Agent Modeling Platform is the ability to execute models within the same environment that they are developed within -- you don't need to launch a separate environment in order to run a model. Agent Execution automatically activates when you launch a model. We'll discuss the agent modeling tools first, and then turn to agent execution.

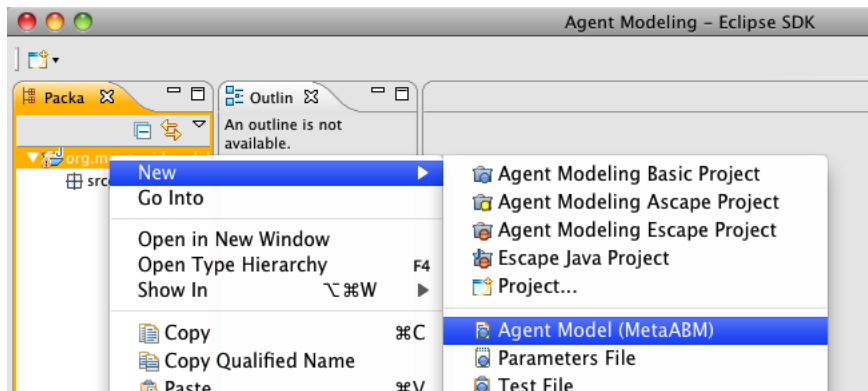
3.2. Modeling

3.2.1. Perspective

The Provides a custom layout menus and tools specific to working with agent models. Because the agent modeling process also often involves Java and Eclipse Plugin development we include easy access to many of those tools here as well.

3.2.2. Creating Projects and Models

You can create projects and project components using the Popup menu. Just click in a blank space within the Package Explorer. Any installed project targets are displayed in this menu. For example, if you've installed the Symphony target, you'd see that displayed in this menu as well.

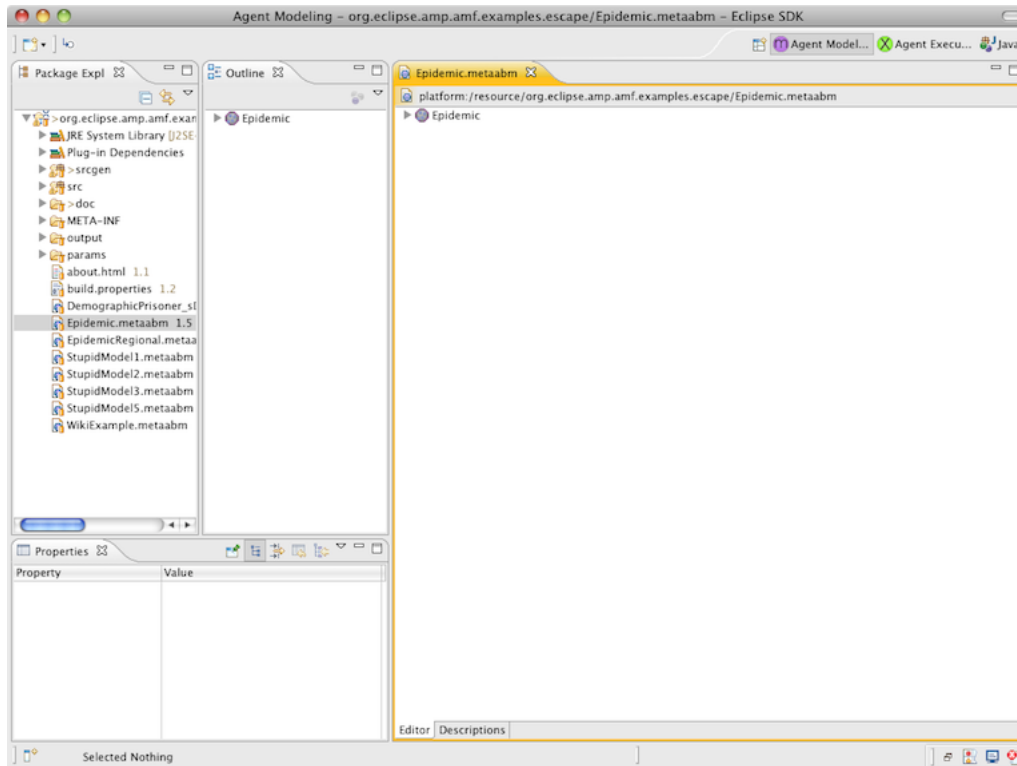


3.2.3. Menus, Popups and Toolbar

The popup menus and application menus provide access to various model features and are context sensitive. Throughout these screenshots, we've customized the toolbar in order to only show the agent Modeling specific features.

3.2.4. Views

By default the workbench includes a number of views. See the Workbench documentation for more details on how they can be customized accessed and used.



3.2.4.1. Editor

This is not technically a view in itself. It is the major focus of the workbench and contains any edit models or other files such as Java source or parameters.

3.2.4.2. Default Views

The default modeling views are visible by default.

Package Explorer

Supports navigation within projects.

Properties

The properties allows you to view and edit specific details for the currently selected object. For example, if you select a model context, you'll be able to edit its attributes here.

Outline

The outline view supports easy navigation within the edited files. See the model editing sections for more on how the outline can be used to assist exploration of Agent Models.

3.2.4.3. Diagnostic Views

There are a number of views that can be used to explore issues that might come up during the modeling process. You can see them on the lower left-hand corner of the screenshot above. Click on one of the icons to view their contents.

Problems

This view is one that you'll become very familiar with. It is used to display specific about problems with any of the artifacts (files) in your modeling project. If you see a red or yellow marker on a file, opening the view will present a list of the issues that are currently being reported. For a usage example, please see the Tutorial.

Console

This view displays text (console) output for appropriate processes. For example, if you launch an Ascape project, this will display any output that would go to the Java console.

Errors

This is another view that you will find valuable in your day to day modeling activities. The error log lists everything important that happens during model execution. It's the first place to look if something mysterious goes wrong, and when you report problems it's always helpful to include anything that could be relevant in the log. Despite its name, the Errors view is not just for reporting errors -- it is also used to report progress on normal operations. For example, when models are automatically generated that is reported to the log, as we can see in the following example:

3.2.5. Modeling Tree Editor

The Eclipse Agent Modeling Framework includes a full-featured model editor based on the Eclipse Modeling Framework's Edit tools. All aspects of an AMF model can be managed from within this tool. Note that the tree editor is only one of many ways of editing a model. Other editors of AMF models include textual languages and custom editors that are part of commercial offerings.

3.2.5.1. Opening the Editor

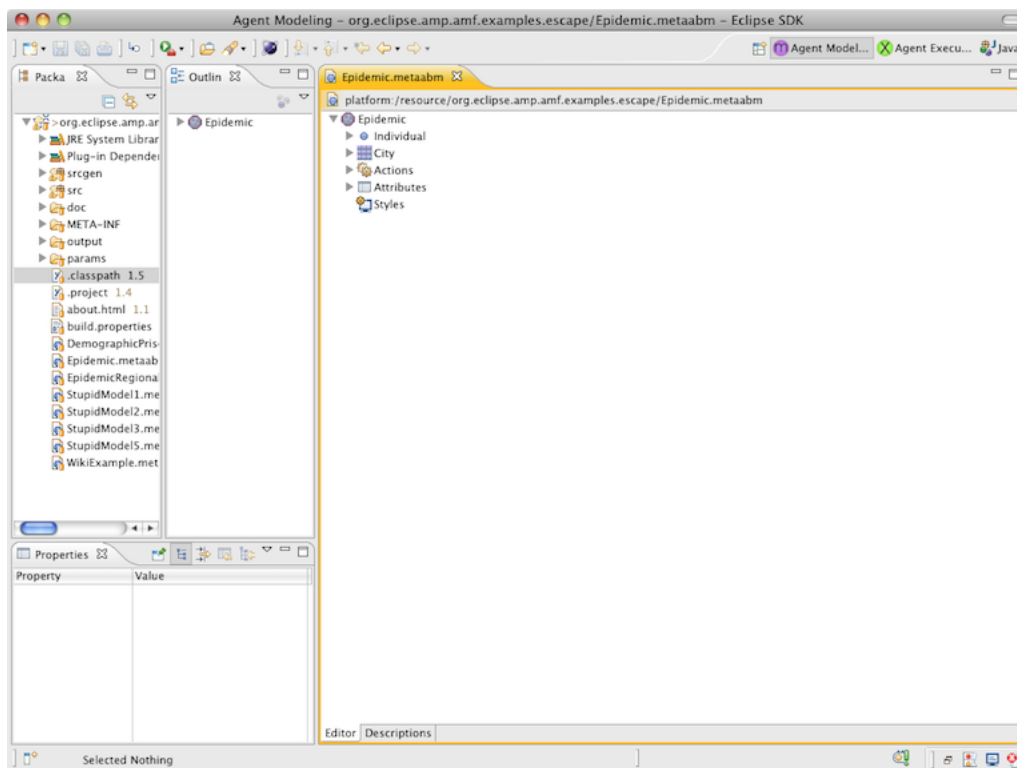
To open a model in the editor, simply double-click on it. If the model doesn't open in the model editor (if for example it had been previously opened using another editor), you can always access the editor using **Open With > Other...** and selecting "MetaABM Editor". The editor has two pages, an Editor page that we'll focus on first and a "description" page that we'll discuss at the end of this section.

3.2.5.2. Structure

The model is composed of nodes representing specific model entities such as agents, actions and spaces. For details about any of these entities, see the Concepts section.

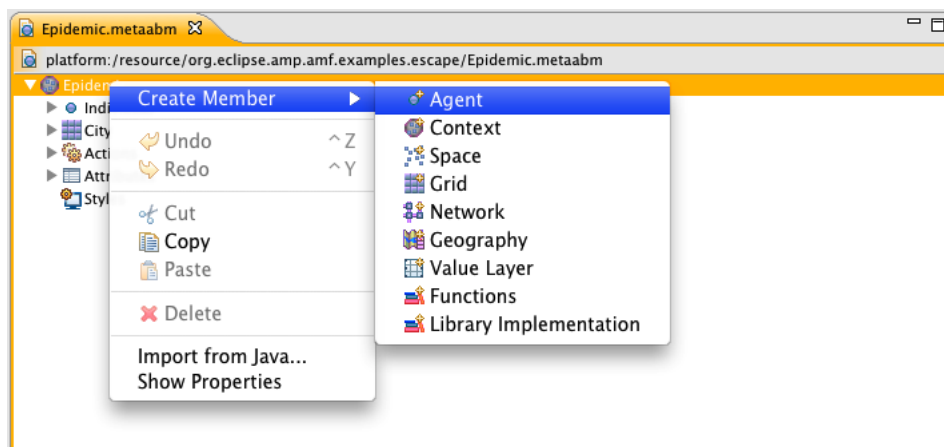
Opening Components

After opening a model, you can see the contents by clicking the Triangle symbol to the left of an item. For example, opening the root context node, we see:



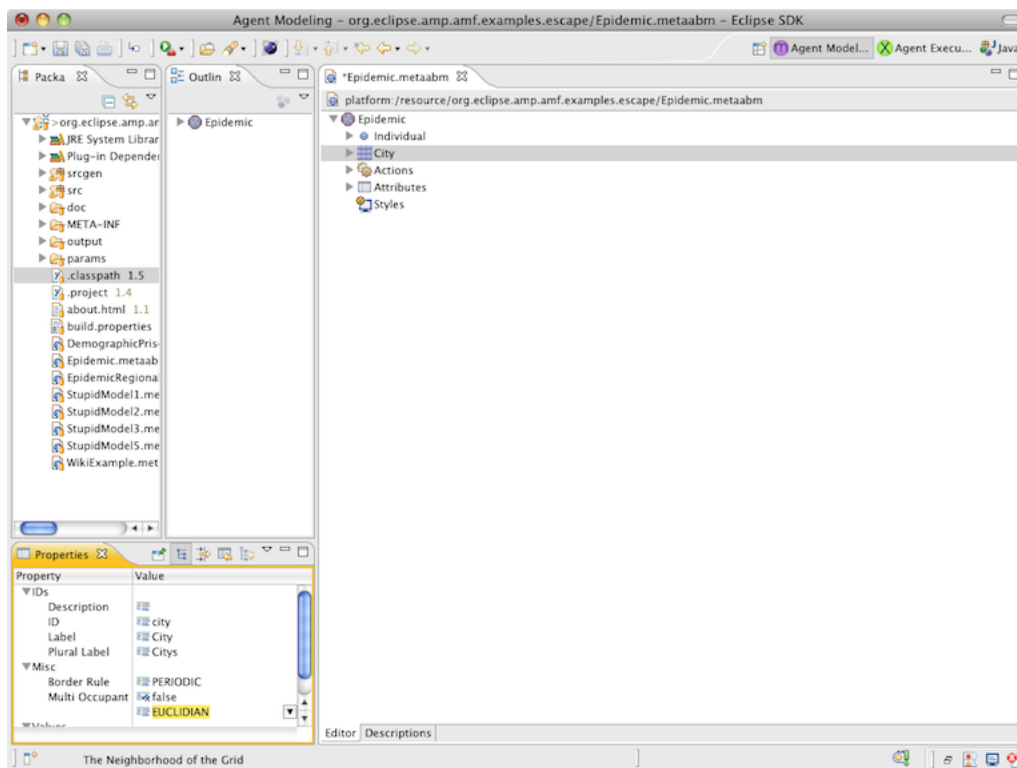
Creating Components

You add nodes by right-clicking on a node, like so:



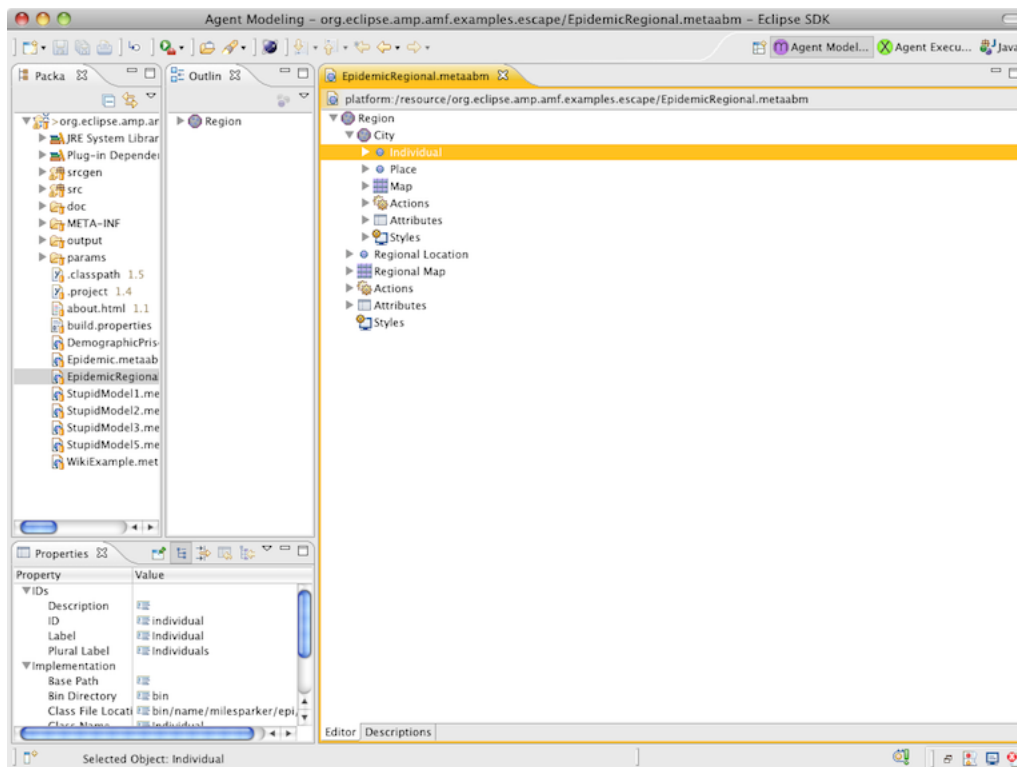
Editing Components

To edit components, select the node you want to modify. The **Properties View** will be updated with the agent details. Use the properties view to modify the values. In the screenshot below, we're editing the values for the City space.



Moving Components

You can often rearrange model components by dragging them from one place to another. In the following example, we're creating a model of a regional epidemic by creating a City sub-context and moving the agent into it:



Removing Components

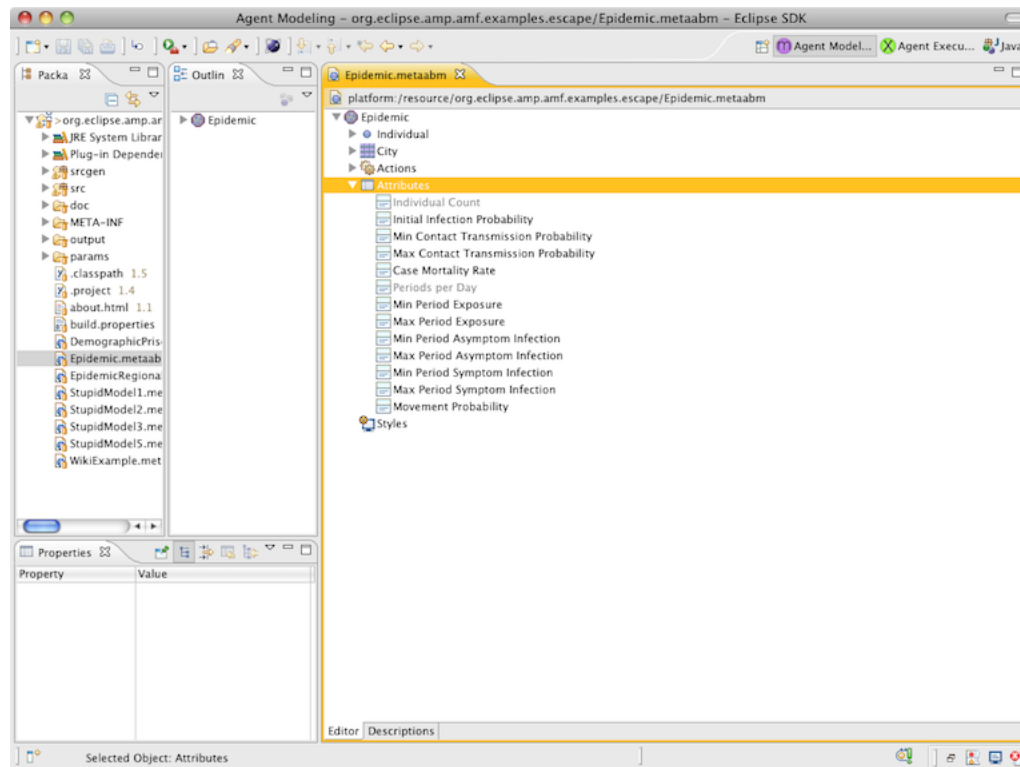
You can remove actions by deleting or cutting them using the popup menu. When you delete an entity, all of its children are deleted as well, except in some special cases (such as with actions) where other paths to that entity still exist.

Copying Components

To make a copy of an entity and place it in a new location, option-drag the entity to the destination. You can make a copy in the existing location by dragging it into the same parent. The name will automatically be updated with "copy" appended to it.

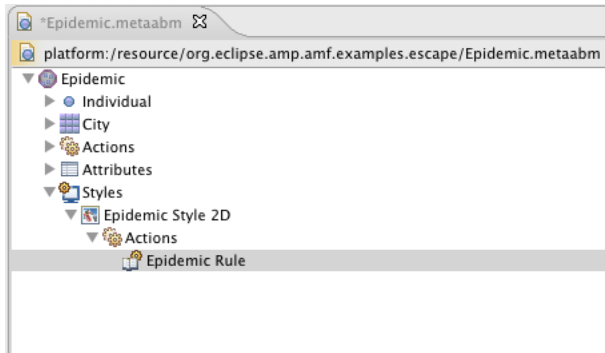
Editing Attributes

The attributes node -- along with the actions and styles nodes -- represents a group of components rather than an entity itself. It contains all of the attributes for the parent agent or context. In the examples below, we can see the attributes for the Epidemic route model. Note that the top level attributes in a root context act as the parameters for the model itself.



Editing Styles

The Styles node is another group node, in this case representing a set of styles that can be used to visualize the agents. When creating a style, you will need to create a Rule for each style as well, like so:



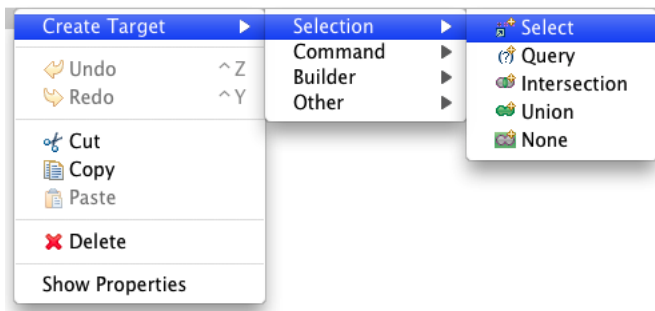
3.2.5.3. Actions

Actions are a key and relatively complex aspect of the model editor. Because actions are actually related in a graph structure, a tree-based editor will not be able to represent the underlying structure of action relationships directly. (This is actually no different from most development environments -- for example, Java code is edited in a text based editor, but the call structure represents a complex graph. Advance tools developed by AMP contributors do support more sophisticated methods for action browsing and we'll have simple versions of them available in future releases of the AMF edit tools.) Therefore, it's helpful to take time to understand how action relationships appear in the editor.

Like attributes and styles, every agent has a single Actions nodes which contains its Root Action(s).

Creating Actions

You create actions as with any other component, by right-clicking on the source action and choosing the new action. The menu is organized by usage.

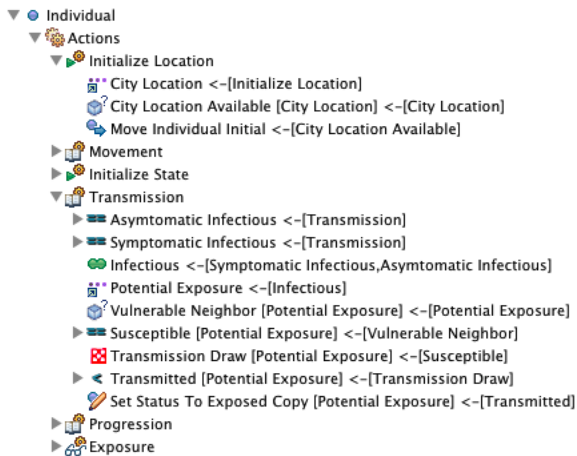


Editing Actions

Actions and their inputs are edited just like any other model component. Click on the action or input you want to edit and then make changes to it in the properties editor.

Order

Generally, action sources will appear above their targets. In the case where there is only one source for a given target, and that target has no other sources, they will appear directly above one another. It's important to note however that *the order of the nodes does not indicate a specific source and target relationship*. For example, in the case where there are multiple targets for a source, they will typically follow that source immediately. To make the relationships clear, every action lists its source actions as part of the action label. Agent selections also appear as part of the label.



In the above example, the Initialize Location "Initialize Action" is straightforward. We simply define a Select Action, a Query Action target for that selection, and then a Move Action target for the query. The "Transmission" Rule is more complicated. Note for example that the "Infectious" Union Action is the target of both the "Asymptomatic Infectious" and "Symptomatic Infectious" Query Actions. The "Vulnerable Neighbor" action has the label "Vulnerable Neighbor [Potential Exposure] <- [Potential Exposure]" indicating that the "Potential Exposure" action serves as its selection as well as its source.

Initial Order

A new action's initial target will be the action that you clicked on when creating it. For an action that should always act within a given root action (i.e. Rule, Schedule, etc..), add it to the root action. A root action can have more than one target.

Changing Sources and Targets (Reordering)

An action's target can be changed by dragging it over the new target action. Note that this is a different behavior from that of standard entity movement. You cannot make an action a target of an action that is itself a source of the modified action! (See the concepts section for why.) Using the default tree editor you cannot change an action's targets directly; instead select the action's target and move that.

Adding Sources and Targets

As discussed earlier, actions often have multiple sources and targets. To make an action the target of an additional action, click on target action, hold down the control key, and drag the action to its new destination. See the tutorial for a complete example.

Removing Sources and Targets

You cannot remove single sources from targets directly using the tree editor. Instead, first drag the agent to a source nodes that will be part of its changed set of sources. That will remove all existing actions sources but for the one you have just dragged it to. Then add back any of the other source and target nodes you wish to retain.

