

A Guide to Application Development

**ENVISION – Software for Alternative Futuring Applications**

**A Guide to Application Development**

**Copyright © 2014**

**Oregon State University**

DISCLAIMER

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

For more information about ENVISION, contact:

John Bolte, Professor  
Biological and Ecological Engineering Department  
Oregon State University  
Corvallis, OR 97331  
Email: boltej@engr.orst.edu

More information on ENVISION is available on the web:

**http://envision.bioe.orst.edu**

Last Revised: December 18, 2014

**Table of Contents**

[Chapter 1. Developers Guide Overview 6](#_Toc391022830)

[ENVISION’s General Structure and Conceptualization 6](#_Toc391022831)

[An ENVISION Flow Chart 6](#_Toc391022832)

[ENVISION Application Development Workflow 8](#_Toc391022833)

[Chapter 2. Input Files and Structures 9](#_Toc391022834)

[Project (.envx) File 9](#_Toc391022835)

[The Integrated Decision Unit (IDU) Shape File 16](#_Toc391022836)

[Chapter 3. Output Files 20](#_Toc391022837)

[Chapter 4. Policies 21](#_Toc391022838)

[Chapter 5. Actors and Decision-making 28](#_Toc391022839)

[Basic Concepts 28](#_Toc391022840)

[Social Networks 30](#_Toc391022841)

[Chapter 6. Evaluative Models 31](#_Toc391022842)

[Chapter 7. Autonomous Processes 39](#_Toc391022843)

[Chapter 8. Visualizers 41](#_Toc391022844)

[Chapter 9. The Delta Array and Dynamic Maps 42](#_Toc391022845)

[Chapter 10. Standard ENVISION Plug-ins 43](#_Toc391022846)

[Simple State Transition Model (SSTM) (Autonomous Process) 43](#_Toc391022847)

[DynamicVeg (Autonomous Process) 44](#_Toc391022848)

[Flow (Autonomous Process) 59](#_Toc391022849)

[Modeler (Evaluative Model and/or Autonomous Process) 62](#_Toc391022850)

[Program Evaluator (Evaluative Model) 65](#_Toc391022851)

[Spatial Allocator (Autonomous Process) 67](#_Toc391022852)

[Sync (Autonomous Process) 72](#_Toc391022853)

[Target (Autonomous Process) 74](#_Toc391022854)

[Trigger (Autonomous Process) 78](#_Toc391022855)

[Chapter 11. Running ENVISION from the Command Line 81](#_Toc391022856)

[Appendix 1. Example LulcTree File 82](#_Toc391022857)

[Appendix 2. Field Info Descriptors 84](#_Toc391022858)

[Appendix 3. ENVISION’s Spatial Query Language 86](#_Toc391022859)

[Appendix 4. ENVISION’s Mathematical Expression Language 89](#_Toc391022860)

[Appendix 5. Frequently Asked Questions 91](#_Toc391022861)

# Chapter 1. Developers Guide Overview

## ENVISION’s General Structure and Conceptualization

This Guide is intended to provide support for developing an application using ENVISION, or for those seeking to understand more of the internal structure that ENVISION uses to represent information and generate future scenarios. Fundamental to this understanding is the general structure ENVISION uses to represent landscape change, shown in Figure 1 below.

**Landscape Evaluators**

Generate Landscape Metrics reflecting scarcity of ecological, economic or social indicators of landscape production

Landscape Feedback

**Autonomous Change Processes**

Models of non-anthropogenic landscape change

**Policies**

Fundamental Descriptors of constraints and outcomes defining land use management decision-making

**Actors**

Decision-makers effecting landscape change by selecting policies responsive to their objectives (self-interest) and scarcities (altruism)

Policy Selection

Actions

Figure 1. General Structure of ENVISION

Each of these ENVISION components are definable by application developers for specific applications. The following chapters provide documentation for how to define these components and create a specific application of ENVISION.

## An ENVISION Flow Chart

ENVISION flow control is given in the figure below.