Removing Actions

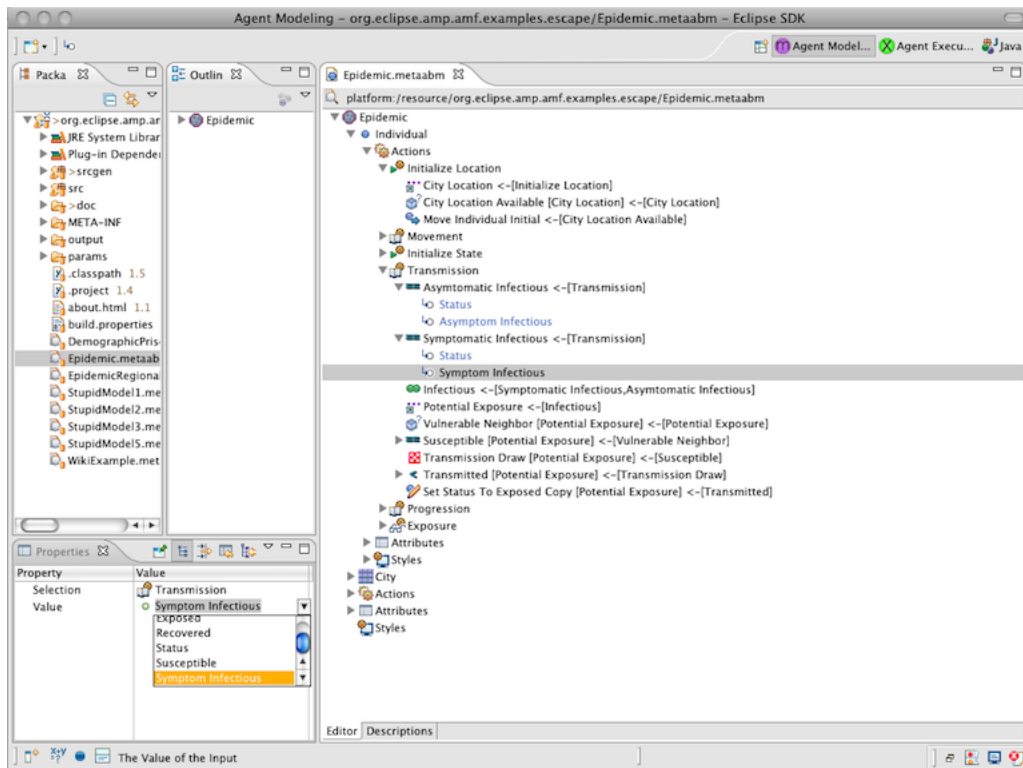
You can remove actions in the same way as with structural model components. Note that just as when you delete an agent, all of that agent's attributes, actions and styles are also deleted from the model, when you delete an Action, any of its targets will also be removed, unless there is some other source action path that connects it to the root action. And of course, any targets of such targets will be affected in the same way and so on. If you remove a node from high in the action tree a lot of nodes could disappear at once! If you have actions that you want to retain as part of the flow, you should first make them targets of a new action before deleting their parent action.

Copying Actions

Copying actions works just as with other entities, and copies will become targets of their option-drag destination.

Query and Evaluation Inputs

The number of inputs is determined by the selected functions. (In rare cases where a function can take an arbitrary number of arguments you may need to create additional values or delete existing ones.) In the following example, we're picking the Symptom Infectious status for a query that will affect the potentially exposed agents. Note that is instead you wanted to compare another kind of value -- for example an Integer value -- you would need to change the first input listed before changing the second input so that you will be able to get the appropriate options for the second.



Input Literals

To create an input value, right click on the input and select **Create Member > Literal**. Then specify the actual value in the Value property in the **Properties View**.

3.3. Building

3.3.1. Building Models

If you've used other development environments, you're probably used to a separate build step. For example, if you edit a set of Java files, you might invoke a compile command. Eclipse and the Agent Modeling Platform support automatic building. This means that in order to build your code, you simply save the model and the environment takes care of the rest. What gets built is defined by the project. For example, if you save a model that is contained within an Agent Modeling Escape Project, the following steps occur automatically:

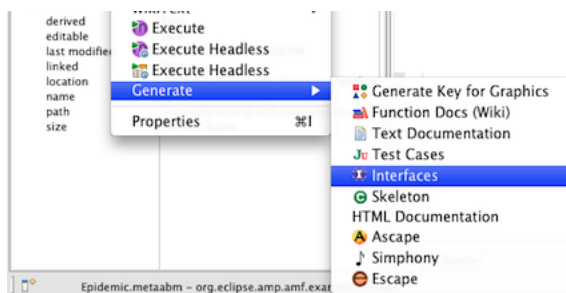
1. The Escape builder generates Java code for the Escape API, including support for specialized graphics and (if enabled) 3D visualization.

2. The Documentation builder generates custom html documentation for the model.
3. The Java builder takes the Java code generated above and compiles it.
4. The Manifest and Schema builders package the project for use as part of the Eclipse plugin environment.

So what do you do if you want to generate code for a different target, such as Repast? Here, you simply create another project and drag the model into it. You can also edit the builders for a given project (see the Workbench Documentation) but you'll only want to do that if you're making permanent changes to the project itself.

3.3.2. Generating Specialized Model Artifacts

You can also manually generate code for models. This is useful if you want to create code for a model in a non target project and don't want the automatic build capabilities. There are also specialized targets that can be generated manually that are bundled with the tools -- AMP plugin developers can easily add custom generation targets for this menu. To generate custom artifacts, right-click on a model and select **Generate**.



Targets include:

3.3.2.1. Platform Targets

Creates code for one of the installed targets, such as Ascape, Escape, and Symphony. (Again, these generators are unnecessary for projects that already have target specific builders configured.)

3.3.2.2. Interfaces

Creates a complete set of interfaces for the model classes. These can be very useful for development and integration in enterprise environments. Generated classes are placed in the src directory with "I" added to the name. For example, if you generate interfaces for a model with an "Individual" agent, this target will create an "IIndividual" interface that includes getters, setters and methods for all of the agent's attributes and actions.

3.3.2.3. Skeleton

Creates a base class for the model. This is essentially a complete implementation, but without the action implementations. Again, these classes can be useful when generating code for use in enterprise and other specialized environments.

3.3.2.4. Test Cases

Generates base support (needing customization) test cases for use in JUnit tests.

3.3.2.5. Text Documents

Creates simple text documentation for use in other documents.

3.3.2.6. Generate Key Graphics

This specialized target supports the creation of graphic keys for the model. To use this target:

1. Generate the code.

2. Add a dependency to the Manifest for "org.eclipse.amp.amf.gen.extras".
3. In the srcutil directory find the Java source code for the class {RootContext}GraphicsWriter. For example, for the Epidemic model, this would be srcutil/name.milesparker.epi/EpidemicGraphicsWriter.java.
4. Right-click, and select **Run As > Java Application**.
5. Refresh the project by right-clicking on it and selecting **Refresh**.
6. The Doc directory will now contain a number of new files, including
 - a. {RootContext}Key.png
 - b. {RootContext}GraphicsKey.html

Both files contain graphic representations of the model using the definitions defined by the model styles, very useful artifacts for inclusion in papers and web pages describing your model. For example, here is the EpidemicKey.png:

<i>Agent Name</i>	<i>Style</i>	<i>Condition</i>	
Individual	Individual Style 2D	Status is Susceptible	●
Individual	Individual Style 2D	Status is Exposed	●
Individual	Individual Style 2D	Status is Asymptom Infectious	●
Individual	Individual Style 2D	Status is Symptom Infectious	●
Individual	Individual Style 2D	Status is Recovered	●
Individual	Individual Style 2D	Status is Dead	●

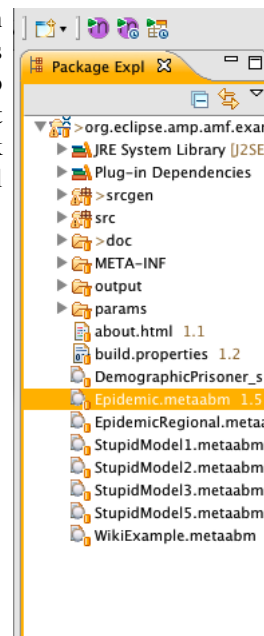
3.3.2.7. Function Docs

Generates WikiText documentation for function libraries. We use it to create the function documentation in this guide!

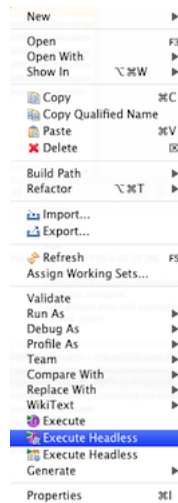
3.4. Executing

3.4.1. Launching a Model (AMF)

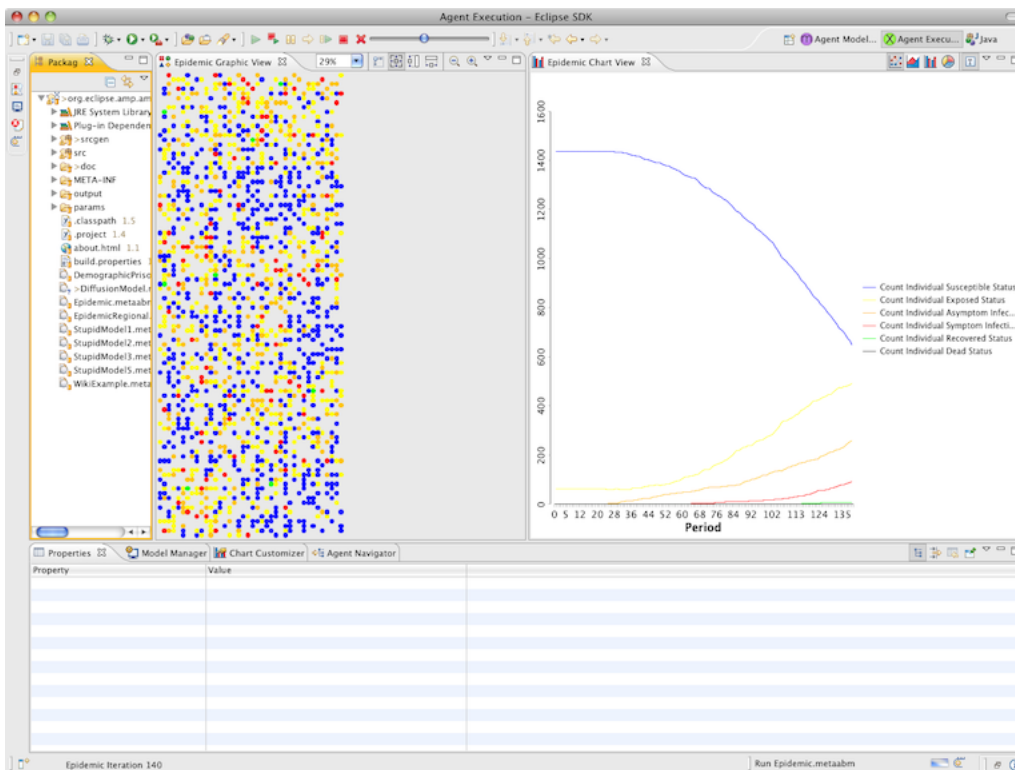
Launching AMF and Escape models is easy. When an AMF file is selected -- in any of the perspectives -- the toolbar and Application menus are updated to reflect the file you've selected and provide convenient access to other functions. For example, when we click on a metaabm file in the package explorer, model execution buttons appear in the toolbar.



If you right-click on a file a pop-up menu appears like the one to the far right -- in this case we're selecting the "Execute Headless" option. To launch a model, just select one of the options. (Note that the execution framework doesn't know whether your code exists in an Escape project or another target project. If you attempt to execute a .metaabm model in an Ascape project for example, you will get an error.)



Once a model has been launched, the Agent Execution Perspective automatically becomes active. The Provides a custom layout menus and tools specific to executing agent models.



The execution options are:

3.4.1.1. Execute

Launches the model using default graphics, opening the Agent Execution perspective.

3.4.1.2. Execute Headless

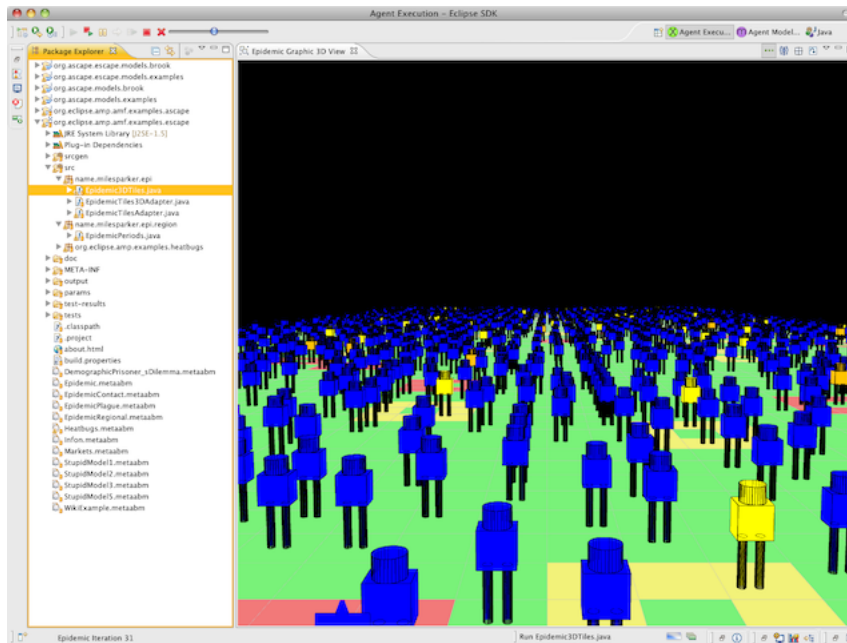
Launches the model without graphics, opening the Agent Execution perspective.

3.4.1.3. Execute Headless (Data)

Launches the model with an observer that collects data into the AMF adata model representation.

3.4.2. Executing a Model (Java / 3D)

You can execute any Escape model directly from its Java file by right-clicking on it. This is used for models that have been written directly in Java, or that you have created or that have been automatically generated, such as the 3D versions of AMF models. In order for the model to launch correctly, it must have as a superclass the Escape "Scape" class. In the following screenshot, we've launched a 3D version of the Epidemic model. The 3D models are automatically generated for all .metaabm models. You can also launch a model into 3D by clicking on the model file and clicking the second (Execute Model with 3D) execute button.



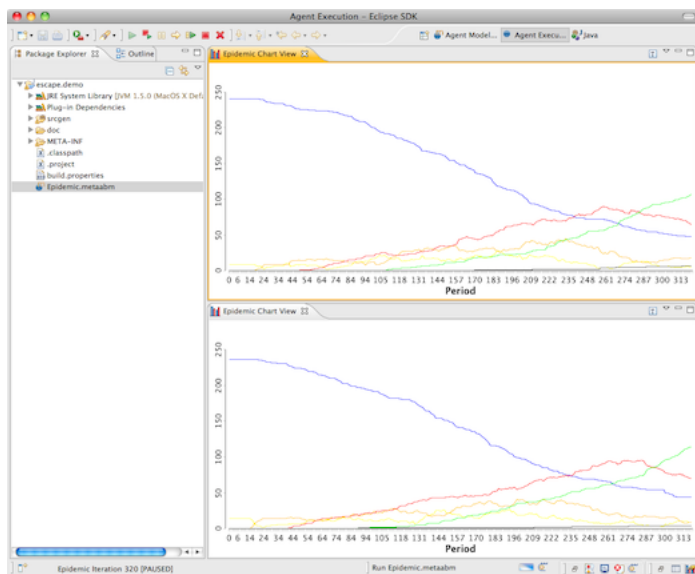
3.4.3. Controlling Models

Once a model has been launched, the toolbar buttons allow you to control model execution.



Model Execution Controls

From right to left, you can start, restart, pause, step, stop and close a model. You can even run multiple models and control them independently. You can also move views around, close them and so on as with any other Eclipse views. Here we're running two separate models for comparison.

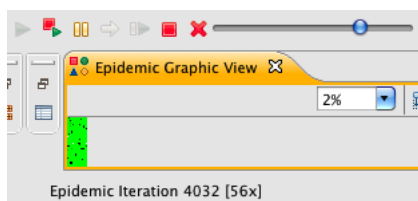


Speed Slider

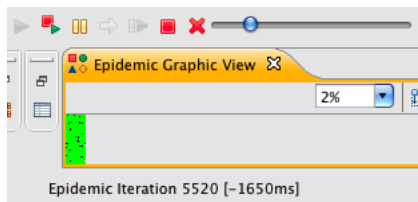
A special feature of the modeling tools is the "Speed Slider". This allows you to dynamically "speed up" and "slow down" the model execution. You can see where you've set the current speed in the status bar in the lower left hand corner of the environment.

Actually, you're not controlling the speed of model *execution* at all. Agent models are almost always constrained not by the time of model execution -- a typical model can execute many thousands of iterations a second -- but by the time it takes to draw visualizations. By updating the views less frequently we allow the model to run at full speed until the next visualization period. This gives the illusion that we're speeding up the model. When we slow the model down, we're inserting a wait period between each model iteration.

To increase model execution speed, move the slider to the right. Here we're only updating the model every 56 iterations.



To decrease speed, move the slider to the left. Here, we're pausing between each iteration for 1.65 seconds.



3.4.3.2. The Active Model

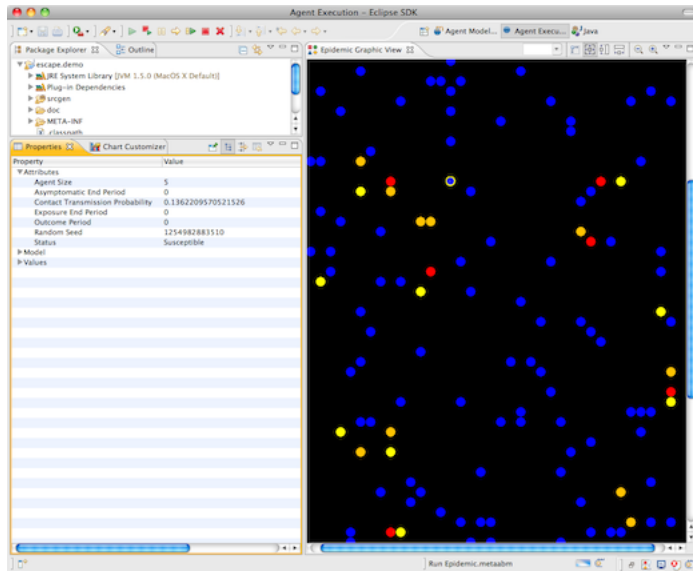
An important concept in the execution workbench is the "active model". The active model is the model that is controlled by the toolbar buttons. As discussed, the Escape environment supports more than one running model at any given time. The active model is the current "focused" or front-most model, there can be only one active model at any given time and whenever any models are running one of them will be active. To make a model become the active model, you simply select a view of that model or select the model in the model manager (see below). When a model is closed, another model is automatically activated.

3.4.4. Views

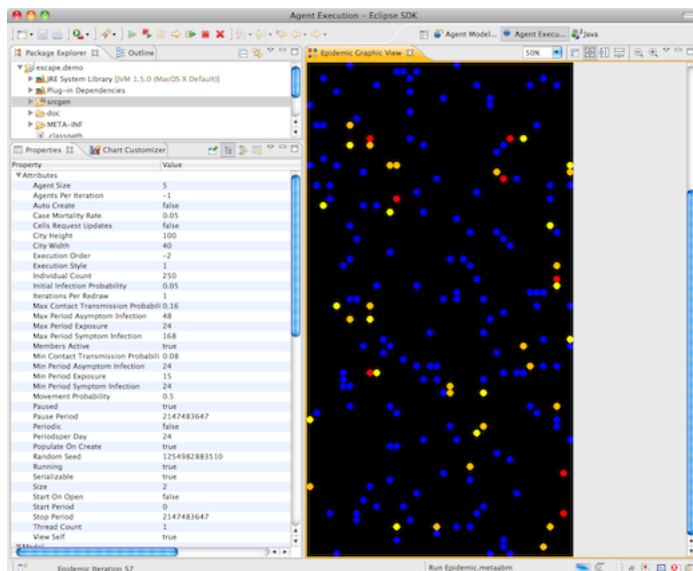
There are many views specific to the Agent Execution environment that will enable you to explore and control running models.

3.4.4.1. Properties

If you want to find out more about an agent, show the properties view, and click on agent in the 2D view or any other views that support agent selection such as the tree view.

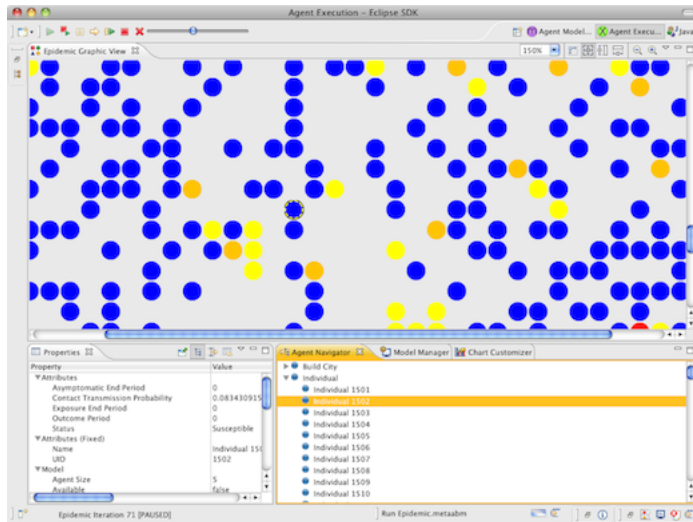


You can experiment with different parameters (settings) for models by then clicking on the model in the Model Manager or by selecting a general mode area such as the gray area enclosing the 2D view.



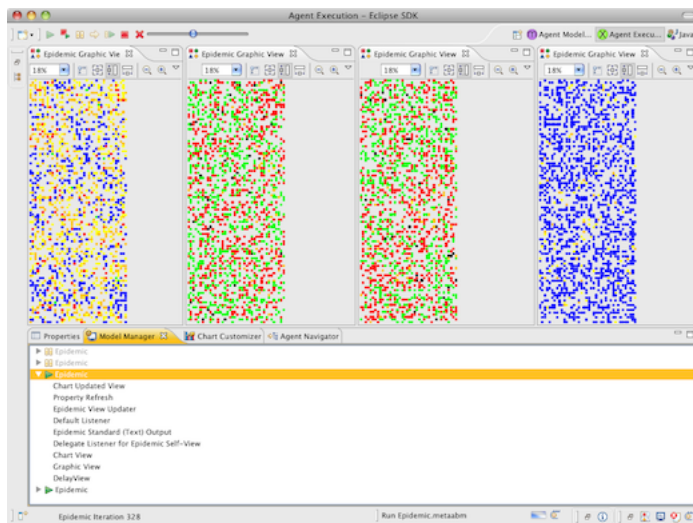
3.4.4.2. Agent Navigator

The **Agent Navigator** allows you to select agents from a tree view. The agent selection is coordinated across views so that for example when you select an agent in the navigator it is also selected in the 2D view. In the following screenshot you can see an agent selected in both views as well as the properties editor.



3.4.4.3. Model Manager

The '**Model Manager**' allows you to examine and control the status of all running models. In the following screenshot, we've launched four separate models so that we can compare the typical model state at different periods.

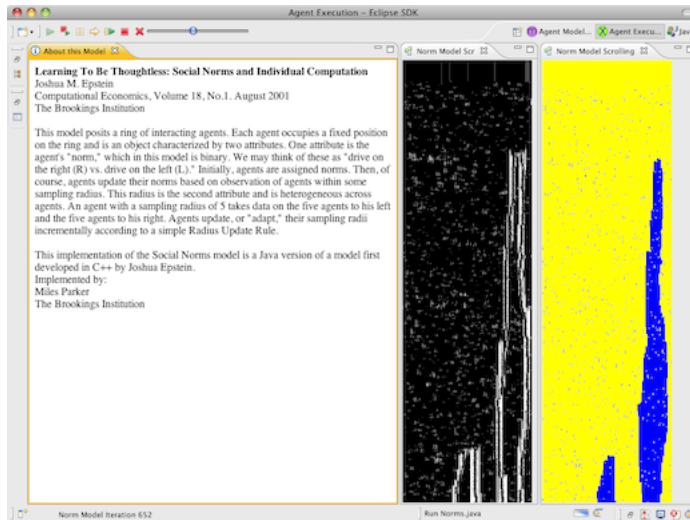


The manager shows that two of the models are running and two are paused. By opening the model node, we can see the views that are currently being displayed. Note that we mean something different by "Views" in this context. Here "Views" are any thing that is monitoring the state of a running model. A view may not have a graphical component at all.

You can make any model the current active model by clicking on its node in this view.

"About this Model"

The '**About this Model**' view displays information about a (AMF or Java based) model if it has been provided by the developer. In order to appear in this dialog, create a file named "About[MyModel].html" where "MyModel" is the model's Scape Class Name (not the AMF model file name). The file should be placed in a "res" source folder in the model project directory in a parallel directory path to the root scape's Java class package. For example, an about file for a model defined by the scape "edu.brook.norms.Norms" should be placed at "res/edu/brook/norms/AboutNorms.html". The file should be an html fragment -- that is, without body and head tags -- and can include any valid html tags, including links.

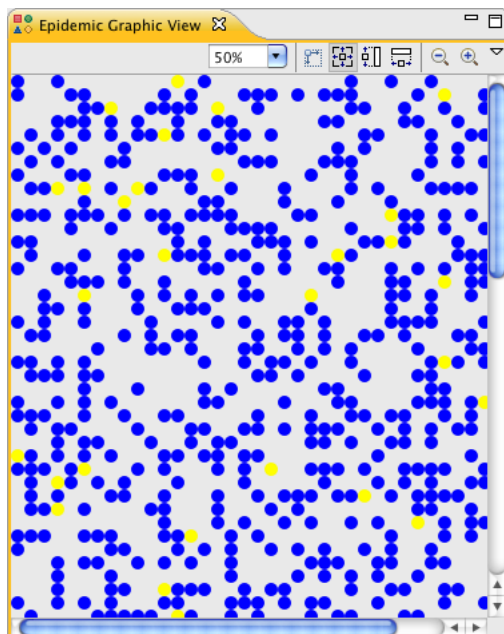


3.4.5. Visualization

The Agent Modeling Environment is designed to support many kinds of 2D or 3D visualization. AMP includes the following views for generated models. The Escape API supports a number of additional visualization to support Ascape models that aren't shown here. Look at the example org.ascape.escape.models.examples and org.ascape.escape.models.brook for examples of those.

3.4.5.1. 2D Views

The **Graphic 2D** view is the most common view way to work with an Agent Model and is automatically generated and displayed for executing models.



There are a number of view widgets in the upper-right hand corner that you can use to modify the view. You can:

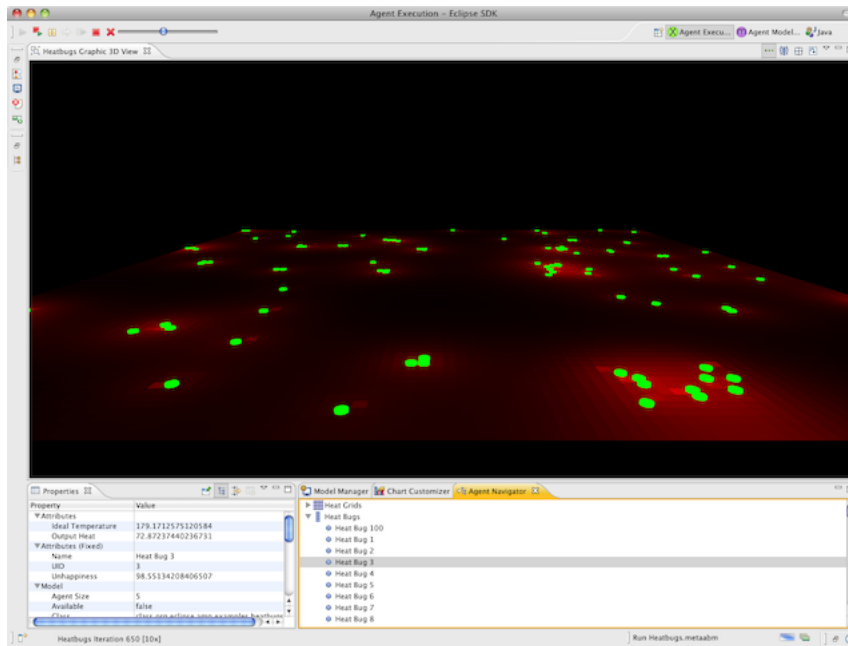
Scaling

1. Enter or select a specific scale in the combo menu.

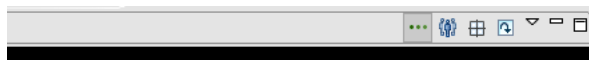
2. Select a button to have the model scale either:
 - a. Freely
 - b. Within the bounds of the view
 - c. Within the vertical bounds of the view
 - d. Within the horizontal bounds of the view
3. Zoom In
4. Zoom Out

3.4.5.2. 3D Views

The 3D views provide a three-dimensional representation of a running model. Currently they support 2D models only, which makes them 2 1/2 D views.



You can navigate and customize the model with the following controls:



From left to right, they are:

Animation

Turns on and off the interpolated (smooth) movement of agents from one cell to another. This is a unique feature of AMP and provides a very nice visualization, but it does slow down model execution. For very large models, you can switch it off -- visualization will be "jerky" but much quicker.

Perspectives

The three perspectives are used to control the camera location -- in other words the perspective from which the movie is being made. Note that if the camera is currently moving to a given perspective and you select a new perspective the camera may become confused! It is best to wait until the camera comes to a stop before choosing a different perspective. You can speed up camera movement by pausing the model temporarily.

First Person

Moves the point of view to ground level, as if the observer were in the space itself.

Overhead

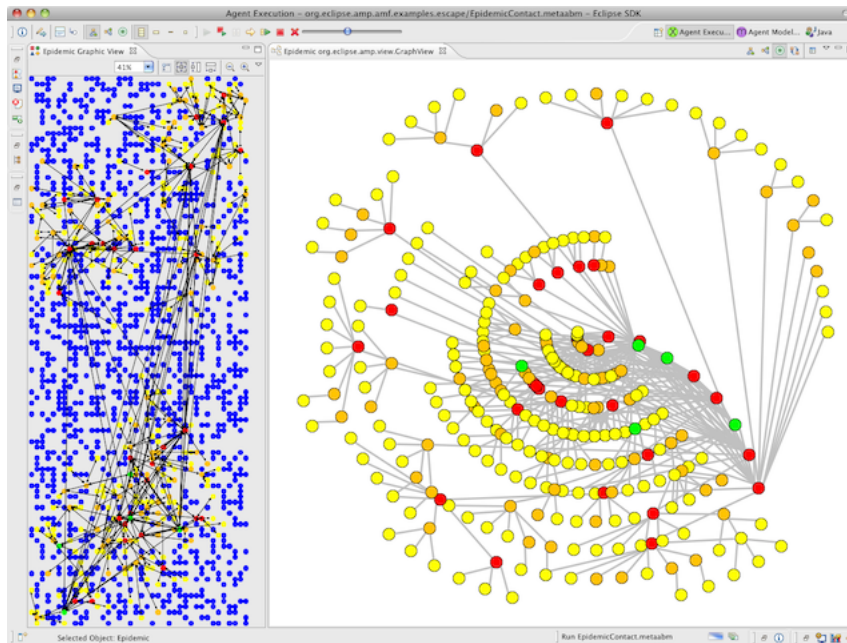
Moves the point of view to overhead, giving a similar view as you have with the 2D view. Note that in many cases the 3D view is actually faster than the 2D view, so this is a good way to observe any kind of 2D model.

Helicopter

Moves the point to an oblique perspective. This is useful for getting an overall sense of model behavior.

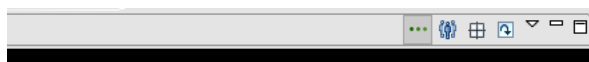
3.4.5.3. Graph Views

Graph views allow you to understand the network relationships between various agents in the model. By default, a graph is created for all relationships, but you can customize that behavior programmatically. Graphs are also super-imposed on 2D models, as we can see in the following example running the EpidemicContact model.



Note that complex models can take a long time to visualize and we'll be looking for opportunities to optimize performance further in the future. If you think the visualization is stuck it is likely that it is simply calculating the next visualization step -- wait a bit before canceling the model or closing it.

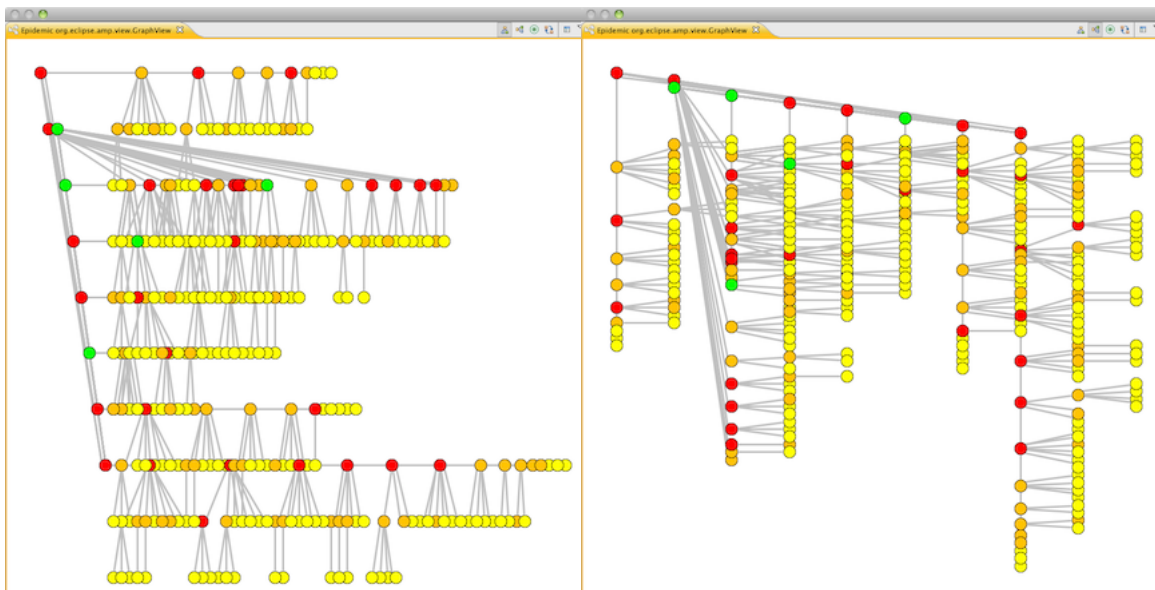
Graph views can be customized with the following controls:



From left to right, they are:

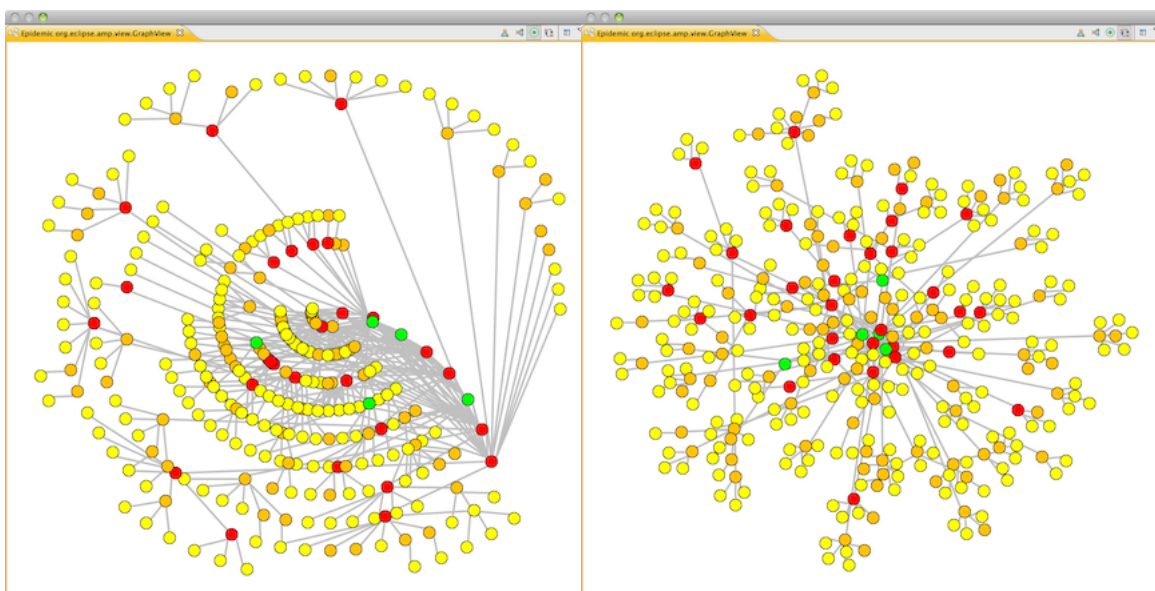
Layout

The layout buttons control how the nodes are placed within the graph. You can switch between layout dynamically to gain insight into network relationships.



Tree (Down): Places the graph nodes into a downward oriented tree formation. You'll find that the tree views are more efficient at visualizing, and can be very good at representing models with an inherent hierarchical structure such as kinship diagrams, but don't always give the best insight into the model for complex network relations.

Tree (Right): The same tree layout oriented from the root-most nodes rightward.

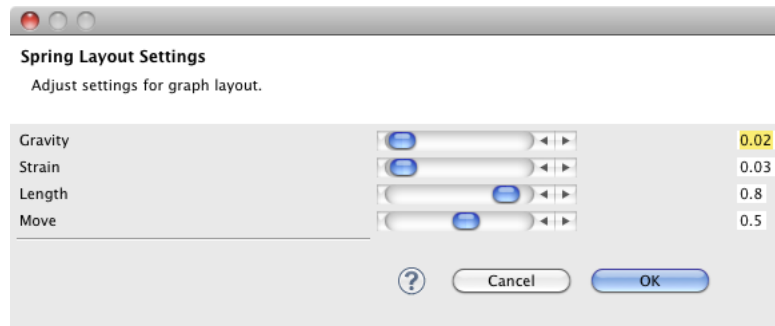


Radial: Places the nodes in a radial layout with the root-most nodes at the center. This is a great way to represent semi-hierarchical models with complex structure, but is a little slower than the tree layouts.

Spring: This layout uses a spring and strain approach that allows nodes to find their own location within the visualization. It can provide beautiful and insightful diagrams, but it is also slower than the other layouts. For more complex models it often works well to use the radial layout and then switch to the Spring layout when examining relationships in depth.

Spring Layout Customization

You can customize how the Spring by clicking on the customization button. This allows you to change a number of values that determine how nodes are layed out. While the default settings work well with many graph structures it can be helpful (and fun) to play with different settings to get the best visualization. |



3.4.5.4. Charts

A **Chart** view is automatically created and displayed for executing models. It can display aggregate values for any of the agent attributes you have set the "gather data" value to true for. Charts can be easily modified. While the built-in view is not meant to be the sole tool for Escape model data analysis, it provides an easy to use and powerful way to explore models interactively. The Chart view widgets allow you to modify the chart with the click of a button.

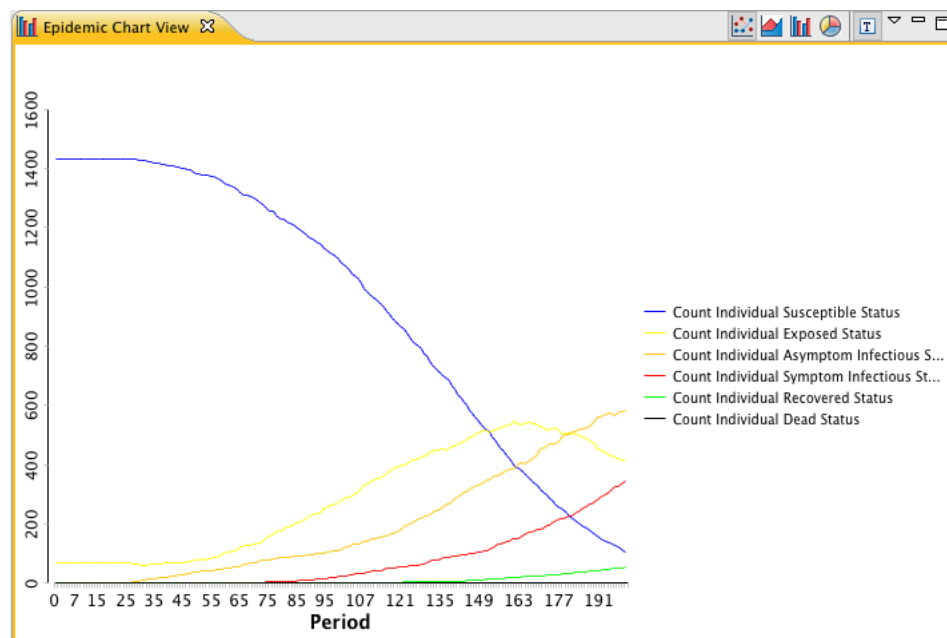
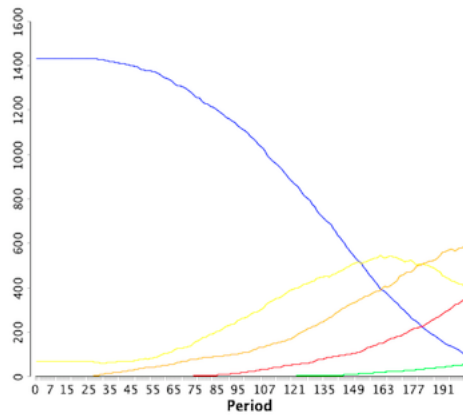
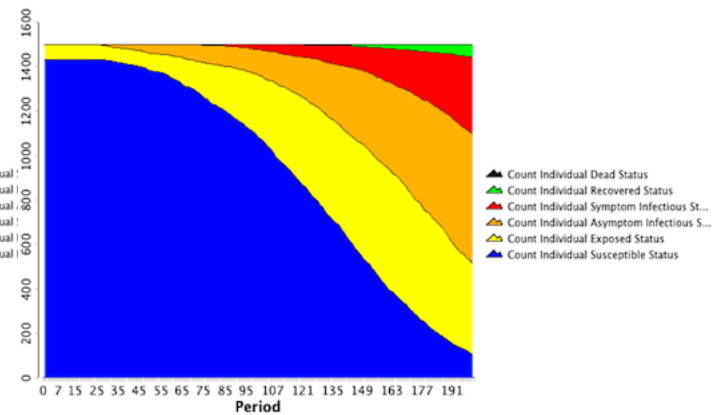


Chart Type

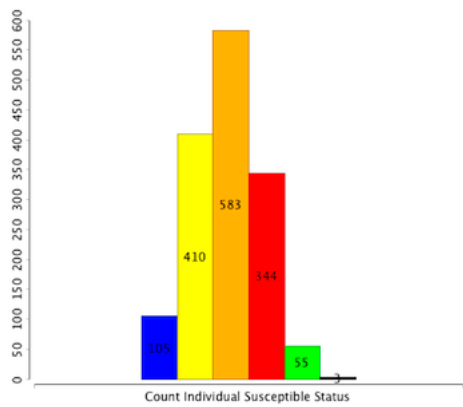
Several chart types are supported: The line, area, bar and pie chart. Click on one of the icons to select that type.



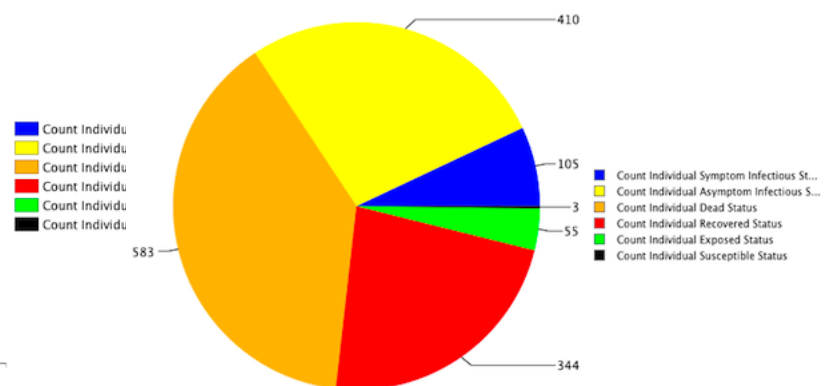
Line



Area



Bar



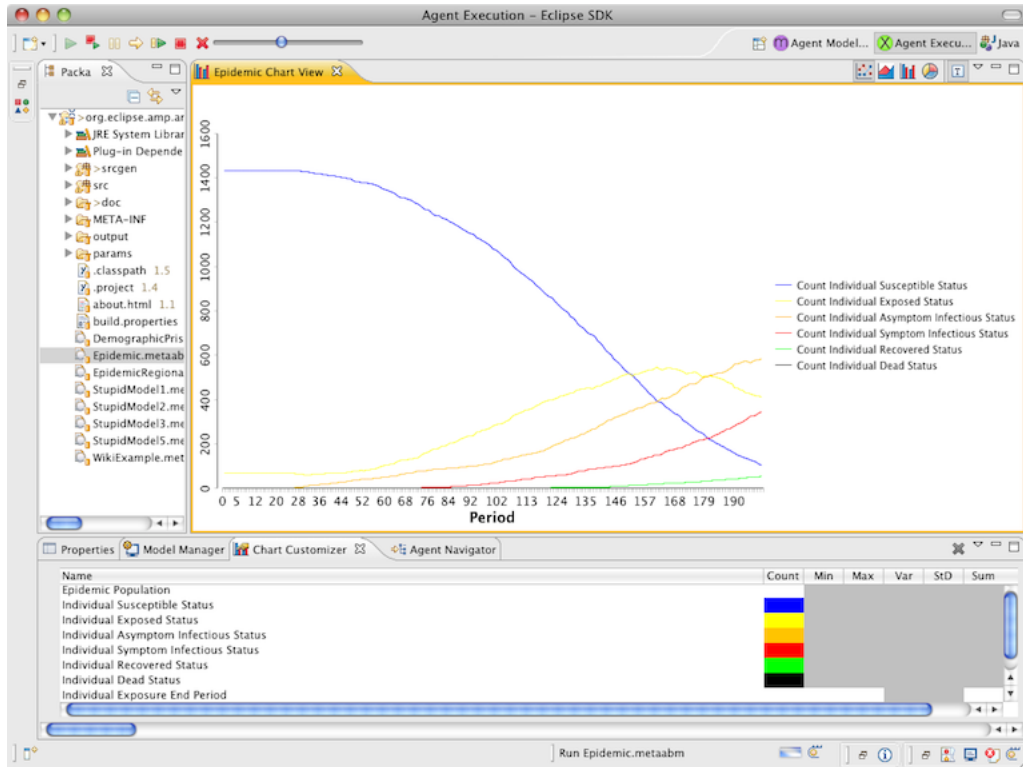
Pie

Chart Legend

Turn the legend on and off by selecting the "T" icon.

Chart Data

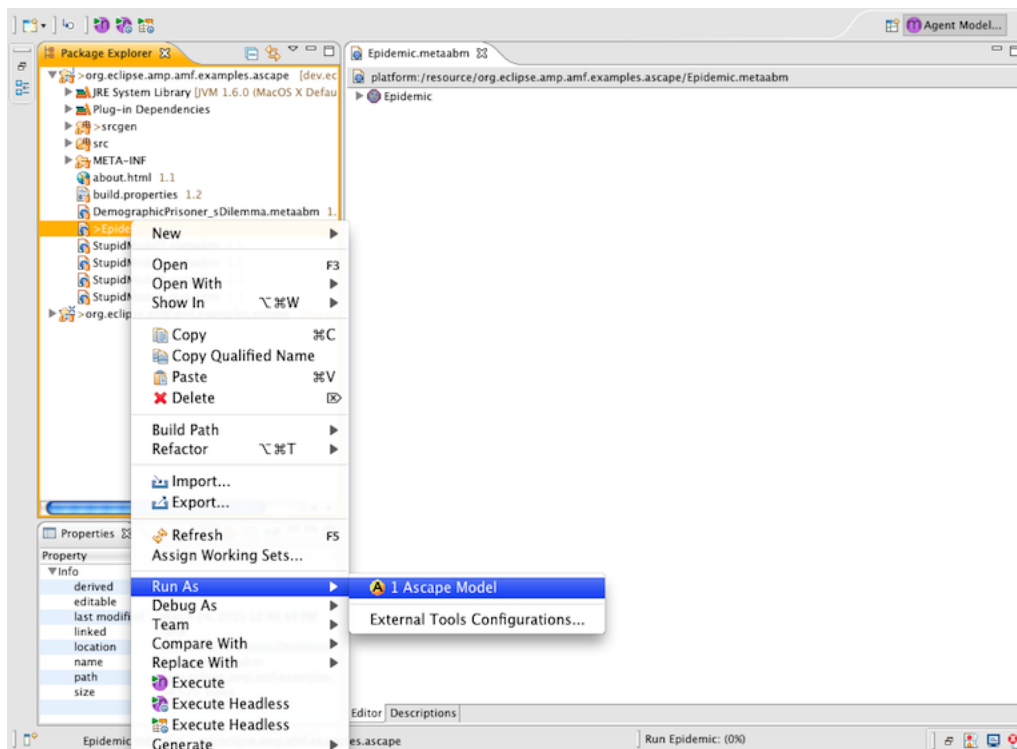
To select the statistics to display, show the **Chart Customizer** view. When the customizer has been selected, select a chart to customize. The customizer presents the possible choices in a 2D format. On one axis are the attributes that have data being collected, and on the other are the measurements collected on those axes, i.e. Count, Minimum, Maximum, Variance, Standard Deviation, Sum and Average. To clear all selections, click the Gray "X" button.



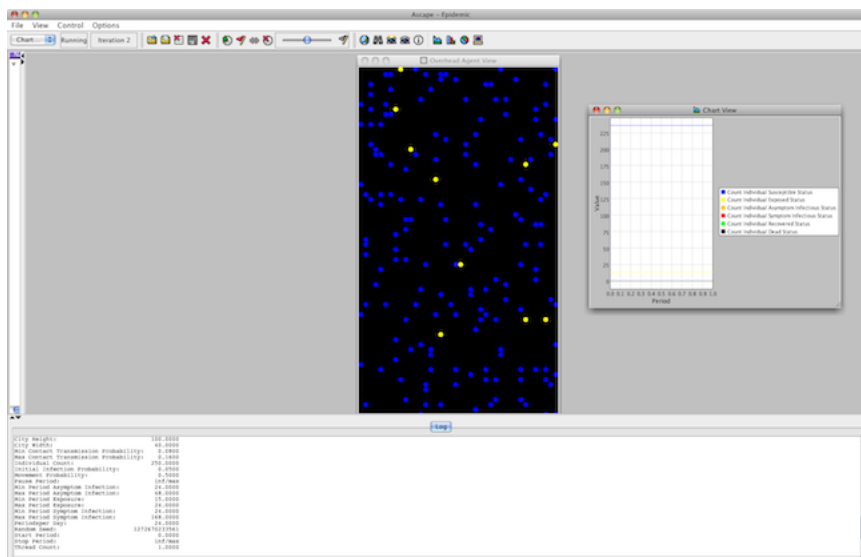
There are a number of other things to play around with, such as zooming the agent view or selecting other chart series to display using the Chart Customizer, so just explore. You can always close an active model by clicking on the close toolbar button. Or if you can't access the models controls for some reason, you can open the progress view and close projects from there.

3.4.6. Launching Other Targets

To execute applications for other targets such as Ascape or Symphony, just right-click on the metaabm model, select Run As.. and pick the target you want to launch. In the example below, we're launching Ascape from within an ..ascape project.