Plug-ins

For each year

Run Each Pre-year AutoProcess, Visualizer

Apply any Scheduled Policies

Run Actor Loop

Run Each Post-year AutoProcess, Visualizer

Compute Landscape Scarcity Metrics Using the Evaluative Models

Collect Data

EMRun()

Computes Landscape Scarcity (-3 to +3) metrics

APRun()

VRun()

APRun()

VRun()

EMInitRun()

APInitRun()

VInitRun()

Actor Loop

For each Actor

For each Actor IDU

Find Relevant Policies:  
Identify all policies satisfying the Site Attribute constraint

Score Relevant Policies:  
Compute Altruistic, Self Interested vectors; score policy based on vectors

Select and Apply Policy:  
Probabilistically select Policy to apply based on scores

Next Actor IDU

Next Actor

Next Year

IDU ShapeFile

Project (ENVX) File  
(contains settings, actor, policy, LULC, model, process descriptions)

ENVISION – Flow Chart

EMInit()

APInit()

VInit()

Startup

* Load Project (.envx) File
* Load Coverages
* Load Policies
* Initialize Actor(s)
* Initialize EvalModels, AutoProcesses, and Visualizers
* Initialize Data Collection System

Start a Run

* Set Scenario Variables
* Initialize EvalModels, AutoProcesses, and Visualizers
* Collect Start of Run Data
* Start annual loop

Additional Coverages

## ENVISION Application Development Workflow

A number of specific tasks are required to develop an ENVISION application for a specific location and to meet specific alternative future needs. These are shown in the figure below.

Establish benchmarks for landscape performance. These are the evaluation metrics.

Define Goals

Define Policies

Run Scenarios

Define Scenarios

Generate and Populate IDU Coverage with Relevant Attributes

Create/Assemble Landscape Evaluation and Autonomous Process Models

Analyze Results

These will specify alternative management strategies considered in the analysis.

These define the alternative policy sets and management strategies considered in the analysis.

The IDU coverage is a polygon-based GIS database that stores the spatial representation of the landscape and contains attributes required for policies and models used in the analysis.

These models are “plug-ins” that conform to ENVISION’s interface requirements. Landscape Evaluation Models generate performance metrics for each goal. Autonomous Process Models simulate change not produced by actor decisionmaking.

Use ENVISION to run multiple scenarios, generate output for analysis.

Use ENVISION’s built-in tools or export data to other analysis tools to interpret scenario results

ENVISION – Typical Workflow

# Chapter 2. Input Files and Structures

ENVISION consists of a number of files that collectively provide input to drive a futuring analysis. These are shown in the figure below and described here.

**Project File (\*.envx)** – An XML file specifying various ENVISION settings, input coverages, policies, actor definitions, land use/land cover descriptors, and plug-in model and process descriptors This is the file specified in the startup dialog box when ENVISION is first started.

**IDU Shape File (\*.shp)** – Contains polygon coverage with attributes needed by the application.

**Evaluative Model/Autonomous Process/VisualizerPlug-ins (\*.dll) –** one or more plug-in modules running models, computing landscape metrics, visualizations

**Additional Input Coverages –** Typically shape files with polygon coverage with attributes needed by the application.

## Project (.envx) File

The Project File is an XML-based file that specifies a number of settings and inputs required for a given ENVISION application. It is organized as a series of sections, specified in a particular order. Each section is specified by a header of the form <*section name>*. The Project File is a standard ASCII (text) file that can be edited with Notepad or any other ASCII editor, or any XML-based editor. Generally, this file is autogenerated by Envision and does not need to be edited directly. Instead, Envision provide a Project File Editor that allow maintenance of this file.

When starting a new project, Envision will prompt for the name and path for this file, and automatically create a basic Project file.

The sections of the Project file include the following:

**<settings>**

This section includes basic settings for ENVISION. Settings are specified by *name=’value’* attributes.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Definition** | **Argument** |
| actorInitializationMethod | Method used to initialize Actor weights | 0 = no actors  1 = actors are created from weights specified in the IDU coverage.  2 = actors are created from Groups defined in the project file. The IDU cover contains the group ID in a field called ACTOR  3 = actors are created based on a query string specified for the actorGroup in the project file  4 = a single actor is created from a single ActorGroup specified in the project file  5 = random actors are generated |
| actorAssociations | Reserved for future use |  |
| loadSharedPolicies | Determines whether shared policies are loaded from the policy database | 0=don’t load shared policies  1=load shared policies |
| debug | Reserved |  |
| logMsgLevel | Level of logging messages | 0=log everything, 1=log critical events, 2= log nothing |
| noBuffering | Determines where buffering is suported in policy outcome. | 0=buffering supported  1=buffering not supported (recommended) |
| multiRunDecadalMapsModulus | Frequency at which decadal maps are generated during multiRuns | Frequency, in years |
| defaultPeriod | Default length of a simulation run | Years |
| dynamicUpdate | Indicates whether on-screen maps are updated during a run (slows performance) | 0=no dynamic update  1=dynamic update |
| mapUnits | Map Units of the IDU coverage | ‘feet’, ‘meters’,’degrees’,’unknown’ (default=’unknown’) |
| spatialIndexDistance | minimum distance for a spatial index to use. | 0 will disable loading of spatial index; positive values will check any existing spatial index to be sure it is sufficient, and rebuild it if necessary. Large values make take a long time to build the index. You should set this to whatever the maximum distance any of your spatial queries, Expand() functions, etc. might use. (Default=0) |
| areaCutoff | Minimum area of polygon for which a label, if defined, will be shown. | Measured in map units. Used to control label display for smaller polygons. 0 indicates to display labels regardless of polygon size |
| deltaAllocationSize | “Chunk” size, in bytes, by which new deltas are allocated | Generally, this can be ignored. 0 (default) allocates delta arrays in 32KB chunks. |
| actorDecisionElements | Indicates globally what elements of decision-making are available to actors. | This is an integer that is the sum of the following:  1 – self-interested decision making  2 – altruistic decision making  4 – global policy preference  8 – utility  32 – social network influences  For example, fi the application wants to allow 1,2 and 4, but not 8 and 32, then this flag should be set to (1+2+4)=7.  Note that 8 (utility) requires a user-defined autonomous process to populate a “UTILITY” field, and 32 (social network) requires that a social network be defined. See relevant topics belwo for more information. |
| actorDecisionMethod | Determines whether actors select policies proabilistically or always selecting the “best” policy | 1=probabilistically select policies based on scores (default)  2= always pick policy with the highest score |
| policyPrefenceWt | default weight used for policy scoring preferences during actor decision-making | 0 to 1 (the sum of actorAltruismWt, actorSelfInterestWt, and policyPrefenceWt should equal 1) |
| shuffleActorPolys | If enabled, the polygon order by which actors make decisions is randomized at each step | 0 = used fixed order (default)  1 = randomize order at each step (slightly more expensive computationally) |
| parallel | If enabled, allows evaluative models and autonomous processes to be run in parallel if multiple CPU cores are available | 0 = disable parallel execution of plugins (default)  1 = enable parallel execution of plugins |
| exportMapInterval | Default map export interval (years) during a run | -1 =do not export maps during a run (default)  Positive value = export IDU map at the specified interval during a run.  Note that this can be controlled interactively during run-time as well. Exported maps will be placed in subdirectories labeled with the scenario name and run number within the directory containing the IDU coverage |
| exportBmpPixelSize | Cell size (in map units) for exporting bitmaps (.bmp). Only used if exportMapInterval > 0 | -1 =do not export BMPs during a run (default)  Positive value = export bitmaps of the IDU map during a run at the interval specified in the ‘exportMapInterval’ attribute.  Note that this can be controlled interactively during run-time as well. Exported maps will be placed in subdirectories labeled with the scenario name and run number within the directory containing the IDU coverage |
| exportBmpCol | String indicating with column(s) of the IDU coverage should be exported. Ignored if BMP export is disabled. | Comma-separated list of field names to be exported a bitmaps. If not specified and exporting bitmaps is enabled, the currently active field is exported. |
| exportOutputs | Enables automatic export of CSV files containing model results and exposed variables at the end of a run | 0 = don’t export model outputs  1 = export model outputs  Exported files will be placed in subdirectories labeled with the scenario name and run number within the directory containing the IDU coverage |

**<layers>**

This section contains information about the coverages to load. At a minimum, one is required. The first layer specified is the IDU layer. Each layer is specified by an XML <layer> element, defined below.

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the coverage. |
| path | The full path/filename to the coverage. If the coverage in the default directory (typically where the ENVISION.exe file is located), only the file name is required. |
| type | Type of coverage: 0=shape file, 1=ASCII EXPORT grid |
| includeData | Flag indicating with the load the associated data as well as the coverage geometry: 0=don’t load data, 1=load data |
| color | Color of the polygon edges. Specified as a RGB triplet, e.g. 255,255,255 indicates white |
| records | Specifies how many records to load. -1=all, positive integer indicates number of records |
| initField | Name of field (column) to display initially. Blank defaults to LULC\_A if it exists, or first column otherwise. |
| overlayFlags | Indicates that an overlay layer should be created from the specified field(s). If specifying multiple overlays, separate fields with commas. Blank indicates no overlays |
| fieldInfoFile | Name of a field info file to load with the coverage. By default, a file with the name/path of the shape file and an .xml extension is loaded. |
| labelField | Field containing the value used to label polygons. If not specified, no labels are generated |
| labelFont | The face name (e.g. “Arial”) of the font used to draw the labels |
| labelSize | An integer indicating the size of the font used to draw the labels |
| labelColor | Color of the font. Specified as a RGB triplet, e.g. 255,255,255 indicates white |
| labelQuery | Spatial Query string specifying which poliygons should be labeled. Default (blank) is all polygons |

**<metagoals>**

This section specifies information about any **metagoals** defined for this application. “Metagoal” is a broad term used in Envision to define goals that are expressed as outputs of evaluative models, policy scores, and actor values (see Chapter 4 for details). While not required, metagoals are a useful mechanism to connect actors, policies, and evaluative models. Each metagoal has its own **<metagoal>** tag as defined below.