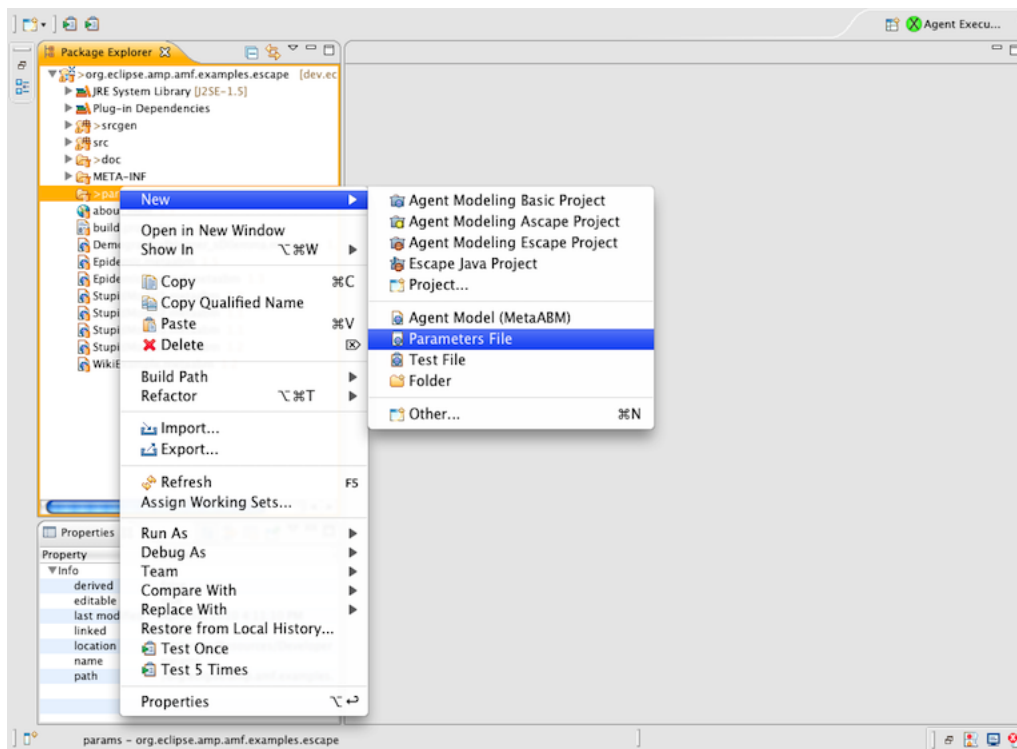
The external tool you've selected will then open into a separate Java application.



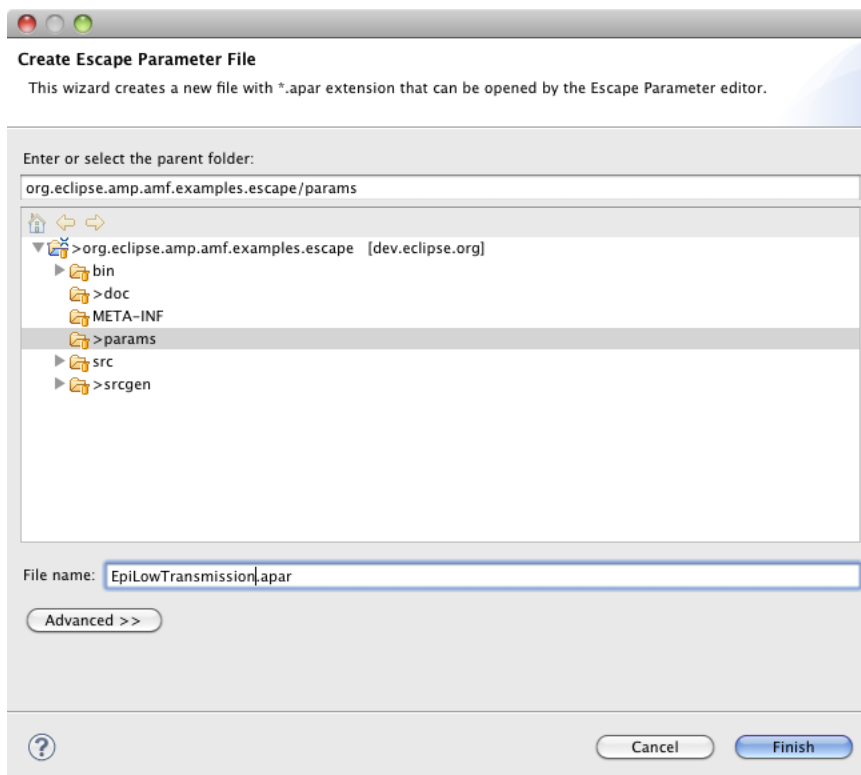
Note: remember that the menu options are active regardless of whether the target is actually supported within a given project. If you attempt to Run (external targets) or Execute (internal targets) models from a project that does not support that target you will get an error!

3.4.7. Model Parameterization

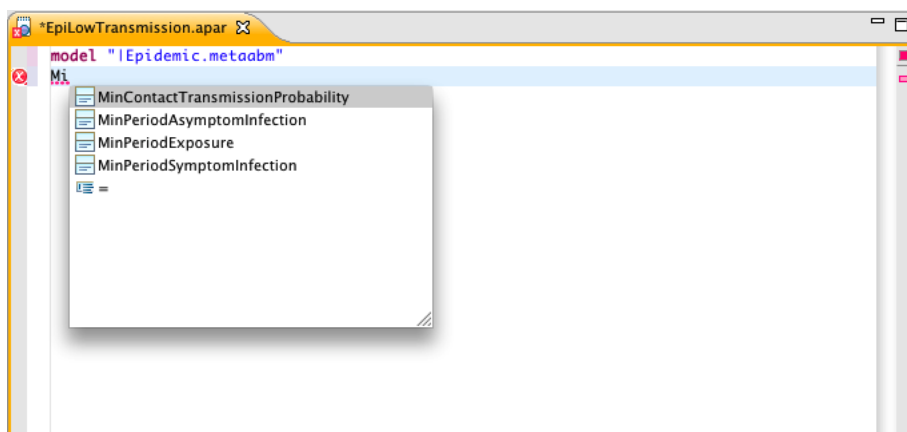
Escape provides full support for parameterization of models. This allows you to create many sets of parameters for a given model, allowing you to decouple model runtime settings from the model itself. (Support for parameter sweeps is forthcoming.) To create a new parameterization, right-click in the location you want to create the parameter file, and select **New > Parameters File**.



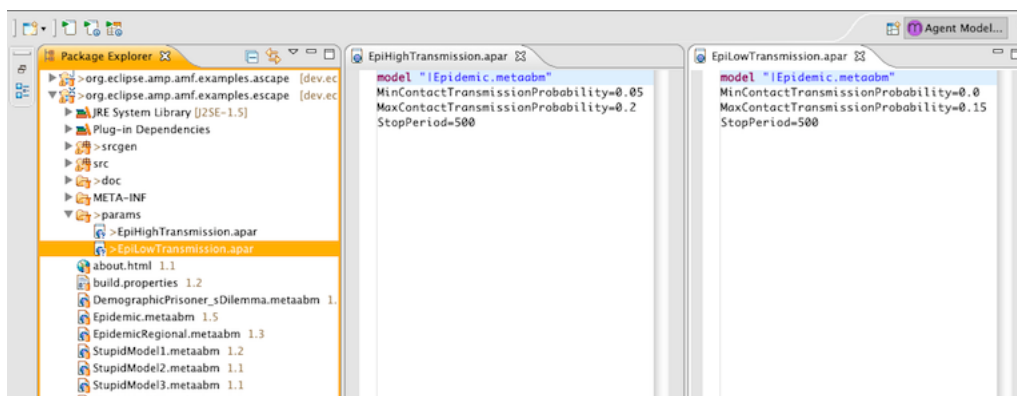
In the wizard that follows, give your parameter file a name:



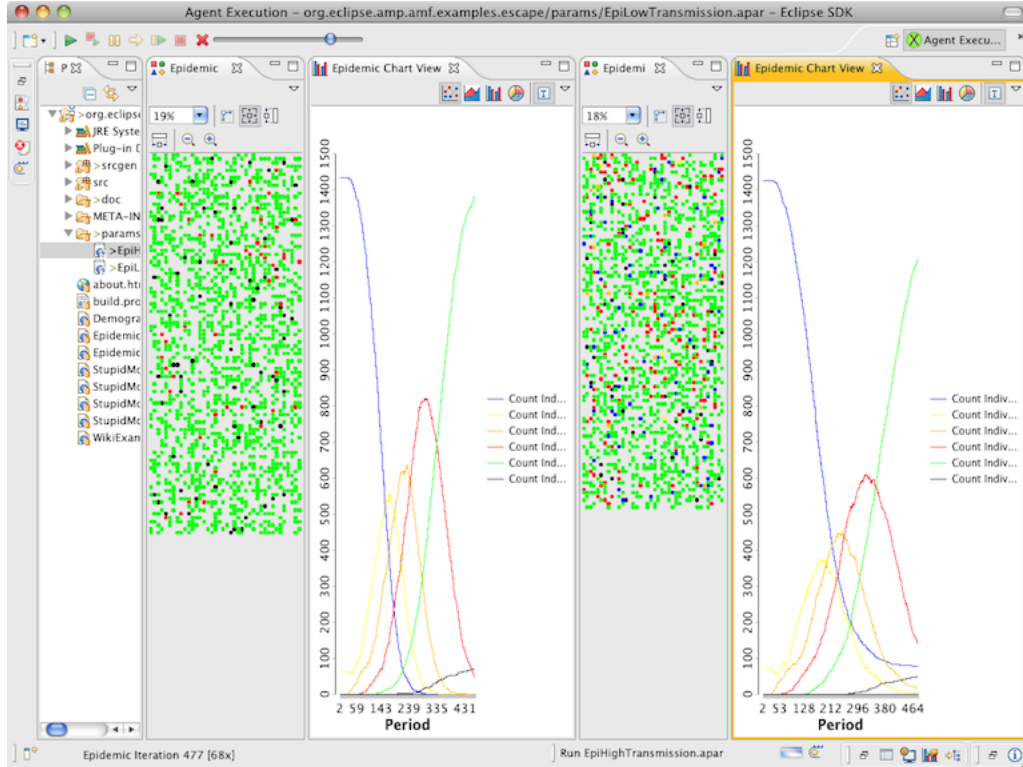
The edit the file. See below for file syntax. The parameter editor has built-in support for code-completion, syntax high-lighting and other editor features.



In the example below, we've created two separate parameter files for the epidemic model with different values for contact transmission probability.



As with other runnable files, you can launch a parameter file simply by selecting execute in the popup menu, application menu or toolbar.



3.4.7.1. Syntax

Parameter files are quite simple. Every parameter file specifies the "model" that will be executed. The "incorporates" keyword supports including parameter values from other files. File locations are relative to the location of the parameter file, but it is usually more flexible to refer to model files relative to their project location. You can do that by inserting a "|" character at the beginning of the file name.

Attribute values are specified by the attribute ID (with the first character in upper-case) followed by "=" and the desired value.

3.4.7.2. Example

We can create a base parameters file called "EpiBase.apar".

```
model "|Epidemic.metaabm"
StopPeriod=500
```

This means that we expect the Epidemic.metaabm model to be at the root of the project, and we want the model to stop at period 500. Then -- in this overly simple example -- we could create two other files, EpiHighTransmission.apar:

```
model "|Epidemic.metaabm"
incorporates "EpiBase.apar"
MinContactTransmissionProbability=0.05
MaxContactTransmissionProbability=0.2
```

and EpiLowTransmission.apar:

```
model "|Epidemic.metaabm"
incorporates "EpiBase.apar"
MinContactTransmissionProbability=0.0
MaxContactTransmissionProbability=0.15
```

Both of which reference the base set of parameters we've just defined and add run specific variables. Note that the incorporates reference is a parameter file relative reference so that we can easily move the set of

parameter files to any location we want. With these two files defined we can click on both of them at once, select the Execute button and immediately compare the two values. (Not shown.)

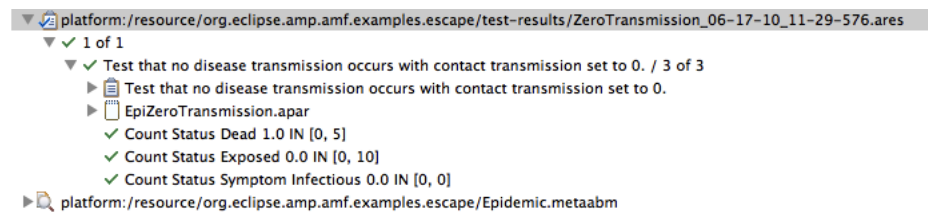
3.4.8. Model Testing

One of the most important and over-looked aspects of modeling is verification. Very loosely speaking, model *validation* is the process of determining whether a model matches up to the "real world". Model *verification* is the process of determining whether your model specification matches up to the model you've actually implemented. In other words, does the model do what you say it does?

To help you answer this important question, the modeling tools include unique support for an approach to validation and verification called "unit testing". A thorough discussion is far beyond the scope of this manual, but this guide should give you enough information to get testing your own models. Testing might seem tedious but it's actually a real time saver, because you can quickly run a set of tests to ensure that your new model change hasn't broken existing functionality. This peace of mind is priceless, and once you start testing, you won't know how you lived without it!

To create a new test, right-click at the location you want to put the test files. (Not shown.) Then we can edit the test file using the syntax shown below. It's helpful to keep them organized in sets of folders. To execute a test or set of tests, simply select the test files and click "Test Once" or "Test 5 times" to run a set of identical tests. The latter is useful for stochastic tests in which tests may create significantly different results for the same random seeds. (We'll provide more options in a future release.) If you select an entire folder, the framework will run all of the tests in the folder. This process is very similar to that for parameter files, so we won't show it here.

Once a test is complete, an ".ares" file is created in the "test-results" folder. To see how your tests fared, open the .ares file. For the example test file below, we should see something like this:



Notice that we can view the expected values and where our results actually fall.

3.4.8.1. Syntax

Test files are also pretty simple. It's a good idea to provide a description of the test so you can understand why you created it and what feature you're testing. Just put quote characters around your description. Then, use the "tests" keyword to specify what parameter file you'll be using. Again you can use file or project relative references. You do not specify a model here -- that's defined by the parameter file. You can use the "contains" keyword just like the incorporates keyword for parameter files, to specify a set of test constraints that you want to include with the new tests you've defined. This is useful for including a series of "sanity checks" with other tests. Then you specify the test constraints themselves. This uses the following form:

```
{Measure}({TestedAttribute}) = [{MinimumConstraint}, {MaximumConstraint}]
```

The measures available are Average, Minimum, Maximum Count and Sum. In order for an attribute to be tested the "gather data" value must be set for that attribute. All Measures are not appropriate for all attribute types. Booleans, Symbols and States should only use the Count measure as they represent discrete values that can either meet some condition or not.

The tested attributes are qualified by their model agent definition. The constraint definition is inclusive, as implied by the square brackets. Future implementations are expected to provide richer expressions and constraints but a very wide range of cases are supported by this construct, especially when combined with derived attributes. Tested Attributes are again specified by the attribute ID (with the first character in upper-case) followed by "=" and the desired value.

3.4.8.2. Example

In the following simple example we want to make sure that an epidemic can't occur without disease transmission taking place. Here we're just looking to make sure that we have some very basic logic right. (This might be almost too simple for a real-world design, depending on how rigorous we want to be.) Note that some individuals are infected at the beginning of the model run so we need to account for that in our expected results. So first we have defined the EpiZeroTransmission.apar file like the other transmission rate parameterizations above except with these settings:

```
MinContactTransmissionProbability=0.0
MaxContactTransmissionProbability=0.0
```

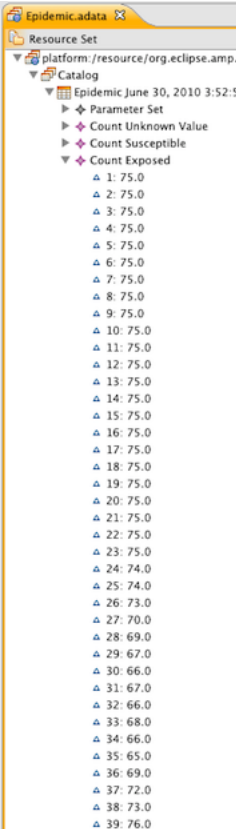
Then we define our ZeroTransmission.atest file:

```
"Test that no disease transmission occurs with contact transmission set to 0."
tests "|params/EpiZeroTransmission.apar"
Count(Individual.Status=Dead) = [0,5]
Count(Individual.Status=Exposed) = [0,10]
Count(Individual.Status=SymptomInfectious)=[0,0]
```

Here we have a nice description, a reference to the parameterization we want to use, and then the set of constraints we will apply to them. After running this test, a test result file like the one above will be created and placed in our "test-results" directory.

3.4.8.3. Model Data

As discussed above, you can collect data for any model by selecting it or a parameters file and clicking the "Execute Model with data collection" button. If you use a parameters file setting the StopPeriod parameter will cause the model to stop executing and save the model at the specified time. Or you can stop the model manually. The model data will be collected into an "adata" file and stored in the "output" directory in your project. A new run entry will be created for each execution of a model. We are currently developing tools for exporting this data to various file formats as well as providing more seamless integration to sophisticated Eclipse hosted charting and report tools. You can use BIRT to interact with the data set using an XML schema data source and there is also an Ecore driver available. Model data can also be produced by writing custom views for data output. See the Ascape programmer guide for more information on how to do that.



Time Step	Count Exposed
1	75.0
2	75.0
3	75.0
4	75.0
5	75.0
6	75.0
7	75.0
8	75.0
9	75.0
10	75.0
11	75.0
12	75.0
13	75.0
14	75.0
15	75.0
16	75.0
17	75.0
18	75.0
19	75.0
20	75.0
21	75.0
22	75.0
23	75.0
24	74.0
25	74.0
26	73.0
27	70.0
28	69.0
29	67.0
30	66.0
31	67.0
32	66.0
33	68.0
34	66.0
35	65.0
36	69.0
37	72.0
38	73.0
39	76.0

Chapter 4. Tutorials

4.1. Designing a Model

In this tutorial, we'll walk through the creation of a complete model. While the tutorial seems lengthy, that's because we've tried to be thorough in describing each step. As we demonstrate in an online video, the actual model can be created in just a couple of minutes.

4.1.1. Model Goals

Before beginning a modeling project it is important to have a clear idea of what is being modeled and how it should be modeled. We need to know a bit about where we are going before we start. For our target, we'll use a model described in a paper written by Railsback, Lytten and Grimm. They describe "Stupid Model", a model that could be used to provide a common standard to demonstrated features of different Agent-Based Modeling platforms. See <http://condor.depaul.edu/~slytinen/abm/StupidModelFormulation.pdf> for the complete specification. Here are the key specifications for the basic model.

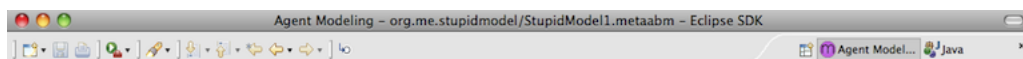
1. The space is a two-dimensional grid of dimensions 100 x 100. The space is toroidal, meaning that if bugs move off one edge of the grid they appear on the opposite edge.
2. 100 bug agents are created. They have one behavior: moving to a randomly chosen grid location within +/- 4 cells of their current location, in both the X and Y directions. If there already is a bug at the location (including the moving bug itself, bugs are not allowed to stay at their current location unless none of the neighborhood cells are vacant), then another new location is chosen. This action is executed once per time step.
3. The bugs are displayed on the space. Bugs are drawn as red circles. The display is updated at the end of each time step.

4.2. Model Implementation

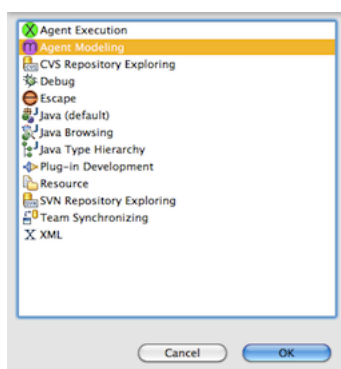
4.2.1. Setup

4.2.1.1. Open Agent Modeling Perspective

A special Agent Modeling **Perspective** can be used to setup your **Workspace** for modeling. Depending on your how you installed the agent modeling tools, you can simply pick the perspective from the choices in the upper-right hand of your workbench:

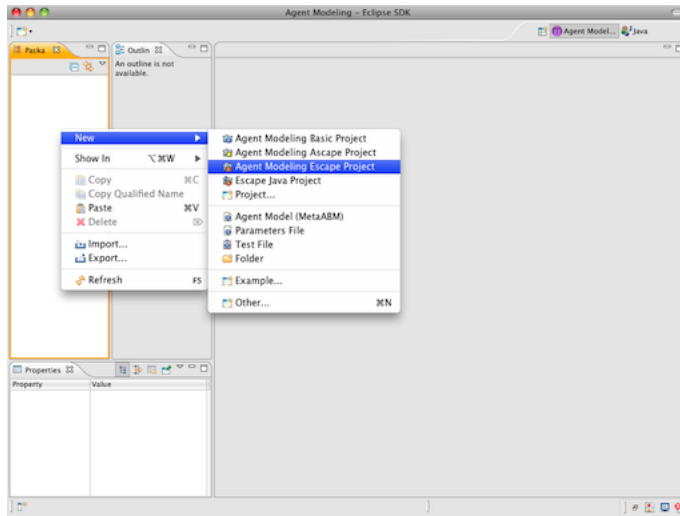


Or, if you can't see it there, you can access it by selecting the menu "Open Perspective > Other..." and then selecting the Agent Modeling perspective from the provided options:



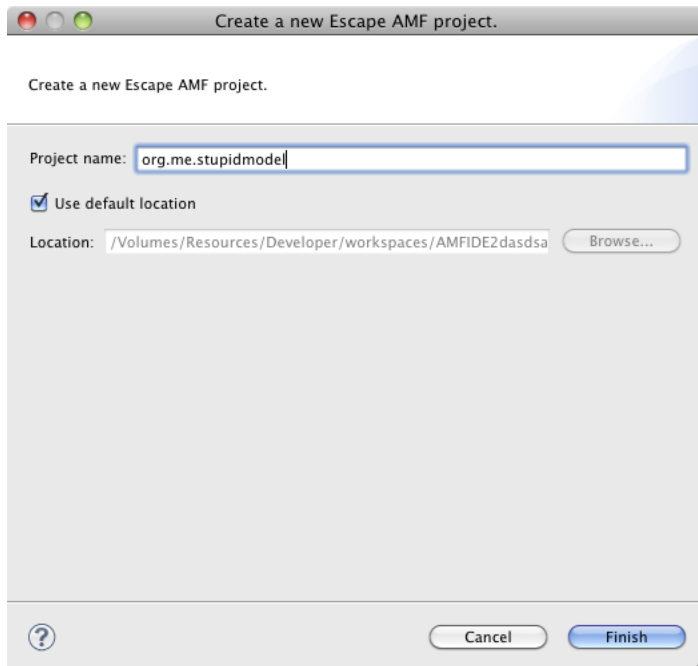
4.2.1.2. Create Project

Before doing anything else, we need a project -- a folder to organize our models within. Right-click in the Package Explorer and select **New > Agent Modeling Escape Project**.



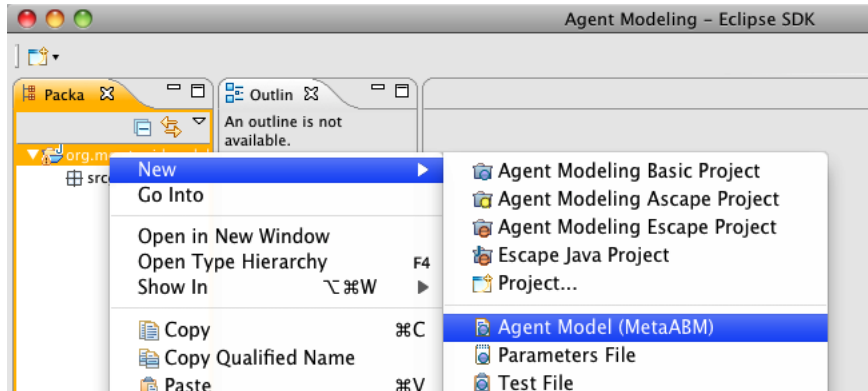
4.2.1.3. Name project

We'll get a dialog allowing us to name the project. Let's call it something simple.

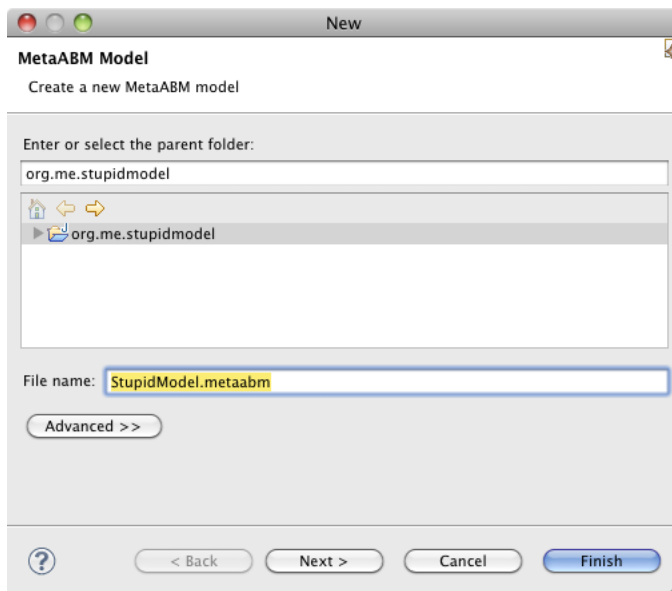


4.2.1.4. Create Model

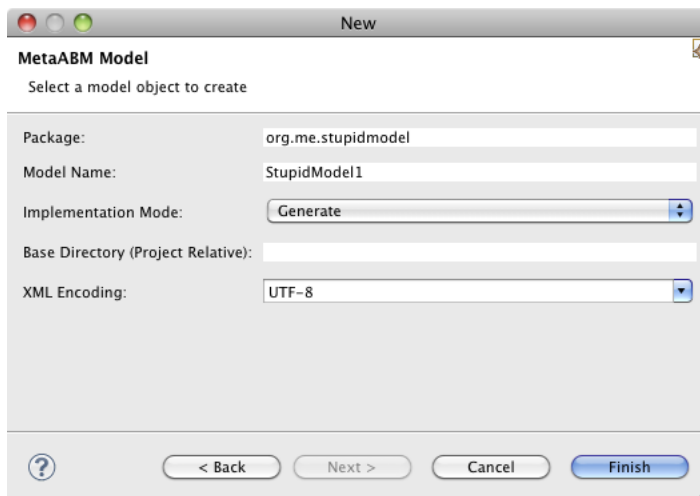
We'll need a model as well. This is the container for all of the the agents and behaviors we'll be defining. Right-click on the project folder and choose **New > Agent Model**.



You'll see a model creation wizard from which you can name the model. Just give it the name "StupidModel.metaabm".



Click **Next** to see some other options. We don't need to change any of this:



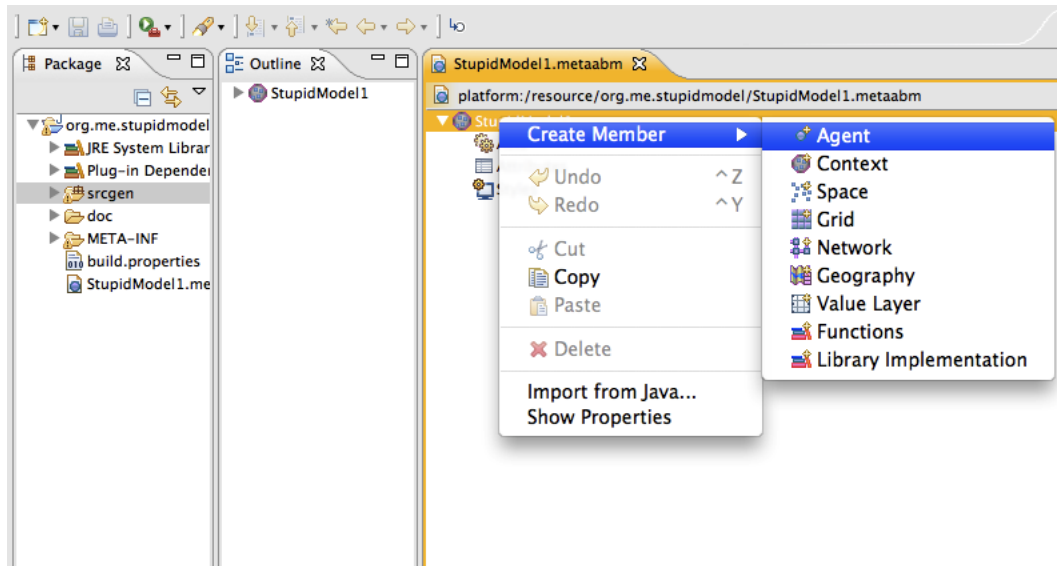
So just click the **Finish** button and we're ready to move on with the tutorial.

4.2.2. Structure

4.2.2.1. Bug Agent

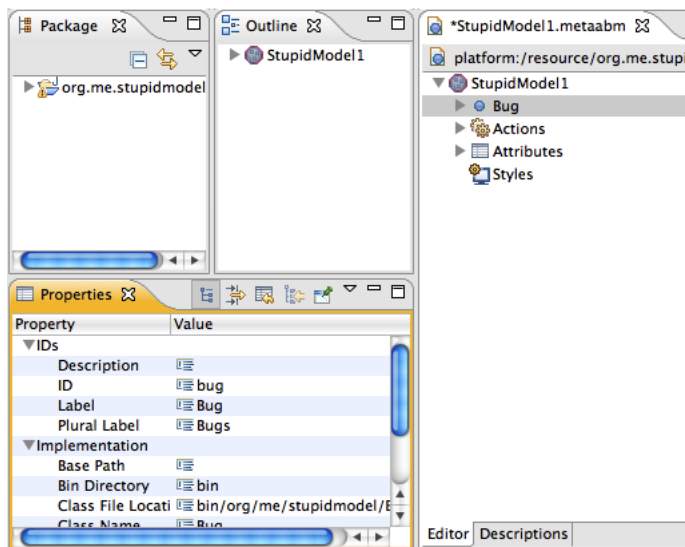
Create New Agent

Our first step is to create an agent. The root contexts and a number of necessary model components were already created automatically. So right-click on the root "StupidModel1" context and select **Create Member** > **Agent**.



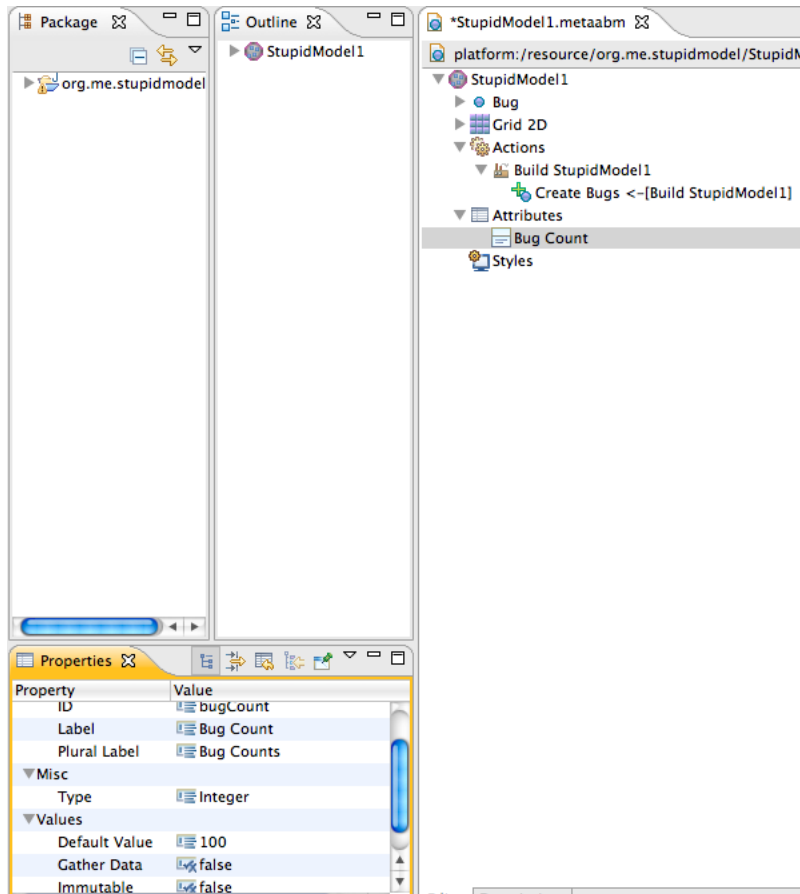
Name Agent

Let's call it "Bug". In the lower left-hand corner you should see the **Properties** View. This is where you'll edit model values when working with the tree editor. The properties view -- as with any other view --- can be moved anywhere within the workbench and if you can't find it -- or any other views -- you can always locate it through **Window > Show View > Other**. Find the "Label" property and type our new name into it. You'll see that all of the related name properties are also updated.

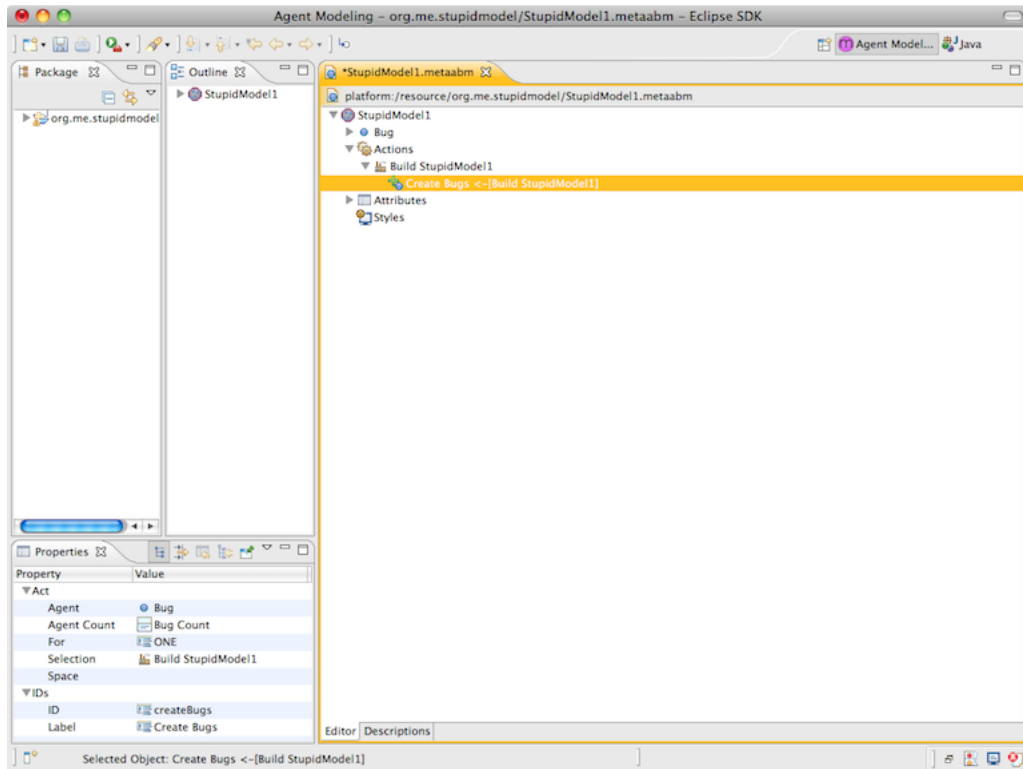


The Create Agent Action

While we've been making these changes, the modeling framework has also been doing some work behind the scenes. For example, it has created an attribute and a rule to create the agents. Open the Actions node and the Attributes node to see these changes. If we look at the attributes for the model, we can see that an attribute specifying the number of agents has been created called "Bug Count". Select it, then enter "100" as the default value. The default value is the value the model will use if no value is specified anywhere else (like in the parameter launcher).



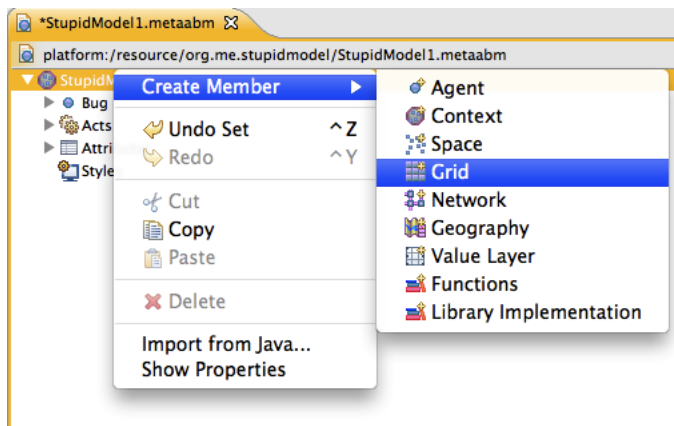
You might want to take a look at the Create Bugs rule, but there isn't anything we need to change there.



4.2.2.2. Grid Space

Create Grid

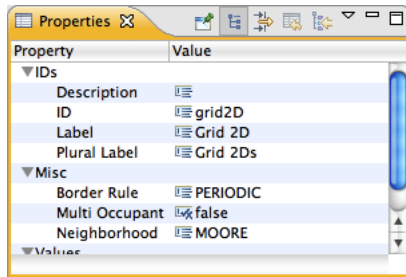
Next, we'll create a grid for the Bugs to move around upon. Right-click on the StupidModel1 context and choose **New > Grid**.



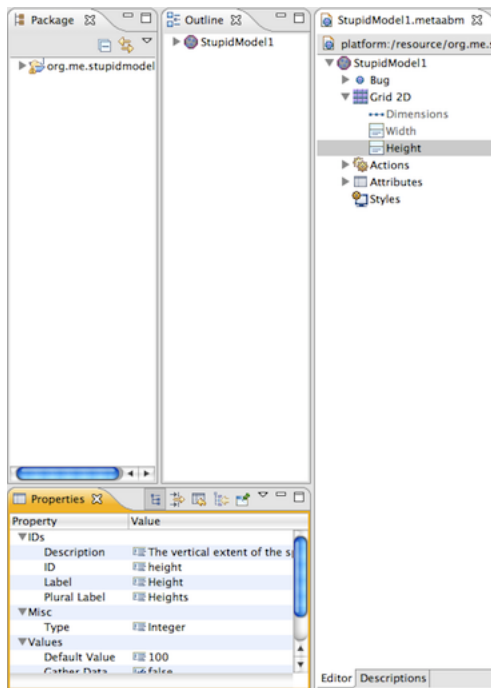
Set Grid Parameters

Now, looking at the properties for the Grid we've just created, we can see that there are a number of properties to set. We want to make a number of changes.

1. Set the space's "Border Rule" property to be "Periodic" -- the edges wrap around from one edge to the other.
2. Set the "Neighborhood" property to be "Moore" -- that's how we are going to interpret the statement "in the X and Y directions".
3. Set the dimensionality to "2". (You might need to enlarge the Property view or use the scroll bar to see the Dimensionality property.)

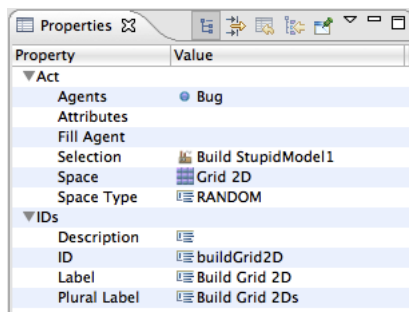


Again, the framework has taken care of some things for us automatically. If we now open the Grid node, we can see that "Width" and "Height" attributes have been created based on the dimensionality we've just defined. We'll set each of them to "100", so that we end up with a 100 x 100 grid. (Which is pretty large for a model of this kind, but we want to go with the basic specification.)

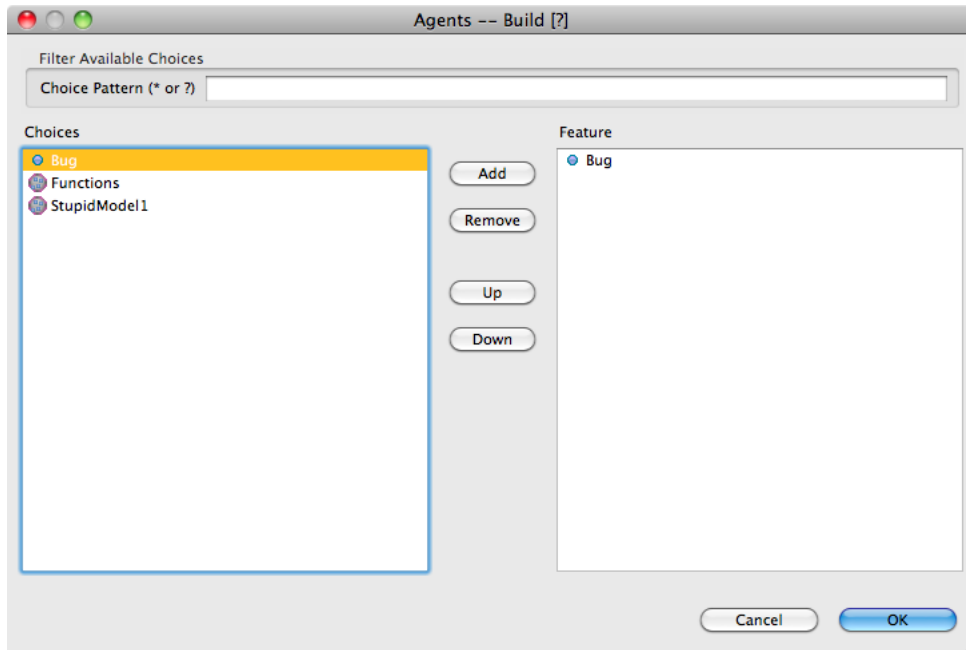


The Build Grid Action

The Agent Modeling Framework has also created a Build Grid Action for us. If you navigate to the StupidModel1 context Actions node again, you can see the Action. If you click on it you should see the following properties set:



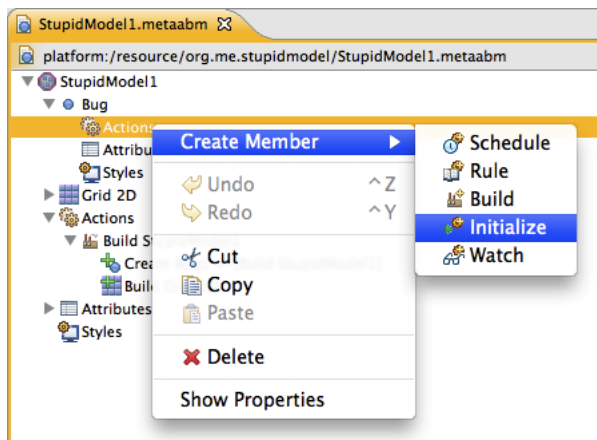
If you click on the ... button next to the "Agents" property you'll see the Bug, which is the agent that will be moving around upon the grid.



4.2.3. Actions

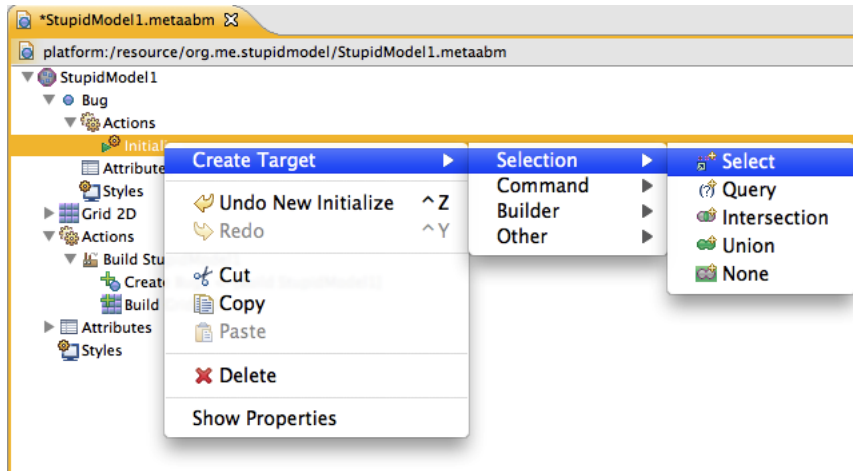
4.2.3.1. Create Initialize Rule

Now we can make these bugs do something. Let's create our first rule. In this case we're going to create a special Initialize Action that executes once when the model is first created for each agent. All Actions begin from the agent (or style) Actions list and there are only a few Actions that can serve as roots. See the Design section of this guide for much more on that. Open the Bug node by clicking the triangle icon, right-click on the Actions node and select the "Create Member > Initialize Action.

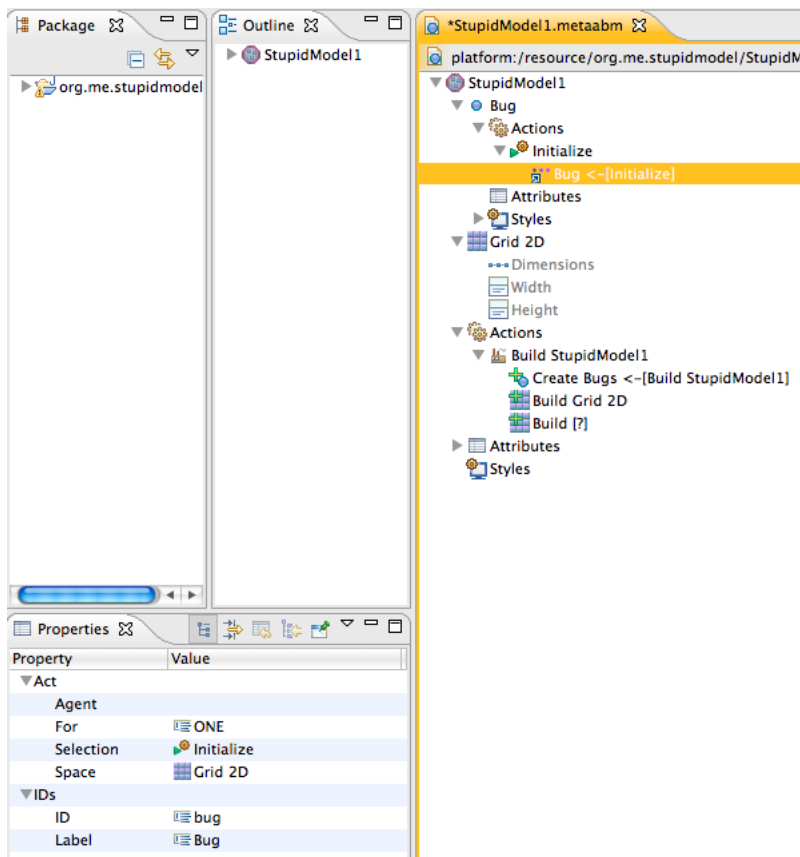


Create Select Action

Next, right click on the new Initialize node and choose **Create Member > Select** to create a Select target.

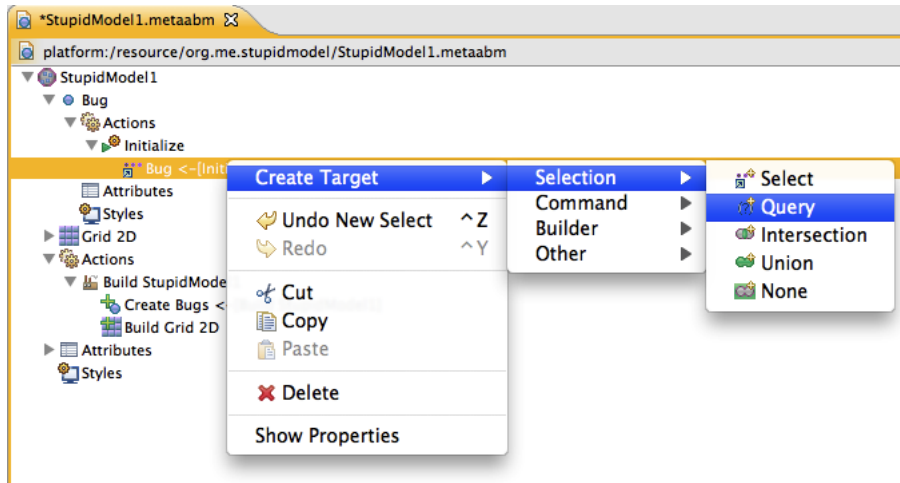


Selects are the central building blocks of model actions and define the basis for Query and Logic Actions that can be modified and used in subsequent targets. Targets are actions that "follow from" other actions. With this Select we are going to be searching for places for the Bug to move to at the very start of the model run. Here we need to specify the space we will be looking for cells within, so we set the Space property to "Grid 2D". Normally, we'd specify an agent as well, but as the Grid 2D cells don't have any attributes, we don't need to define them explicitly. So we just need to make sure that the Agent property is null (blank).

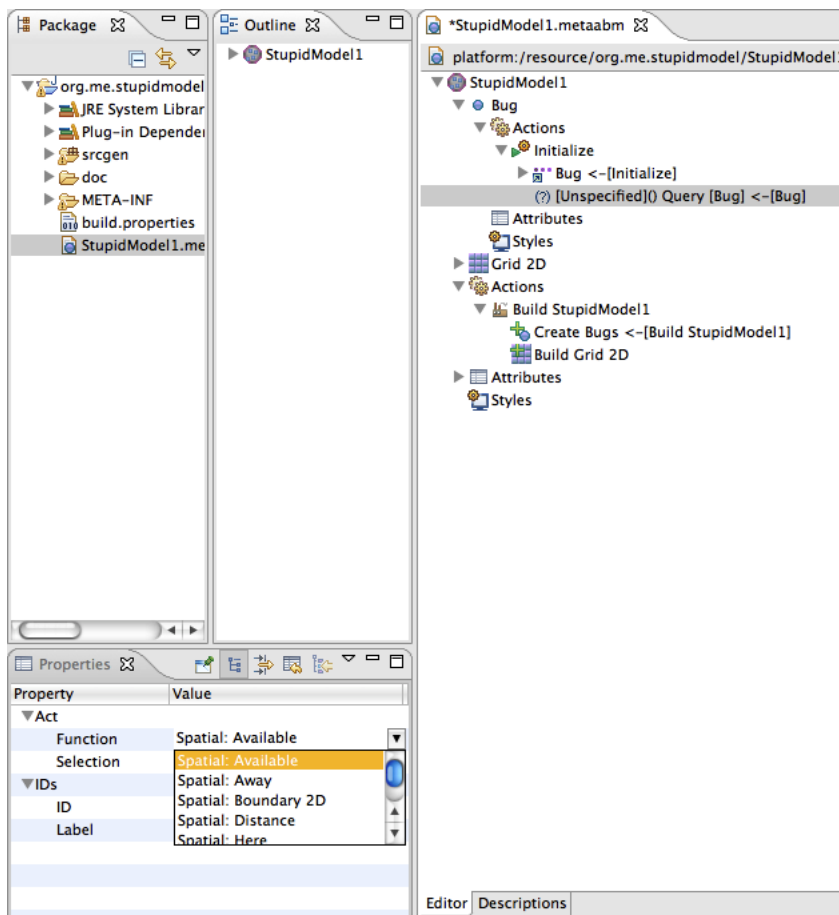


Create Query Action

Next, we create a Query Action. A query is really a query term; we're specifying some aspect of the search that we're interested in.