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the metagoal. It can contain spaces. (required) |
| model | name of associated eval model, if used in altruistic decision-making. This must correspond to entry in the <models> section. If this metagoal is not used in altruistic decision-making, it is optional |
| decisionUse | 0=don't use in decision  1=use in actor self-interest (value) decision only  2=use in altruistic decision only  3=use in both  This is a required attribute. |

**<visualizers>**

This section specifies information about the visualizers used by the application. Each visualizer is specified in its own **<visualizer>** tag; each entry consist entirely of non-whitespace characters, with the exception of “Init Function Info” and “Name”. Visualizer descriptors include the following attributes:

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the visualizer. |
| path | The full path/filename to the DLL. If the DLL is in the default directory (typically where the ENVISION.exe file is located), only the file name is required. |
| id | A unique integer identifier for the visualizer. The ID is used to distinguish multiple visualizers contained with a common DLL. |
| use | 0=don’t use this visualizer, 1=use this visualizer |
| type | A flag indicating how the visualizer is used in ENVISION. The flag can be:  1= this is an input visualizer  2 = this is a run time visualizer  4 = this is a post-run visualizer |
| initInfo | A string that is passed to the visualizer during its Init() call. See Chapter 5 for more information on model interfaces. |

**<models>**

This section specifies information about the landscape evaluation models used by the application. Each model is on its own **<model>** tag; each entry consist entirely of non-whitespace characters, with the exception of “Init Function Info” and “Name”. Model descriptors include the following attributes:

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the model. |
| path | The full path/filename to the DLL. If the DLL is in the default directory (typically where the ENVISION.exe file is located), only the file name is required. |
| id | A unique integer identifier for the model. The ID is used to distinguish multiple models contained with a common DLL. |
| use | 0=don’t use this model, 1=use this model when running scenarios |
| ~~decisionUse~~ | ~~A flag indicating how the model is used in ENVISION. The decision use flag can be: 0=don't use in decision,  1=use in actor self-interest (value) decision only  2=use in altruistic decision only  3=use in both~~ |
| freq | Frequency (years) that this model is run. Default is 1. Not currently used by Envision |
| showInResults | Indicates whether ENVISION collects outputs from the model to present in the “Evaluative Model Results” plot/table after a run is made. The Show in Results flag can be:  0=don't show anywhere  1=show everywhere  2=show where unscaled (beyond [-3..3]) can be shown. |
| fieldName | Specifies whether the model reserves a column in the IDU database for it’s own use. If a column name is specified, a column in the IDU databased is reserved for this model. If the column doesn’t exist, it is created. IF left blank, no column is reserved. |
| initInfo | A string that is passed to the model during its Init() call. See Chapter 5 for more information on model interfaces. |
| dependencies | Comma-separated list of model names that this model is dependent on. Only used if parallel execution of models/processes is specified in the <settings> section |
| initRunOnStartup | Call InitRun() after Init() during startup, if exported from the plug-in. This is useful if your Init() and InitRun() code are the same. Default is to disable this feature |

**<autonomous\_processes>**

This section specifies information about the autonomous process models used by the application. Each autonomous process is it’s own **<autonomous\_process>** tag; each entry consist entirely of non-whitespace characters. Attributes include the following fields:

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the autonomous process. |
| path | The full path/filename to the DLL. If the DLL is in the default directory (typically where the ENVISION.exe file is located), only the file name is required. |
| id | A unique integer identifier for the autonomous process. The ID is used to distinguish multiple autonomous processes contained with a common DLL. |
| timing | Specifies whether the autonomous process is executed at the beginning of a yearly step (0) or at the end of a yearly step (1) |
| freq | Frequency (years) that this model is run. Default is 1. Not currently used by Envision |
| sandbox | A flag indicating whether this autonomous process is included when ENVISION does automatic Policy Effectiveness calculations via ENVISION’s “Sandboxing” capability. 0=no, 1=yes. |
| fieldName | Specifies whether the model reserves a column in the IDU database for it’s own use. If a column name is specified, a column in the IDU databased is reserved for this model. If the column doesn’t exist, it is created. If left blank, no column is reserved. |
| initInfo | A string that is passed to the autonomous process during its Init() call. See Chapter 6 for more information on autonomous process interfaces. |
| dependencies | Comma-separated list of model names that this model is dependent on. Only used if parallel execution of models/processes is specified in the <settings> section |