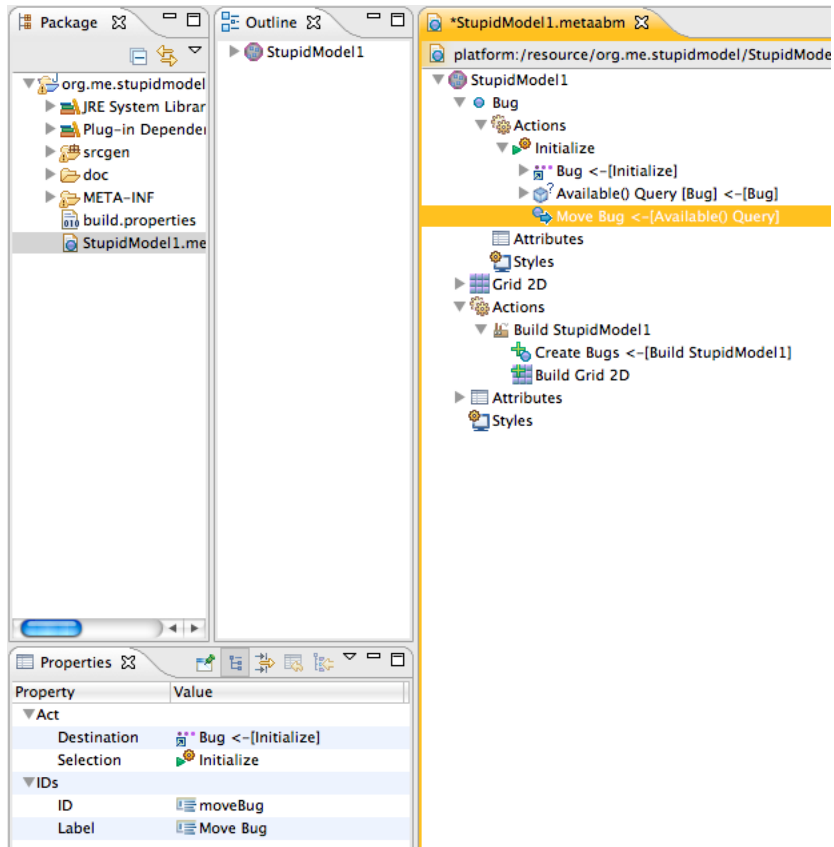


In this case we want a special kind of query, a query within space for any agents that are available; that is unoccupied by another agent. So we select "Spatial: Available" form the drop down list of query functions.



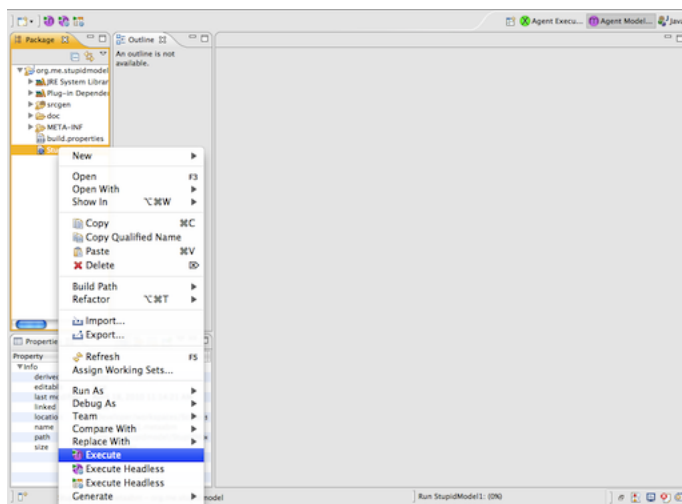
Create Move Action

Finally, as the last part of the initial action specification, we'll create a Move Action using **Create Member > Move**. (Not shown.) The properties should be correct, but check to make sure that the selection property is "Initialize" (the root), and the destination property is "Bug" (the Select Action we've defined above). Like this:

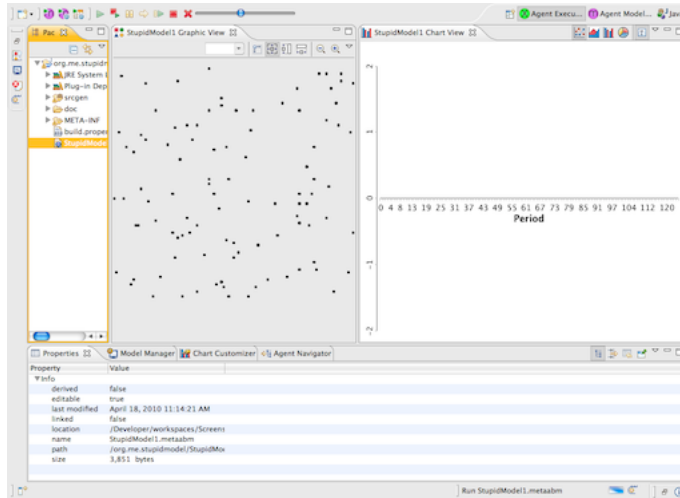


4.2.3.2. Run Initial Model

That's all we have to do to create our first working model! Now, go to the **Package Explorer** and select the `StupidModel1.metaabm` file. Right-click on it and select **Execute**. Or click on the first "m" button in the toolbar. (Again, we've simplified the toolbar for this tutorial, so you may have more buttons appearing than we're showing here).



A new perspective for "Agent Execution" will automatically open up and you can see that the agents have placed themselves in the grid. They don't do anything else yet.

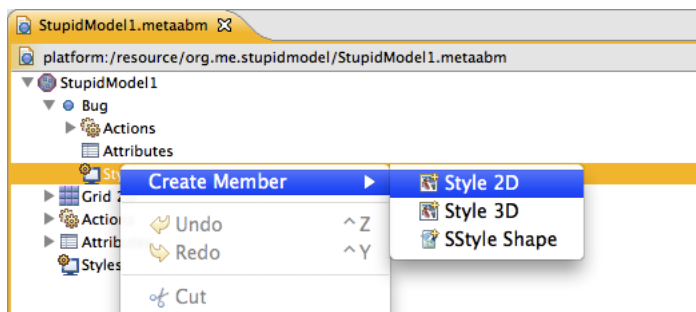


4.2.4. Styles

4.2.4.1. Create Bug Style

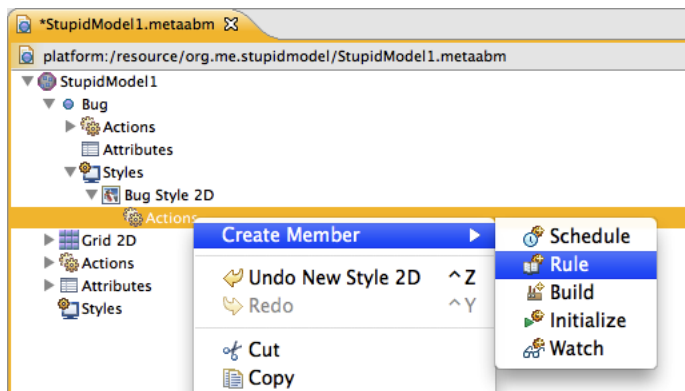
Create Style

While Escape assigns a default color of black for the agents, we'll want something more interesting -- so let's make the Bugs red. Agent visualization is accomplished using the same mechanism as ordinary agent rules. So first, let's create a new Style. Right click on the "Styles" node and select "



By default the style will be called BugStyle2D. Open the Style to the Actions node and create a rule.

Create Style Rule

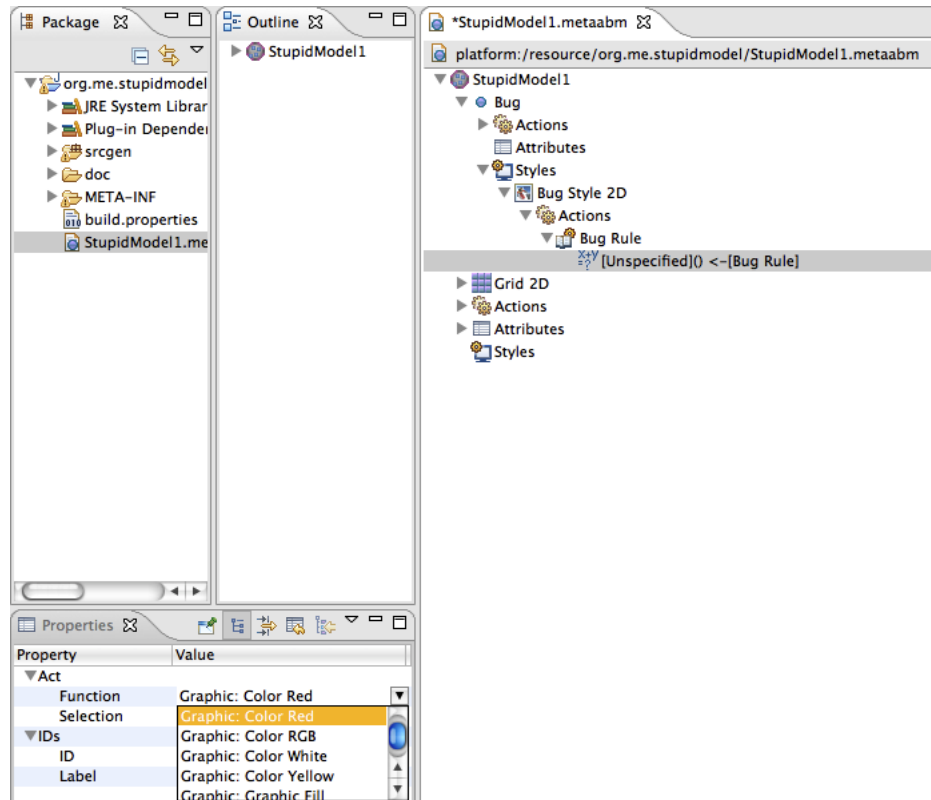


Now we need to create a series of evaluations. An Evaluation is like a Query in that it performs some kind of function in the broadest sense, but unlike a Query, it does not affect whether target (downstream) Acts

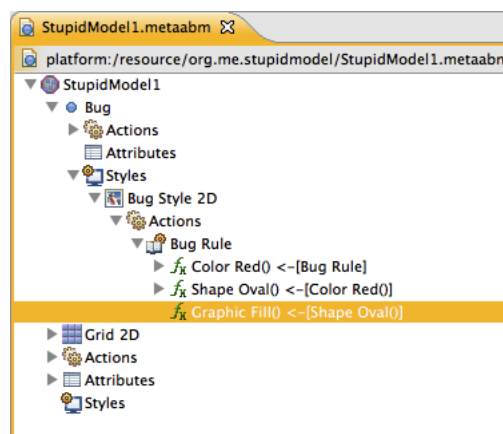
are performed. For Styles we have a set of Graphic functions that we will chain together to draw a figure. First we create the Evaluation.

Create Evaluations

We make the first Evaluation define a Color of "Red". Right click on the new Rule, and select **New > Command > Evaluate** (not shown) to create an Evaluation. Then in the function property, select "Graphic: Color Red".



We'll create two more evaluation targets. Right-click on the "Color Red" evaluation and create an Evaluation. Pick the "Graphic: Draw Oval" fill. For the last part of our style, we need to actually draw the figure. To do this we create a last Evaluation target for "Draw Oval" and give it the "Graphic: Fill Shape" function. By now it should be clear how to do this. "Fill" or "Outline" will always come last in a chain of graphic evaluations, but otherwise the order shouldn't matter. See the Demographic Prisoner's Dilemma model for an example of how this can be used to selectively pick Colors and or Shapes. You should end up with something like this:



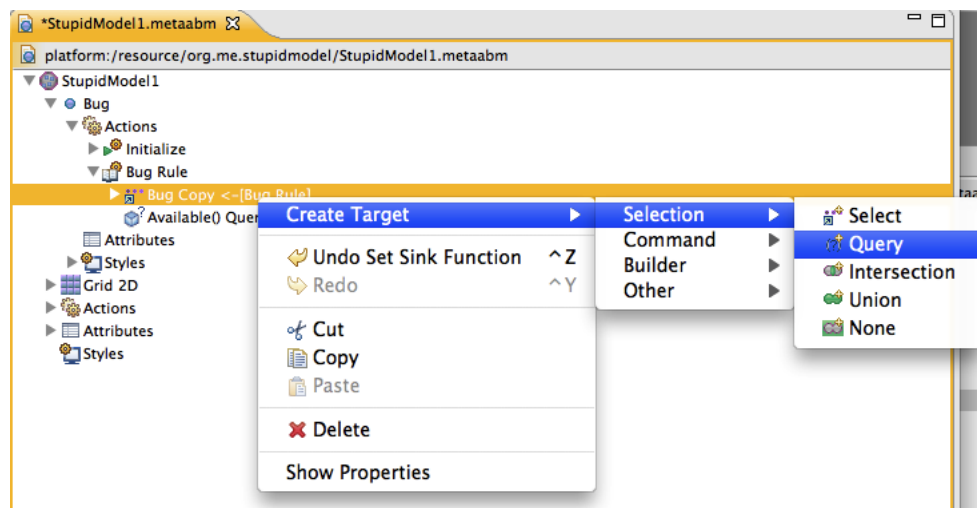
4.2.5. Actions 2 Movement Rule

To make our agents move randomly every iteration, we will create a Rule. (For those familiar with Repast or other scheduling oriented AM platforms, a rule is just a schedule that gets activated each and every period for all agents.) At this point in the tutorial, we'll show screenshots only for aspects of the modeling process that haven't already been covered.

Create Select and Query Actions

The first part of the Bug Rule is exactly the same as the Initialize rule. Create a Selection Action. As before, we'll make the Select statement "Agent" property blank, as we want to move to a cell. As before, we want to make sure the Cell is free before we try to move in, so we'll select a "Spatial: Available" Query function. (By the way, to make scrolling through the list easier, you can type the first letters of the selection, in this case "SP".)

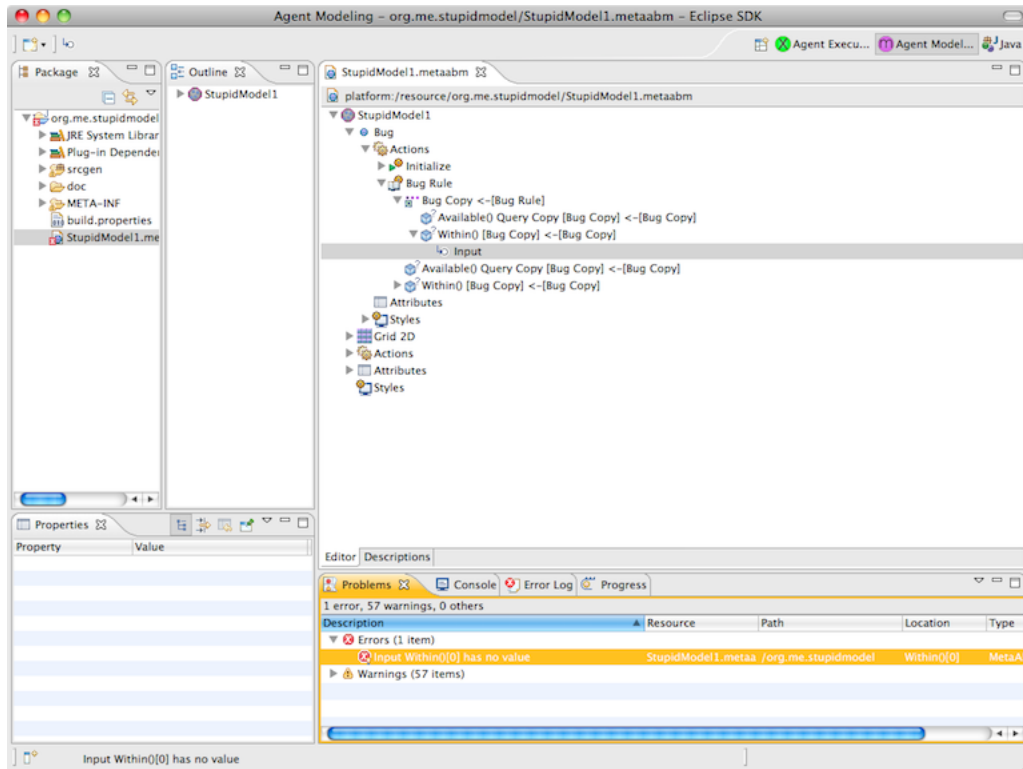
But now, instead of moving *anywhere* we want to move nearby. So now, we create a *parallel* or "sibling" Query from the same Select root. Because this new Query Action is a child of the Select Action and not the "Space Available" Query we've just created, we'll right-click on the *Select* Action and choose **Create Target > Select > Query**. Like so:



Often, we would pick the "Spatial: Neighbor" function to take a random walk, but according to the specification, we actually want to move into some random Cell within distance four (4). So we will pick "Spatial: Within" from the list of functions.

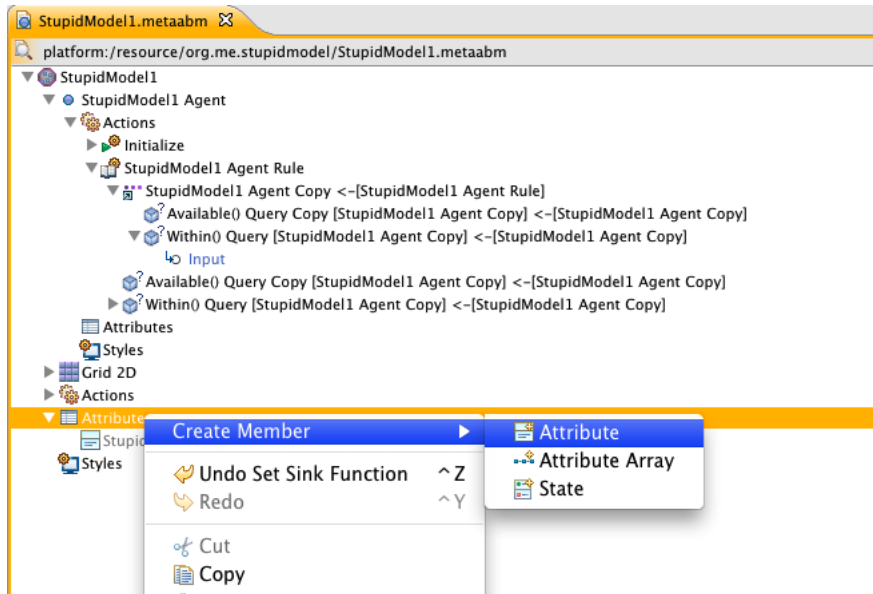
Model Error Handling

Instead of specifying the actual distance now, let's see how the Agent Modeling Framework error checking capabilities can help us in model development. Save the model by selecting **File > Save** from the application menu. Notice that a red marker appears next to the StupidModel1.metaabm file. If you reveal the **Problems View** you'll see a list of current errors and warnings and if you open the errors node in that list, you'll see an item reporting that the input has no value. If you double-click on that item, you'll be taken to the input for the Within query and you'll be able to edit it.

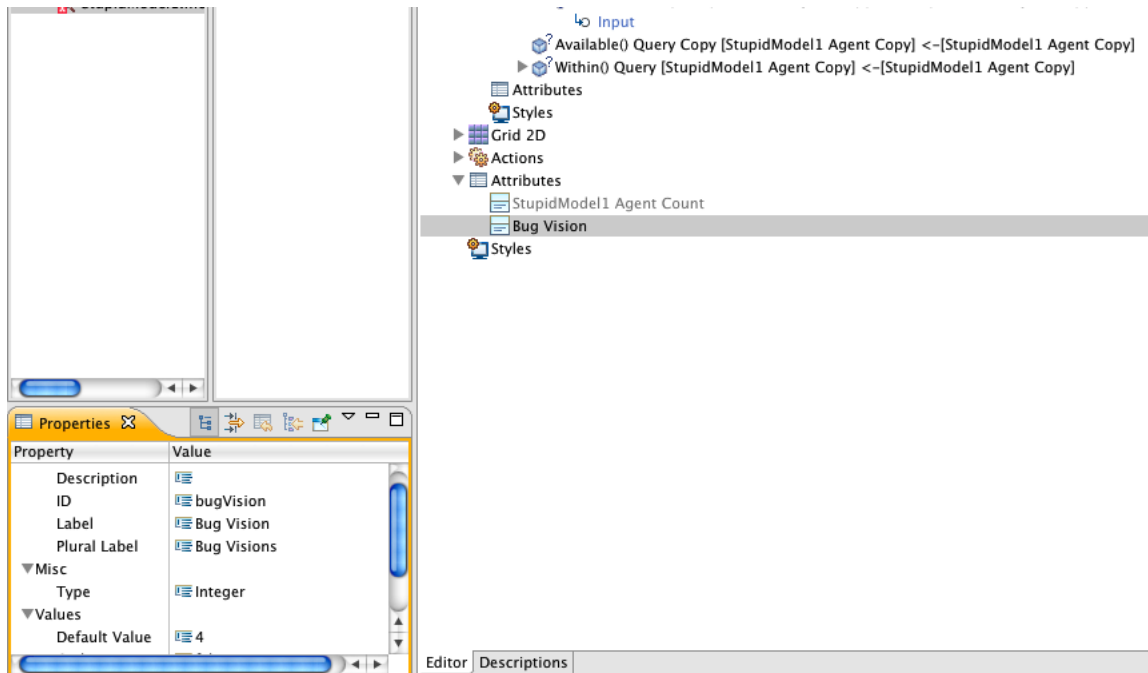


In addition to error markers, metaABM provides warnings designed to help you avoid common design mistakes. For example, a warning will be issued if you attempt to move an agent from its selection to the same selection; this is not strictly speaking an error, but it doesn't make much sense. You'll also receive warning for default values that have not been specified. You might notice that the Demographic Prisoner's Dilemma model has warning markers, this is because we were happy with the '0' default values for the attributes and didn't provide any. (By the way, you might also notice a number of Java warnings. By default the Eclipse IDE is very conservative when it comes to warnings, and expects the usage of language constructs that we choose not to use. A full discussion of all of this is beyond the scope of the tutorial, but see the Eclipse documentation on problem views for more ideas about how to customize these views.)

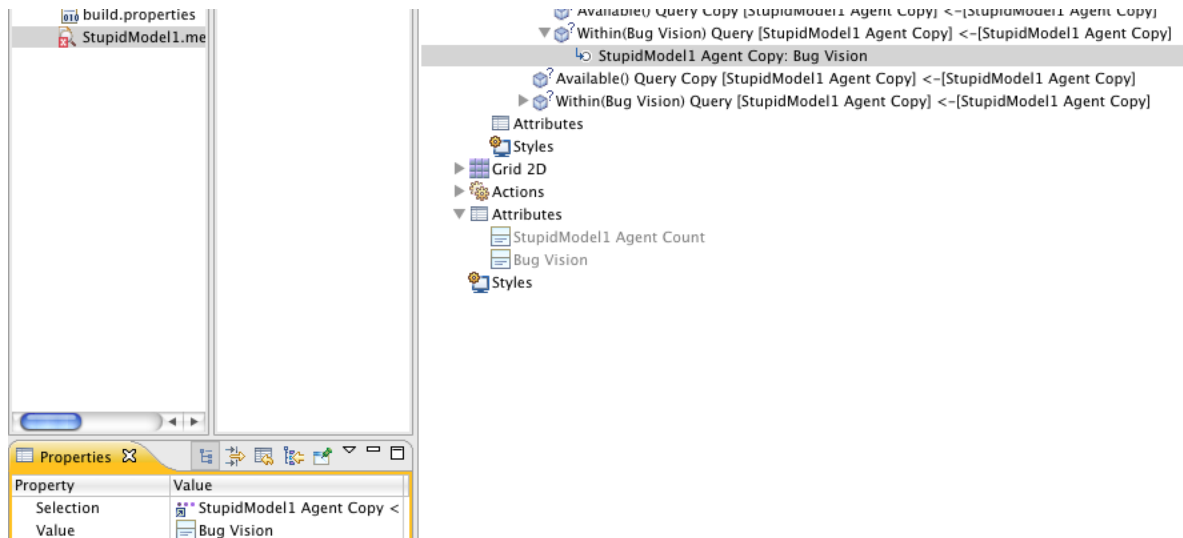
To fix this error we just need to assign a value for search distance. We **could** simply create a literal for the Spatial: Within Query but that isn't really good practice. (Originally the Agent Modeling Framework didn't even support Literal values, but we added them for convenience.) Now, we will create our first Attribute. In this case, we want the Attribute to belong to the Stupid Model as it will be the same value for all Agents for now. So right-click on the Stupid Model context Attributes node and create a new one.



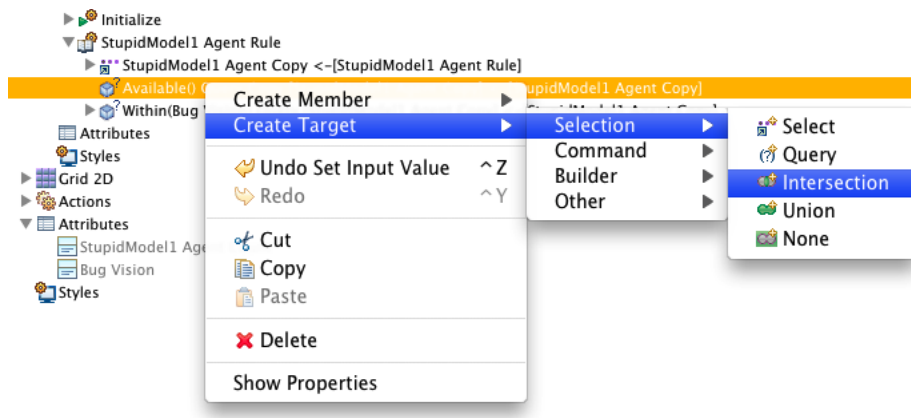
Name the Attribute -- "Bug Vision" seems descriptive -- give it a type of Integer, and assign it a default value of 4. This will allow most ABM platforms to provide the ability to change vision at runtime or through batch definitions, something we couldn't do if we had used a literal value.