**<app\_vars>**

This section specifies information about any **application variables** defined for this application. Application Variables (AppVars for short) are variables that are specific to an application and can be “seen” throughout Envision, meaning they are accessible by plug-ins, policies, queries, etc. They are a convenient way to share information between these components. AppVars are computable expressions; the supported syntax for mathematical expressions is given in Appendix 4. They can be “*globally*” defined, which means they are calculated independently of any specific IDU, or “*locally*” defined, in which case they are calculated for each IDU – Envision determines this automatically based on whether the expression for the AppVar contains any IDU references (field names included in the expression). Each AppVar is on its own **<app\_var>** tag as defined below.

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| name | Name of the AppVar. This identifies the AppVar, and should therefore be unique. It must start with a character or underscore. Beyond the first character, valid characters include a-z and A-Z, underscores, and numbers only; no spaces or special characters (e.g. periods) are allowed. Valid examples include “hello”, MyVariable, \_MyVariable, my\_variable (required) |
| description | Text description of the variable (optional) |
| col | Field name to populate with the computed value of this expression. Only valid for local expressions. If the field doesn’t exist in the IDU coverage, it will automatically be added. If you don’t specify a field name, the variable value will still be calculated, but the IDU coverage will not be populated with this information. (optional) |
| timing | Specifies whether the AppVar is evaluated the beginning and/or end of a yearly step.  1 = evaluate at the beginning of the time step  2 = evaluate at the end of the time step  3 = evaluate at the beginning and end of the time step  This is a requred attribute |

**<lulcTree>**

The <lulcTree> section contains the Land Use/Land Cover class hierarchy. Up to four tiers can be defined. These are, by convention, not requirement, generally referred to as LULC\_A, the most highly aggregated version of the LULC classes, through LULC\_D, the most highly articulate LULC classes. Each more detailed class must nest within the classes defined at the level above it, forming a LULC “Tree”. Applications must define at least one tier of LULC descriptors. Any LULC classification scheme can be used, as long as it is placed within this hierarchical tree structure.

An example LulcTree file (coded with the LULC classes defined by the 2001 National Land Cover Database (NLCD) classification scheme) is given in Appendix 1.

The format of the lulcTree XML is straightforward. Each tier of the classification is defined within an Xml **<classification>** tag. Classification tag attributes include the name of the class, which corresponds to the database column in the IDU shape coverage that holds this classification code for each IDU polygon, and the *level* of the class, corresponding to the level of this classification in the LULC three-tier hierarchy.. For example, if the classification corresponds to a database column called LULC\_A representing the top tier (most aggregated) level in the lulcTree, then the classification **name** attribute will be “LULC\_A”, and the **level** attribute will be “1”.

Within a classification, each LULC class is defined with a **<lulc>** tag. This tag defines specific lulc codes for a given level in the classification hierarchy. Each **<lulc>** tag has attributes giving the classification code for the class (**id**), and a text description of the class (**name**). Additionally, with the exception of the top level (level=1) classes, each **<lulc>** include the classification code of its parent class in the LULC hierarchy (**parentID**).

|  |  |  |
| --- | --- | --- |
| **XML Element** | **Attribute** | **Description** |
| <classification> | name | Database column name for this level of classification (e.g. LULC\_A) |
| level | The level of his classification – 1=top, 2=second, 3=third, etc. |
| <lulc> | id | The classification code for this class. It should be unique within its LULC classification level, e.g. each LULC\_A class ID should be unique. This is the code that is stored in the IDU database and referred to in Policy Site Attributes and Outcome strings. (required) |
| parentID | The classification code for the parent of this class. (not included in top-level classes, required for send and below tier classes) |
| name | Land Use/Land Cover class name (optional, this generally is specified in a fieldInfo file instead of here) |

Additional top-level tags in the project files include those specifying policies (**<policies>**), actors (**<actors>**) and social network (**<social\_network>**) – these are all optional and are defined in their respective sections below.

## The Integrated Decision Unit (IDU) Shape File

The Integrated Decision Unit (or IDU) shape file is a GIS coverage that contains the polygons that form the decision units for ENVISION. It must conform to ESRI’s Shape File specification. The polygon geometry can be defined by the application developer at what ever scale and with whatever geometry is appropriate for the application. If the decisions to be modeled are made at a tax lot (parcel) level, then parcels might be an appropriate IDU. If some other decision unit is appropriate, the IDU geometry should reflect that.

The IDU Shape file to use for a given application of ENVISION is specified in the **[layers]** section of the Project file. Note that multiple layers can be specified in this section; the first is assumed to be the IDU coverage. All IDU attributes must be contained in the IDU coverage – multiple IDU coverages in a single application are not currently supported in ENVISION.

It is important to emphasize that ENVISION can ONLY deal with landscape information that is represented in the IDU coverage. **The IDU coverage should contain any attributes that are needed by any of the following**:

* Policy Site Attribute Query
* Policy Outcomes
* Evaluative Model
* Autonomous Processes
* Visualizers

Additionally, there are are a small number of attibute fields required to be present in the IDU coverage, specified as follows. These can be added in with any software capable of manipulating Shape file databases; additionally ENVISION has the ability to add these fields automatically or manually (see *Data Preparation Functions* for more information.)

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Data Type** | **Description** |
| LULC\_A LULC\_B LULC\_C | Integer | Land Use/Land Cover (LULC) classes. ENVISION employs a n-tier hierarchy of LULC classes, with up to four tiers specified. LULC\_A is the top-tier class, representing the most aggregated LULC descriptors. LULC\_B is a more articulated version, and LULC\_C is the most articulated version of the LULC classes. This concept of an LULC hierarchy is employed to allow Site Attributes and Outcomes to be specified at various levels of generality. The specific classes used in each classification level can be defined by the application in a comma-delimited file specified in the **lulcTree** setting in the Project file. If only LULC\_C attributes are specified in the IDU coverage, ENVISION can populate the LULC\_B and LULC\_A fields automatically – see *Data Preparation Functions* in the manual for more information. LULC\_A is required at a mimimum; LULC\_B and LULC\_C are optional. |
| AREA | Numeric, Single or Double | Contains the area of the IDU. This can be populated automatically by ENVISION - – see *Data Preparation Functions* in the manual for more information. |
| POLICY | Integer | ENVISION uses this column to store the most recent Policy ID to be applied to an IDU. It does not need to be populated by the developer; it just needs to exist in the IDU coverage. |
| POLICYAPPS | Integer | ENVISION uses this column to store the most number of policy applications made to an IDU. It does not need to be populated by the developer; it just needs to exist in the IDU coverage. |
| LASTD | Integer | ENVISION uses this column to store the year in which the last actor decision was made for this IDU. It does not need to be populated by the developer; it just needs to exist in the IDU coverage. |
| NEXTD | Integer | ENVISION uses this column to store the year in which the next actor decision is scheduled for this IDU. It does not need to be populated by the developer; it just needs to exist in the IDU coverage. |
| STARTLULC | Integer | ENVISION uses this column to "remember" the initial LULC\_C value of the IDU. This columns needs to be present, but ENVISION populates it automatically. |
| UTILITYID | Integer | This is only used if Actor decision-making based on utility is enabled. In that case, an autononous processes should be defined that populates this field with the policyID of the policy that maximizes utility for the given IDU. |

**ENVISION’s Built-in Data Preparation Functions**

ENVISION contains a number of built-in data preparation functions that simplify and automate certain aspects of IDU coverage development. These are listed under **Data Preparation** tab on ENVISIONs main menu, and are described below:

|  |  |
| --- | --- |
| **Menu Option** | **Description** |
| Merge With a Different Database | Allow the columns of a separate spatial database (dbf) to be merged into the IDU coverage. Individual fields to be merged can be selected. The database to be merged should have matching geometry and IDU’s. |
| Add Field(s) | Adds a new (unpopulated) column (field) to the IDU database. |
| Remove Field(s) | Removes column(s) (field(s)) from the IDU database. |
| Calculate Field | Populates a field with a value computed from other fields. |
| LULC Hierarchy | Examines the LULC\_C field and populates the LULC\_B and LULC\_A fields based on the current LulcTree hierarchy information. |
| Distance To | Allows population of “closest distance to” fields by calculating the distance from every point or the centroid of every polygon in a target layer to the points or polygon edges in a source layer. For example, the distance to a stream or business district for each IDU can be calculated and populate an approriate field, as long as a query identifying a stream or business district in some layer can be formulated. |
| Adjacent To | Allows population of a database field with the index of “adjacent” polygons. |
| Area | Computes the area of each IDU polygon and sores the result in a field called AREA. It will create this field in the IDU database if it doesn’t currently exist. |
| Project to 3D | In conjunction with an ESRI ASCII GRID DEM file, projects the IDU to 3D and allows storage of the IDU coverage as a 3D shape file. |
| Populate AREA Field | Populates the AREA field with an IDU’s area, computed from the IDU geometry in the units of the shape file. |