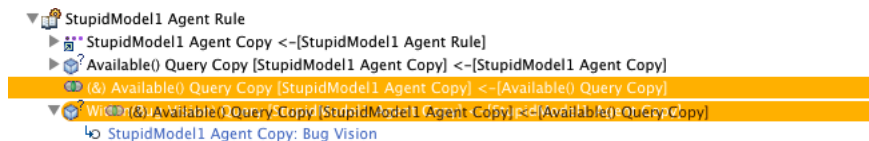
Finally, we assign the Vision attribute to the "Input" node in our Spatial: Within Query.



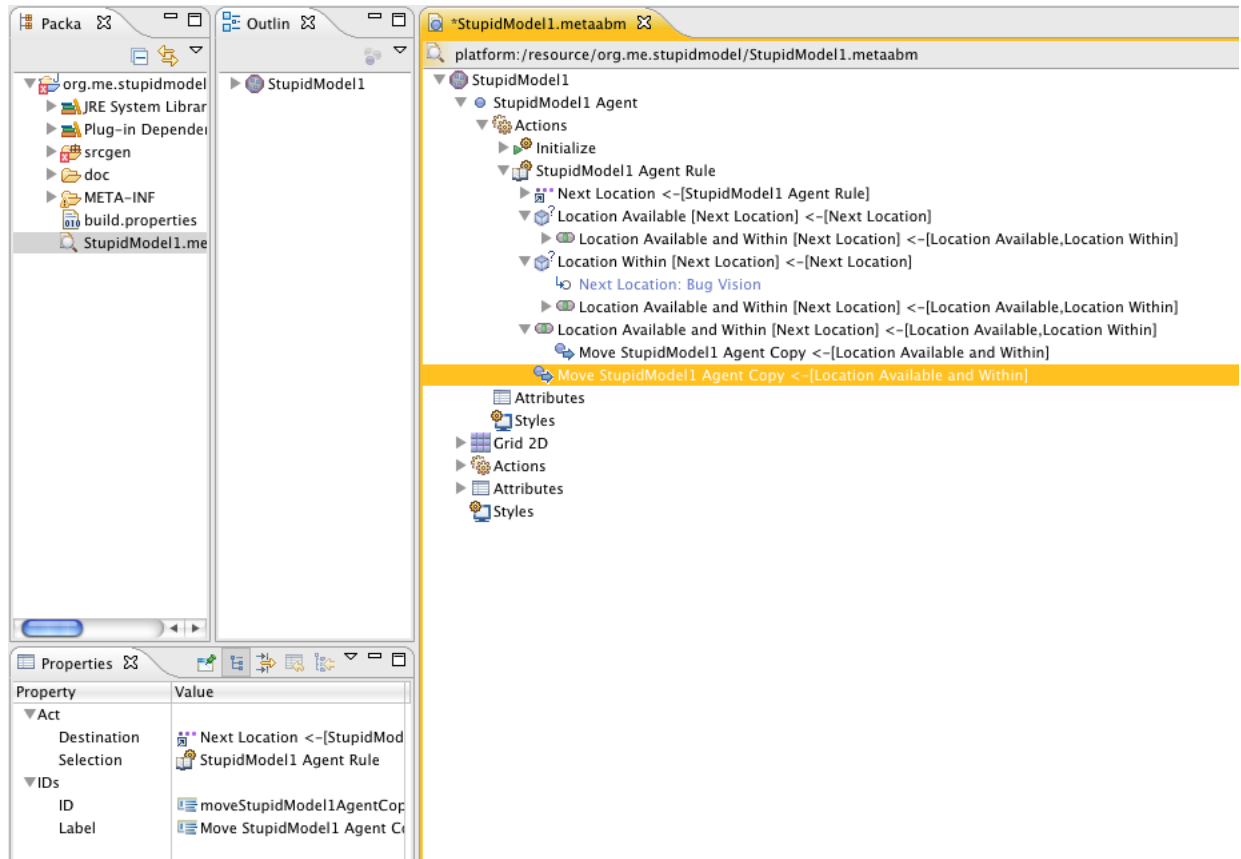
Now, we need to combine the requirement that the cell be available with the requirement that it be within 4 cell's distance. To accomplish this, we'll add an Intersection Action. The Logical Actions, including "Intersection", "Union" and "None" define how Query Actions work together. So create an Intersection Actions as a target of the Spatial Available Query. (Not the Within Query).



The Intersection Action needs to be a target of *both* the Available and Within targets. To accomplish this, drag the Intersection Action over the Within Query. It's a bit hard to see this in a static image, but you simply need to click on the Intersection Action, move it so that it is over the Within target, and release the mouse.



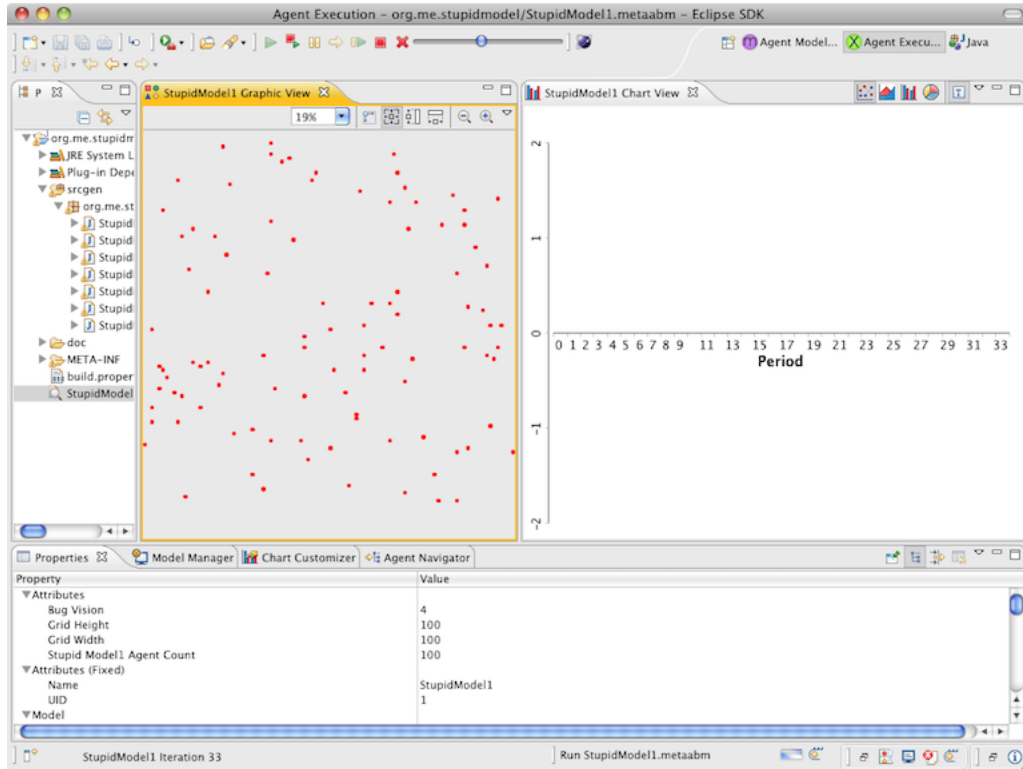
Finally, add a Move target to the Intersection.



A few notes on the screenshot above. To make things a bit clearer, we've edited the names for the final panel. Also, the Intersection node might not appear in the same place. We've expanded the nodes so that you can see that while the actions are all listed together, they are actually defined as a tree structure internally. You can follow that tree to see all of the actions that might be the result of any of the actions in the list. To help understand the structure at a glance, the labels include an <- indicator showing the immediate sources of each of the nodes. *Note especially that while the targets for actions often appear immediately following their source actions, this is not always the case.*

4.2.5.2. Run Final Model

Now, we can select the model in the **Package Navigator** again, and run the model. It doesn't look much different in a snapshot, but those agents should now be moving around. We have built a complete Ascape model from our model.

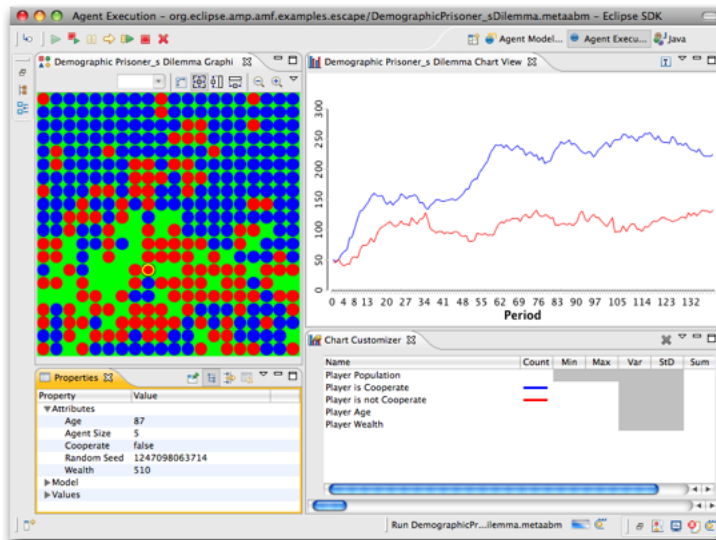


We hope this tutorial has enabled you to get up and running and given you some familiarity with the basic Agent Modeling Framework approach. The example models include the other "stupid models" from the paper, as well as a number of other interesting models.

Chapter 5. Programmer Guide

5.1. Overview

Escape is a full-featured Agent-Based Modeling (ABM) integrated development environment (IDE) with a Java based API and end user modeling tools. It's based on Ascape, which has been in use for more than 10 years. The core API is very stable, and that should give users a way to explore the features of AMP without concerns about keeping in synch with the rapidly evolving AXF /AGF API. It allows modelers to code in Java and/or generate models with AMF and then execute those models within the same development environment.



A primary purpose of the Escape project -- apart from the obvious goal of providing a nice Eclipse hosted ABM toolkit -- is to provide an exemplar to demonstrate how any arbitrary agent modeling framework can be integrated within the AMP environment. A cool aspect of this is that AMP has no dependencies on Escape, but also that the underlying ABM modeling framework for Escape has no dependencies on AMP -- instead Escape simply provides the runtime glue between AMP and the ABM framework. The underlying ABM framework uses the Ascape API, an API that first began development more than ten years ago and that has not been modified at all in order for it to work within the AMP environment.

5.2. Documentation

In addition to this documentation, there are a number of other resources available related to Escape development. Because its API is based on Ascape, a lot of resources developed for Ascape are equally applicable to Escape. The Ascape documentation is bundled with some distributions of the AMP tools, but if you are using the Eclipse AMP download, you can access it online or install it from the Ascape update site.

5.2.1. Online

There is a complete [Javadoc](#) as well as an [excellent manual](#) and other web resources. See the [Ascape Website](#) for general information.

5.2.2. Update Site

Or, you can install the Ascape Manual into the eclipse help system by adding the following update site and selecting the "Documentations and Source Code" feature. See the Installation Guide for more information.

<http://ascape.sourceforge.net/eclipse>

5.3. Installation

See the Installation Guide for Detailed Information -- that all applies to general Escape usage as well.

Note that there are no dependencies on AMF, and therefore EMF or any other modeling technologies to do Java development for Escape or for running generated models within Escape. The deployment environment can be quite light-weight. Simply install the AXF, AGF, and AMP Escape features. You can build from source but in that case you'll need to launch a self-hosted runtime, so it's best to simply install from the update site if you're not actually doing AMP development.

Escape is [installed](#) by default with the other model components. If you want to do 3D (really 2 1/2) visualizations, you will want AGF3D and the LWJGL dependency.

5.4. Example Java Models

5.4.1. Exploring Example ABM Models

The first thing most people will want to do in AMP is to play with the example models. You can explore AMF models as well as some really interesting classic ABM models.

5.4.1.1. Installing Models

See the Installation Guide to find out how to get the example models.

AMF Models

All AMF models produce pure Java code so they can be used like any other Escape Java model.

`org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.escape`

To understand the capabilities of AMF in producing other kinds of Java models, you might also want to try out running the same models automatically generated for Ascape or Repast. These are located at `dev.eclipse.org` in `cvsroot/modeling`:

`org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.ascap`
`org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.repast`

Classic Escape / Ascape Java Models

Many models have been created using Ascape over the years, including all of the classic models created at Brookings and some other cool models such as Craig Reynold's Boids and a pretty cool little traffic model, and they've all been converted to run in Escape. They're in Java but you can execute and explore them in exactly the same way as the Epidemic model above. All of the Ascape example models have been converted (a straightforward process) from their initial Ascape incarnations. For licensing reasons (they're BSD, not EPL) we can't host them directly on the Eclipse site. You can get the projects in two ways:

Once you've downloaded the projects, open up the `src` folder and navigate to the Java files for the actual models. For example, to run Conway's Life, you'll want `src/edu.brook.life.ConwayLife`. Right-click on the Java file and select "Execute". There are many models to explore! (It's not always obvious which Java files are for models, so you may have to poke around a bit. We need to put together a catalog here. Another nice opportunity for user contributions.)

These are SVN projects in the sourceforge SVN. For help, see http://www.eclipse.org/subversive/documentation/teamSupport/find_check_wiz.php this page.

`http://ascap.svn.sourceforge.net/svnroot/ascap/org.ascap.escape.models.brook`
`http://ascap.svn.sourceforge.net/svnroot/ascap/org.ascap.escape.models.examples`

5.4.1.2. Generating Models

And of course, you can always use the AMF tools to generate models and then customize them from there. See the following section on integrating AMF with Java for more on that. There are a number of models

developed with AMF that can automatically generate example models to run in Escape. There are two ways to run the example models:

Creating a New Escape Project

The best option is to simply create a new Escape project and drag the models into it!

1. If you're not already there, open the Agent Modeling Perspective. Select Window > Open Perspective > Other.. and then Agent Modeling. (Or you can choose Window > Reset Perspective and the Agent Modeling perspective should appear in the Open Perspective list itself.)
2. Create the project where we'll place our model. Select File > New > Escape AMF Project. Give it a name like "escape.tutorial", then click Finish.
3. Finally grab the Epidemic.metaabm model at <http://download.eclipse.org/amp/models/Epidemic.metaabm> and drag it into the project. (Depending on your browser the file may load as text directly into your browser, in which case just right-click on the link and save it to a file.)

5.5. Developing Models

The easiest way for Java developers to get started doing agent-based modeling in Eclipse is to begin to write programs using it. To develop new models, you can:

5.5.1. Cheatsheet

By far the simplest way to get started is with the cheatsheet. This walks you through the complete process of creating a Java based model. **Help > Cheat Sheets...** and then select "Create an Escape Java Model" from within the "Agent Modeling" category.

5.5.2. Steps

1. Create a new Escape project. The Escape projects are actually configured for AMF code generation so there are dependencies and builders in there that you don't need; you can remove all of the escape builders and any of the kitchen sink items. The "Create an Escape Java Model" cheat sheet will walk you through the process of creating a new Java based project and model.
2. Or for more experienced Eclipse users, you can simply to just create a new Plug-in Project and add the necessary dependencies there. Have a look at the example escape project to see what you need. Note that you will likely need more dependencies than you need to simply build -- this is because the class loader uses the classes from the project path and so runtime classes have to be there as well. We may simply package an Eclipse runtime convenience plug-in to gather these dependencies up a bit. Then just create new Java classes for your root model scape and agents just as you would for an Ascape project.

5.6. Executing Models

See the User Guide Execution Section for information on launching and controlling models.

We have a complete cheat sheet supporting this tutorial. Simply go to **Help > Cheat Sheets...** and then select "Run an Example Model" from within the "Agent Modeling" category.

5.6.1. Tutorial

We've developer have a complete cheat sheet supporting this tutorial. Simply go to **Help > Cheat Sheets...** and then select "Run an Example Model" from within the "Agent Modeling" category.

5.7. Extending and Customizing AMP

This subject is beyond the scope of this Agent Modeling manual. A future Agent Modeling Platform Guide will discuss these issues in detail. For now, please keep an eye on the Wiki and committer blogs. And of

course please ask questions on the AMP forum and developer mailing list. Letting us know you're interested in doing something is the best way to get it documented!

5.8. Integrating Java and AMF Models

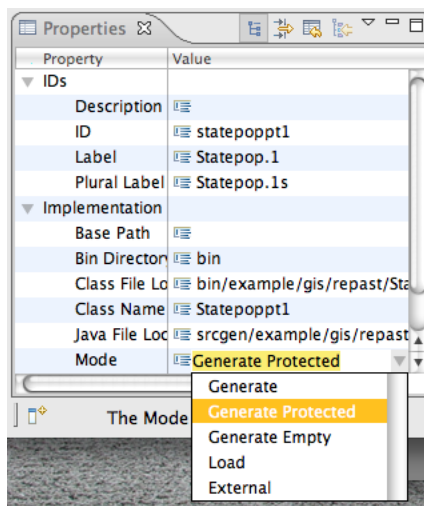
If you're like many Java developers, you might find point-and-click interfaces a bit lame. Personally, I've changed my tune in this, and I now define all of my ABM models from the editor, saving Java for truly specialized tasks. But even without generation of agent behavior, Acore can be a really valuable tool, as the GIS example shows. The way to look at metaABM is as a way to compose your overall model and automate the tedious parts. Apart from Java generated code, the AMF meta-model maintains a number of very useful artifacts. For example, the Repast Symphony target maintains `model.score` and all of the `model.rs` component. Generally, AMF should save time and hassle while making your models far more transparent even if you never use the Actions component to define agent behavior.

5.8.1. Method Action

As explained in the action section, you can simply create a "Method" act with hand-written Java code. This option is nice because all code is contained within the AMF file. But it can be difficult to maintain large blocks of Java code as you aren't using a Java editor to edit the Java code itself. One way to get around this is to create your code in the generated Java method and then copy it into the Method action. Note one important issue here -- you'll generally have to fully qualify your Java references as you won't be able to change the imports statements directly.

5.8.2. Protected Code Regions

You can mix and match Action behavior with Java and generated code with POJOs. One way to do this is through using protected regions. Select the agent you want to create protected methods for and then select "Generate Protected" from the "Mode" property. Now, create actions just as you have before, or use your existing ones. On code generation, open up the relevant java file and examine the methods that have been created.

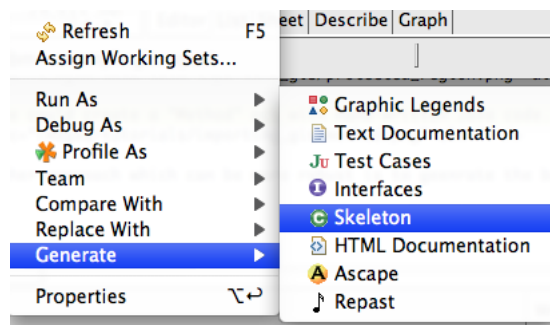


You can put whatever you want within the PROTECTED REGION comments and those changes will be preserved when the model is regenerated. You can create a schedule, rule or watcher, maintain custom code for the actual implementations, and still have the model adapt to changes in the underlying data structure -- if for example you want to import a modified shape file.

```
@repast.simphony.engine.schedule.ScheduledMethod(start = 0, interval = 1, priority = 0)
public void doSomething() {
    /*
     * PROTECTED REGION ID(example.gis.repast.Statepoppt1_doSomething)
     * ENABLED START
     */
    if (getINPOVRTY() < 0.2) {
        //do something else..
    }
    /* PROTECTED REGION END */
}
```

5.8.3. Interface and Base Class Generation

Another approach which can be more robust is to generate the basic model stubs (like an abstract base class except that it isn't abstract) and then override your model with implementations. AMF provides support for generic skeletons and interfaces.



5.9. Converting Existing Ascape Models

There are only a few changes should have to make to existing Ascape models or to use existing Ascape documentation to develop Escape models.

5.9.1. Model

The core model is completely API compatible. No changes!

5.9.2. View

Because Escape uses SWT and Ascape uses Swing, there are a few unavoidable incompatibilities. Most of these we avoid by using higher level APIs but here are the key changes that you're likely to have to make:

5.9.2.1. Convert the low-level imports from AWT and Swing to SWT

The simplest way to accomplish this is to remove all of the imports and then organize imports. For example:

```
java.awt.Color => org.eclipse.swt.graphics.Color
java.awt.Graphics => org.eclipse.draw2d.Graphics
```

Then just do a global find for all of the imports and replace them with nothing. This is a great place for a regexp. Try:

```
find: import java\.awt\(.*)
replace: [nothing]
```

You don't have to replace these with the SWT equivalents, just click on the project, right-click and choose "Source:Organize Imports.." By the way, a great way to avoid having to select the right entries in optimize imports and to alert you when you have missed anything is to prevent the awt and swing classes from being used at all. Right-click on project, choose "Build Path:Configure Build Path", go to Libraries tab, open JRE System Library, choose "Access Rules", edit, and then add entries for java/awt/** and javax/swing/**. The code will regenerate and you'll have error markers for all of the stuff that won't work with Escape and Eclipse.

5.9.2.2. Convert color features

You can't use AWT colors either so you'll need to replace any colors. AMP provides a convenience classes for Colors called ColorFeature and ColorFeatureConcrete. You can use these or any of the other ways to define SWT colors. For example:

```
Color.lightGray => ColorFeature.LIGHT_GRAY
new Color(Math.min(1.0f, (float) (redEnergy + orangeEnergy)), (float) orangeEnergy * .8f, (float) blueEnergy) =>
    ColorFeatureConcrete.create(Math.min(1.0f, (float) (redEnergy + orangeEnergy)), (float) orangeEnergy * .8f, (float) blueEnergy)
```

5.9.2.3. Change agent color getters

If you've defined colors through overriding Agents as in most models, you'll need to change the method signature. You could just do a global replace for this one.

```
public Color getColor( => public Object getPlatformColor(
```

5.9.2.4. Get rid of image features

Escape doesn't support them. In practice they haven't been used a lot. At some point perhaps we'll have nice sprite support instead. :)

```
public Image getImage() {**} => ""
```

5.9.2.5. Modify usages of DrawFeatures

If you've created any of your own draw features, you'll need to change them slightly to accommodate the differences between the AWT and SWT / Draw2D APIs. This should be pretty straightforward. For example:

```
g.setColor(...)
g.fillOval(..)
becomes:
g.setBackgroundColor(
g.fillOval(..
```

That's about it, though it's probable that we've missed something. Please post a message on the amp newsgroup if you run into any other conversion issues.

5.9.2.6. Configuration

Instead of using Ant, Escape has really nice support for parameterizing and testing models. See [Experimenting and Testing](#) (todo). If you do want to work with ant you will need to install the plugins from the Sourceforge download site. (These have dependencies that I didn't want to bother with going through the Eclipse IP process to get in. If for some reason you think these should be part of core Escape let Miles know and I'll take care of it.) Ant has not been tested for Escape so it's not even clear it will work.

Chapter 6. Installation Guide

6.1. Tools

6.1.1. Complete IDE

You may already have obtained a complete IDE bundled with AMP and other modeling tools. The project contributors offer open source and commercial versions of the AMP tools, which you can find at their sites. If so, you can mostly ignore this chapter unless you want to install an optional component like Repast.

6.1.2. Eclipse and the Agent Modeling Tools

You can find more detailed instructions [here](#), but here are the basic steps.

1. Install the latest version of Eclipse from [here](#). AMP requires version 3.5.x (Gallieo) or higher. Version 3.6 (Hellos) which is currently a development (beta) release works well too. We recommend the "Classic" release at the bottom of the page as the other releases have a bunch of stuff you won't need.
2. Add update sites for AMP and AMP dependencies. See the screenshot below and the [Eclipse Documentation](#). There are different versions of the update sites, depending on whether you want to use the most stable version of the toolset or the "latest and greatest". The update sites are listed below.
3. (Optional) If you install and AMP extensions, add Update Sites for them -- they're also listed below.
4. Select the features that you want to install.
5. Click the **Next** button, review the licenses, and then click **Finish**.