# Chapter 3. Output Files

ENVISION can generates a number of output files over the course of a session. Generally, four types of files are created:

1. Maps of the current state of the IDU coverage at periodic intervals during the course of a simulation, in the form of ESRI shape files that contain all attributes in the IDU coverage. These are generated if the ‘**exportMaps’** attribute in the <**settings**> section of the project file is set to ‘1’. The interval that these maps are created is defined by the ‘**exportMapInterval’** attribute in the <**settings**> section of the project file. Both can be set from the Envision user interface (**Run** tab) as well.
2. Comma-delimited text files (CSV files) – these contain time series summaries for all internal Envision variables and any output variables exposed by plugins specified in the current project. These are generated for each Envision run.
3. Pivot-table ready
4. The Delta Array, in the form of a CSV file contain information on each delta generated during the course of a run.

In addition to these outputs, individual plug-ins may generate additional output files – see the documentation for individual plug-ins for more information.

All outputs are generated in a defined directory structure as follows:

|  |  |  |
| --- | --- | --- |
| Type | Directory | Filename |
| Time Series Maps | ***{IDU Directory}/***Outputs/ ***{ScenarioName}***/Run***{N}***/ | ***{ScenarioName}***\_Run***{N}***\_Year***{Y}***.shp ***{ScenarioName}***\_Run***{N}***\_Year***{Y}***.dbf ***{ScenarioName}***\_Run***{N}***\_Year***{Y}***.shx |
| Output Variable CSV files | ***{IDU Directory}/***Outputs/ ***{ScenarioName}***/Run***{N}***/ | {Model\_and\_AppVar\_Outputs\_ ***{ScenarioName}***\_ Run***{N}***.csv  ***{Output\_Specific\_Filename}***\_ ***{ScenarioName}***\_ Run***{N}***.csv |
| Pivot Table-ready CSV files | ***{IDU Directory}/***Outputs/ | ***{Output\_Specific\_Filename}***\_pivot\_***{Timestamp}***.csv |
| Delta Array CSV file | ***{IDU Directory}/***Outputs/ ***{ScenarioName}***/Run***{N}***/ | DeltaArray\_***{ScenarioName}***\_Run***{N}***.csv |

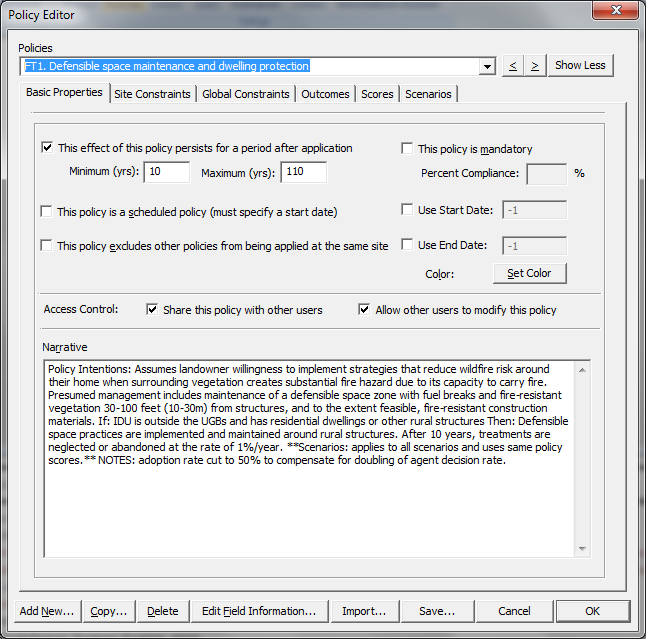
# Chapter 4. Policies

Policies in ENVISION represent the fundamental unit of decision-making – they define and constrain the range of possible decisions an actor can make, and define the outcomes resulting from an actor selecting and applying the policy to an IDU. Policies can be thought of as “strategies” or “decision rules” available to actors. Policies have a number of characteristics in ENVISION: where in a landscape they are applicable, what outcomes are generated in response to an actor choosing to select and apply a policy to an IDU, what management goals the policy is intended to be responsive to, and other characteristics. Policies definitions must be supportable by the underlying landscape attributes contained in the IDU coverage; for example, if a policy is to be constrained to only apply in riparian areas, then the IDU coverage must have a representation of what defines a riparian area. Similarly, if a policy outcome is to change a site from undeveloped to developed, then the IDU coverage must have an attribute or attributes consistent with that representation.