Agent Modeling Platform update sites:

Milestones (recommended): <http://download.eclipse.org/amp/updates/milestones>

Interim (newer): <http://download.eclipse.org/amp/updates/interim>

Nightly (bleeding edge): <http://download.eclipse.org/amp/updates/nightly>

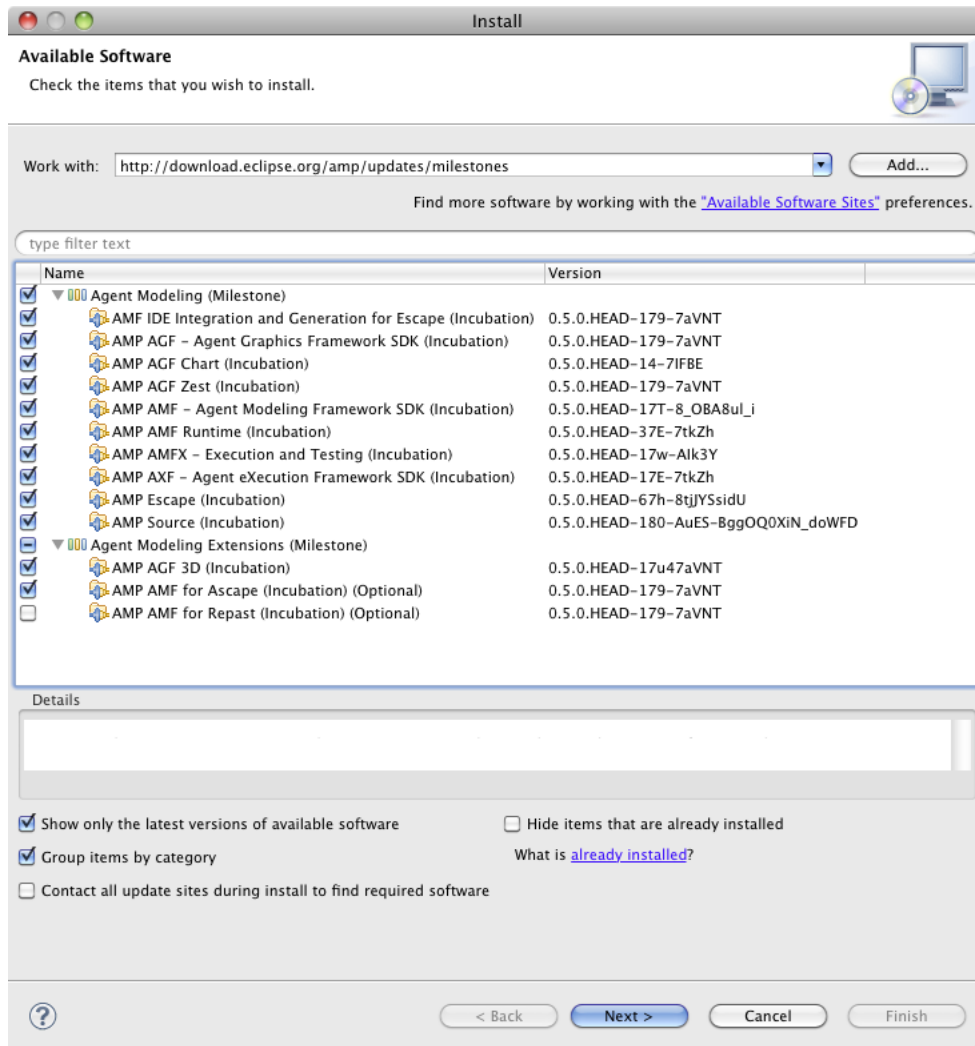
Update sites for AMP Extensions

AGF 3D (recommended): <http://lwjgl.org/update>

AMF for Ascape (recommended for Java developers): <http://ascape.sourceforge.net/eclipse>

AMF for Repast (for people using Repast): <http://mirror.anl.gov/pub/repastimphony/site.xml> (non P2)

In the following screenshot, we've added the LWJGL site (not shown) as well as the Ascape site (not shown), but not the Repast site, and we're about to install the respective features.



Once you've done that, select the **Help > Help Contents** menu, find the Agent Modeling Guide and open this overview page within that guide.

6.1.3. Extensions

6.1.3.1. Ascape

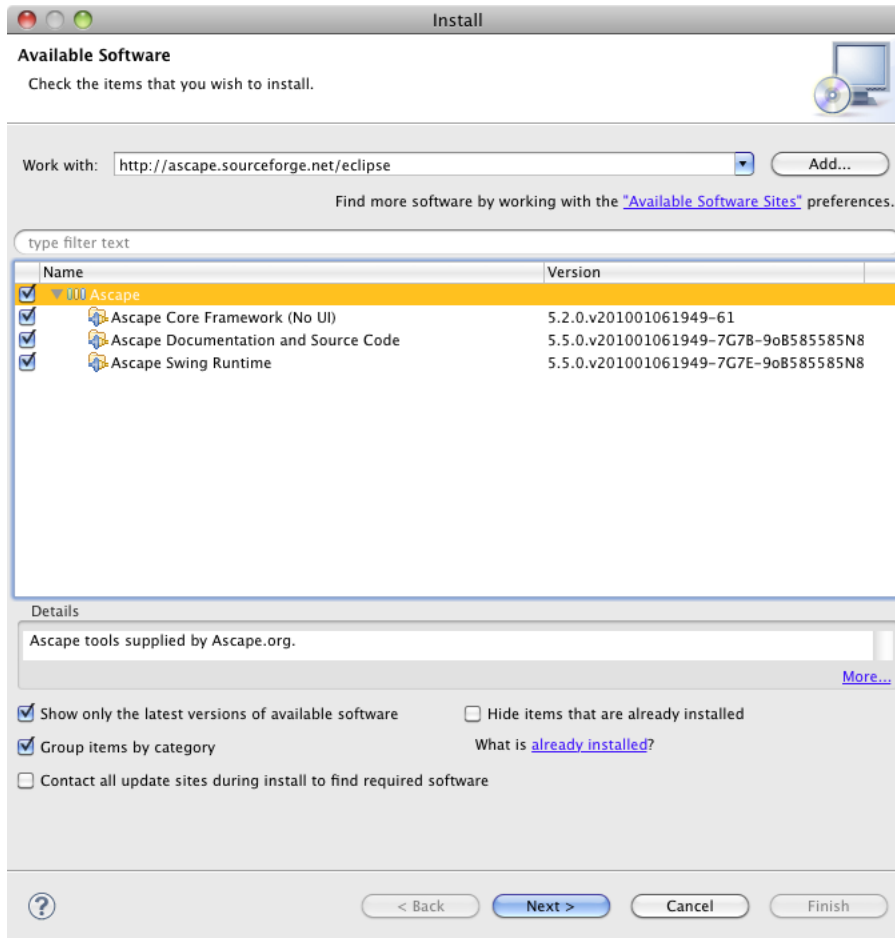
For licensing and copyright reasons we can't provide all of the Ascape tools and manual on the Eclipse hosted Agent Modeling Project site. None of these tools are necessary to use AMP (the core Ascape support is an IP approved part of the AMP distribution) but they're highly recommended. The Ascape tools include:

1. An excellent Manual covering programming Java models in Ascape and Escape. (The API is the same.)
2. Support for creating Ascape Java projects and executing the Swing-based Java Applications. Swing deployed Ascape Java Applications can be as small as 10MB and can be deployed to the web.
3. Eclipse based IDE support for plain Java Ascape projects.

The Ascape update site is at:

<http://ascape.sourceforge.net/eclipse>

To install Ascape, use the update manager. Select **Help > Install New Software...**, add the Ascape update site, select the Ascape category and click the **Finish** button.



6.1.3.2. Repast

The Repast update site is at:

<http://mirror.anl.gov/pub/repastsimphony/site.xml>

More information on downloading and installing repast can be found [here](#). Follow the instructions for "Configuration B".

6.2. Models

6.2.1. Obtaining Example Models

There is a cheat-sheet available for obtaining the example models. If you would rather obtain them yourself, here is the CVS information:

6.2.1.1. Team Project Sets

You can use the following team project sets. See [this document](#) for more information about using project sets.

<http://eclipse.org/amp/project-sets/ascape.psf>

<http://eclipse.org/amp/project-sets/escapeAMFExamples.psf>

6.2.1.2. From Version Control Systems

Import the CVS projects from the AMP project repository. For help, see [this page](#). Here are the details for CVS Access:

CVS Repository:

Connection type: pserver
User: anonymous
Password: [none]
Host: dev.eclipse.org
Path: /cvsroot/modeling
Module: /org.eclipse.amp

AMF Escape Models:

org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.escape

AMF Ascape and Repast Models:

org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.ascap
org.eclipse.amp/org.eclipse.amp.amf/examples/org.eclipse.amp.amf.examples.repast

The Ascape example projects are available at the Sourceforge SVN server. You can get them at the following locations. See the http://www.eclipse.org/subversive/documentation/teamSupport/find_check_wiz.php Subversive User Guide for more information about using SVN. (You'll need to install Subversion if you don't already have it.)

Escape Java Models:

<https://ascape.svn.sourceforge.net/svnroot/ascape/org.ascape.models.brook>
<https://ascape.svn.sourceforge.net/svnroot/ascape/org.ascape.models.examples>

Ascape (Swing-based) Java Models:

<https://ascape.svn.sourceforge.net/svnroot/ascape/org.ascape.escape.models.examples>
<https://ascape.svn.sourceforge.net/svnroot/ascape/org.ascape.escape.models.brook>

Chapter 7. New and Noteworthy

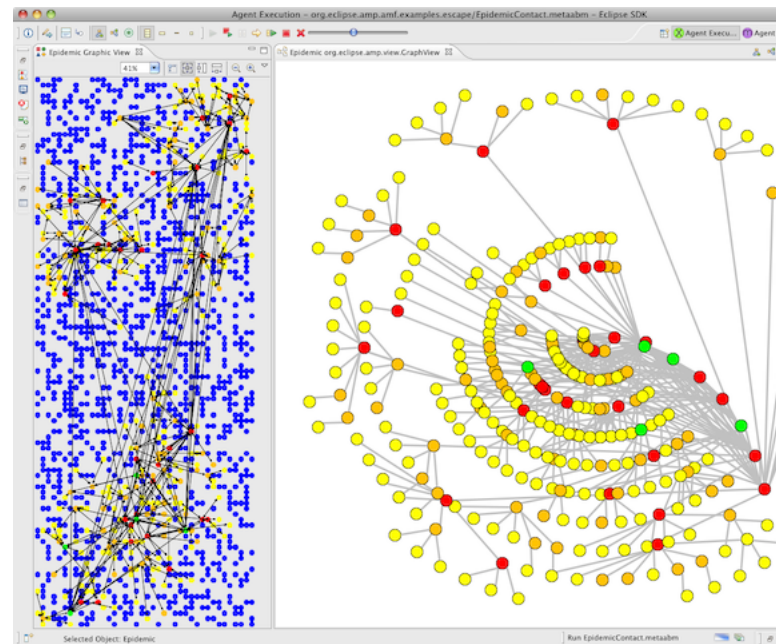
7.1. Release 0.8.0

The first official release of AMP includes many new or expanded features in addition to overall quality and usability improvements. The highlights are listed below.

7.1.1. Model Visualization

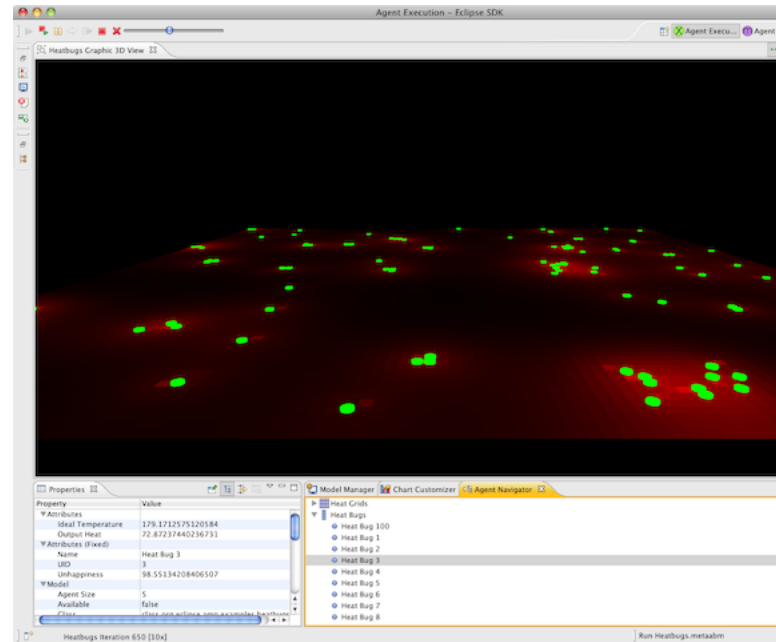
Graph Visualization

Built in support for visualization and generation (as always with AMF, no code to write!) of graph models.



3D View Improvements

2 1/2 visualizations now support interpolated positions for agents. You can't see that here, but it means that the agents move smoothly through space, even if they are on a fixed grid. You can turn that feature on and off and you can change perspective easily.



API Improvements

There have been significant changes to the APIs. As always, we've followed common Eclipse and EMF idioms, which means that to customize model visualization you typically only need to implement figure, shape or color providers and then adapt them to your model. And of course all of your existing models will generate code for the new APIs.

More Legacy Views

In addition, we've supported all existing Ascape views (not all of these are supported through AMF generated code..yet!).

7.1.2. Modeling Framework

New Actions

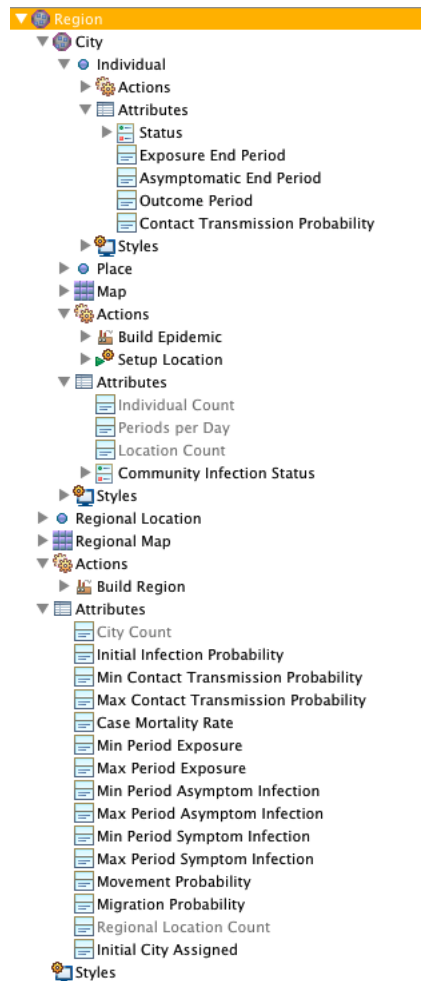
Some key new Actions have just been introduced. These provide support for recursion (Cause and Perform), minimization and maximization queries, diffusion (Diffuse) and derived (Derive) values and give the framework a much more general purpose flavor while preserving model generality. You should be able to build many more kinds of models without resorting to custom coding. See the Modeler's guide Actions section for the details.

Improved Model Representation and Generation

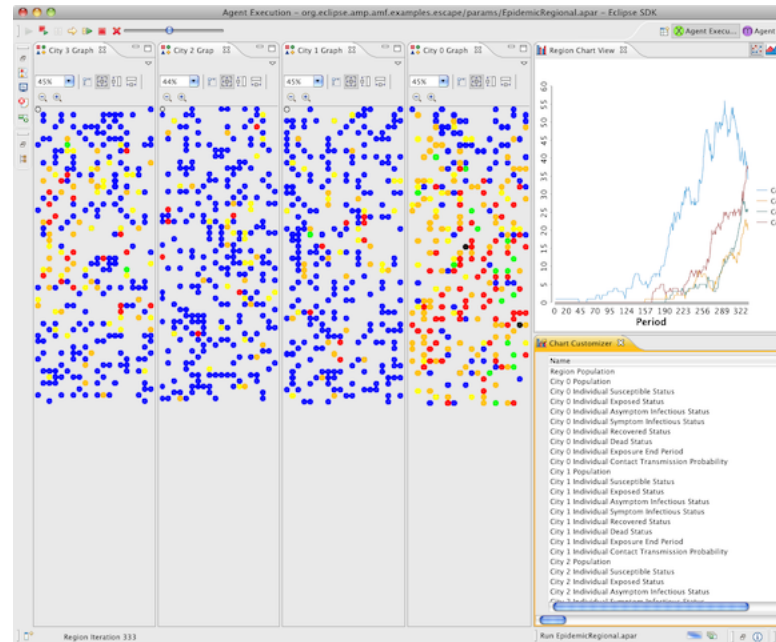
There have been many improvements to how models can be designed and the code that is generated from those designs. As just one example, you can now search for agents within a model without having to specify an explicit space. As with model editing improvements, many of these changes have been driven by user input. If there is something you'd like to be able to do with AMF but can't, let us know by filing a feature request or posting to the newsgroup.

Multiscale Support

Support for modeling across scales has been greatly improved. For the Escape target it is very easy to define models with enclosing hierarchies. For example in the EpidemicRegion model, we've defined Regions which contain Cities which contain Individuals.



When the model is executed, views are automatically created for each member Scope (context) and data is collected for each level of scale.



7.1.3. Model Editing

Overall Usability

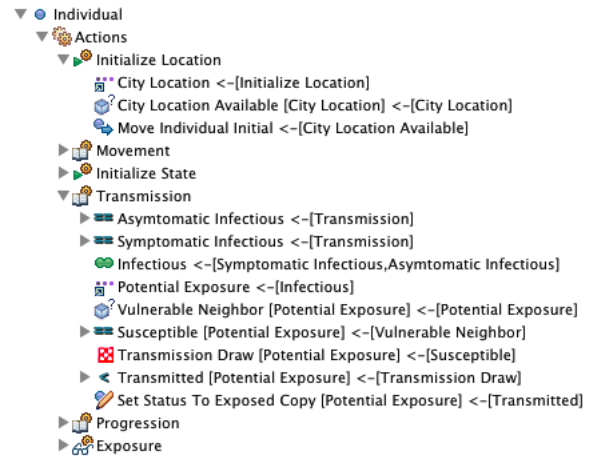
Significant effort has gone into improving the overall user interface for editing models. User feedback has been enormously helpful here -- please keep letting us know what works and what doesn't! In addition to the obvious changes, a lot of work has been done under the covers to provide a more seamless model building experience. We've fixed some glitches, but we've also added a number of things to help simplify and automate the model building process. As an example, when you create an agent, a style is automatically created along with default color, shape and paint actions.

New Icons

Most of the existing icons have been replaced or custom designed (which has incidentally required a lot of documentation to be recreated) to provide a better and more consistent user interface.

Action Lists

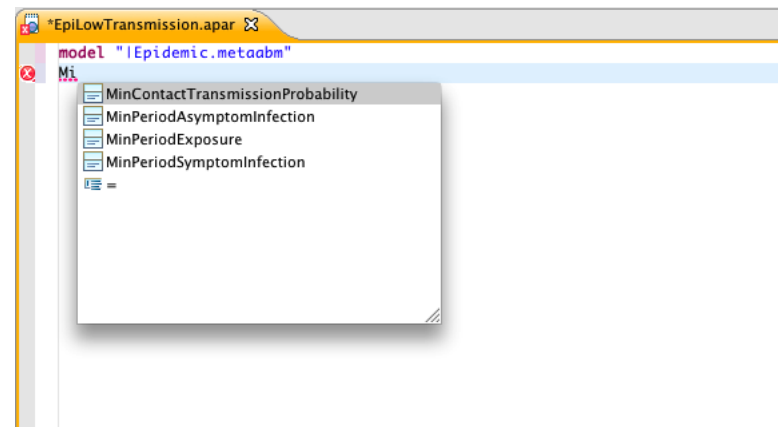
Actions are now displayed in lists, not in a tree structure. As the actual Actions are graph structures, the tree structure never matched well with this representation. Taking a simpler approach has made the action definition process much more transparent.



7.1.4. Modeling Tools

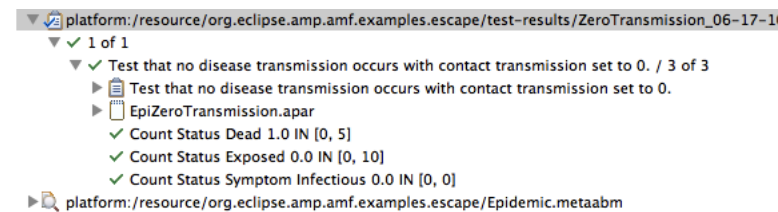
Parameter Management Support

AMP now has full support for editing and launching of parameter files, including code completion. There is also support for creation of tests and analysis of test results. Finally, data can now be generated to a common EMD based (adata) data format.



Test-Driven Modeling and Simulation

Tests can now easily be defined and analyzed.



Automated Data Export

Data can now be easily generated to a common EMF based (adata) data format.

7.1.5. User Experience

Overall

Last but certainly not least, a lot of effort has gone into making it easier to get into AMP as well as to understand how to work with its more powerful

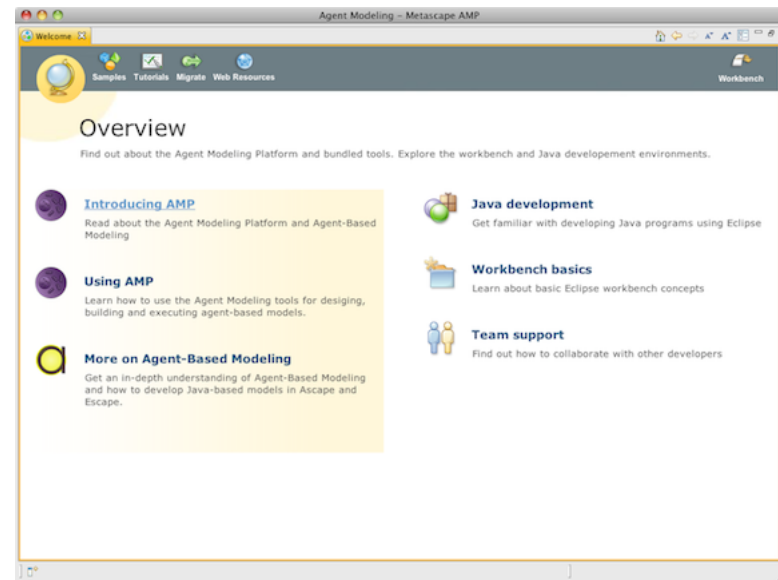
Documentation

features. Eclipse supports a number of sophisticated User Assistance features, and we've taken advantage of most of them.

Over a 100 pages of documentation, plus another 100 pages of Ascape documentation converted to Eclipse Help and PDF.

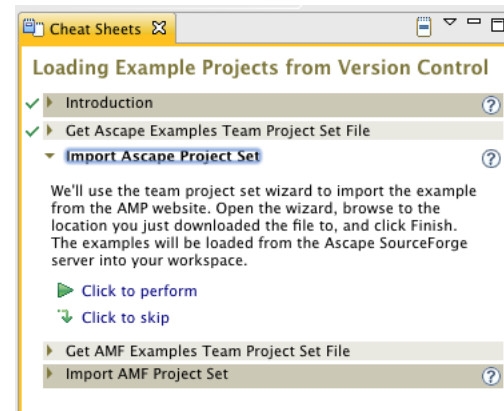
Welcome and Intro

When starting AMP you'll be met by a custom welcome screen providing an easy path into Agent Modeling on Eclipse.



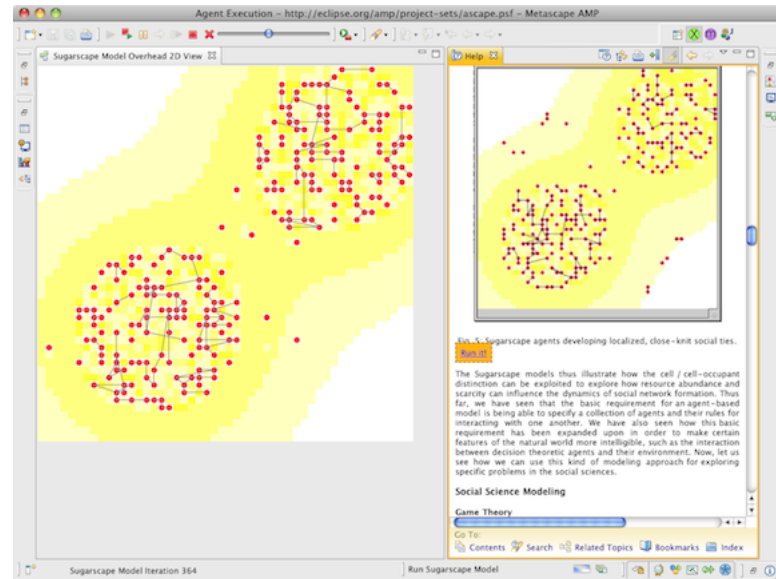
Cheat Sheets

Helpful guides walk you through initial steps.



Integrated Models

You can run example Ascape models directly from the documentation!



Chapter 8. Resources

8.1. Websites

[Eclipse Agent Modeling Project \(Incubation\)](#)

[Ascape](#)

[Metascape, LLC](#)

8.2. Papers

[Ascape Agent-Based Modeling Why Model?](#)

Chapter 9. Support

9.1. Issues

Eclipse and the AMP project have a mandate for open communication. Please note that unless you are a customer of an AMP project contributor we will not respond to direct requests for support. We *will* make every effort to support your use of AMP in the following ways:

9.1.1. General Support, Questions and Discussion

The [Agent Modeling Platform Newsgroup](#) is the place to discuss or get support for any of the Eclipse AMP tools. Don't hesitate to ask questions about model implementation or bring up more general topics for discussion.

9.1.2. View Existing Bugs and Feature Requests

If you have a problem or idea for improvement, there's a good chance that its already been discovered. To avoid duplication, please check the current bugs before filing a new bug report or feature request.

[Search Existing Bugs](#)

9.1.3. File a Bug

This may sound obvious, but if something doesn't work as it should, the only way that we're going to find out is if someone reports it. All AMP related bugs or feature requests should use the Eclipse Bugzilla.

[File a Bug or Feature Request](#)

9.1.4. Discuss AMP Development and Project

All discussion about software development, design and project management issues take place on the AMP dev-list. If you're interested in contributing to the AMP project, this is a good place to contact us. <https://dev.eclipse.org/mailman/listinfo/amp-dev>.

9.2. Other Contacts

9.2.1. Professional Support

The AMP project developers -- among the most experienced developers of agent models and tools anywhere -- are naturally the best source for Agent Modeling development, support and training. Please feel free to contact the project lead or other project contributors directly to discuss professional support options.

9.2.2. Media and Private Communications

For project related but non-public communications such as media enquires or potential project sponsorship, please contact Miles Parker, the Eclipse AMP project lead, at amp_enquiry@metascapeabm.com. (Requests for support and other communications to this address will be ignored.)

9.3. Get Involved

If you find this tool valuable, please be aware that most of its development occurred through volunteer contributions from individuals and companies that have received no direct funding. You can support continual development efforts financially by purchasing related software tools from companies and organizations that help develop it, including funding for developer time in grant proposals. You can support the Eclipse platform in general by becoming a member organization of Eclipse. But the most important thing that you can contribute is your own time and enthusiasm, either through directly contributing to AMP or by promoting

its use in blogs and other social media. For more detailed information about contributing to AMP, please see [our wiki](#).