**Landscape policies are decisions or plans of action for accomplishing desired outcomes (Lackey 2006).**

from:

Lackey, R.T. 2006. Axioms of ecological policy. Fisheries. 31(6): 286-290.

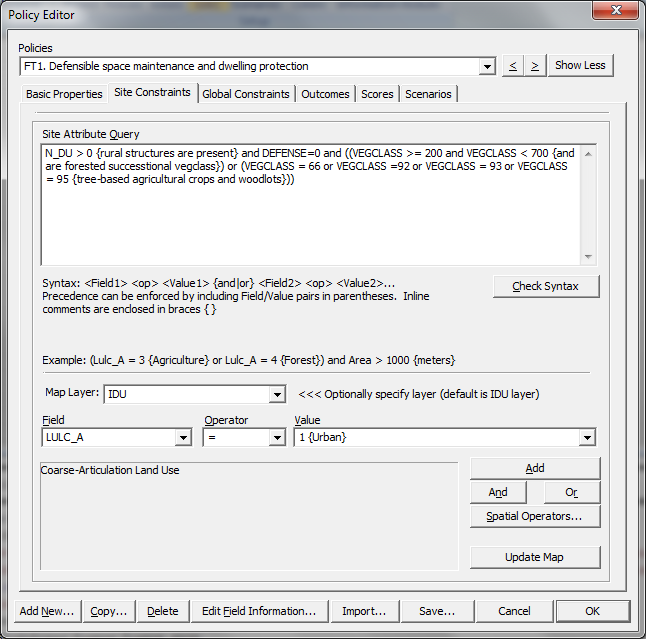
ENVISION can contain any number of policies. Users can define new policies, or edit/delete existing policies, using the Policy Editor. The Policy Editor is available from the **Edit Policies** option on most ot the main ribbon tabs. Policies are stored as XML in either the project (.ENVX) file or as a separate XML document, as indicated in the **<policies>** section of the project file.

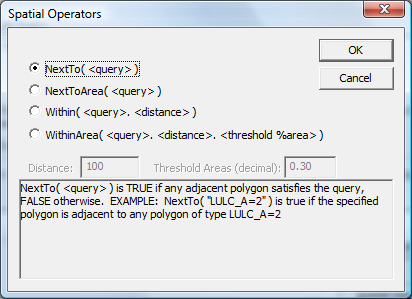
Policies in ENVISION have a number of attributes that can be specified by a user or application developer through the Policy Editor. The Policy Editor presents these attributes on a series of tabs in a dialog box. The attributes, organized by tab, are described below.

**Basic Properties Tab:** This tab, shown to the right, allows specification of some basic policy attributes as described in the following table. Additionally, it allows the creation of new policies, cloning of existing policies for modification, deletion of policies, and the selection of the color used to represent the policy application in thematic maps and plots.

|  |  |
| --- | --- |
| **Policy Attribute** | **Description** |
| Effectiveness Period | How long (in years) this policy remains in effect after an actor selects it, expressed as a range between some minimum and maximum number of years. This is useful if you want to specify a temporal “footprint” for a policy. It is particularly useful in combination with the “Exclusive” attribute describe below. To utilize this attribute, check the “The effectiveness of this policy persists for a period after application” checkbox and specify the minimum and maximum persistence periods. The actual effectiveness period of a given policy application is selected stochastically from a uniform distribution between the minimum and maximum specified. |
| Is Mandatory | Specifies whether this policy is mandatory. Mandatory policies are always applied by actors on IDU’s where the site attributes are satisfied |
| Start Date | Specifies, if checked, that the policy does not become available to actors until the specified number of years have passed from the beginning of the analysis period. |
| End Date | Specifies, if checked, that the policy is no longer available to actors after the specified number of years have passed from the beginning of the analysis period. |
| Is Scheduled Policy | Indicates, if checked, that the policy occurs automatically (independent of an actor decision) at the specified point in time (specified as the start date, the number of years since the beginning of the analysis period) |
| Is Exclusive | Indicates that this policy, when selected by an actor and applied to an IDU, prevents any other policies from being applied to the IDU until the policy expires, as indicated by its effective period. |
| Shared Policy | Part of the editing control system for policies. If checked, the policy is available to any user of the policy database specified in the Project file; otherwise, only the author of the policy can use it in an analysis. |
| Editable Policy | Part of the editing control system for policies. If checked, the policy can only be edited by the author of the policy; otherwise, any user of the policy database specified in the Project file can edit/modify the policy. |
| Narrative | Textual description of the policy. This can be used to document the policy or provide other information about the policy |

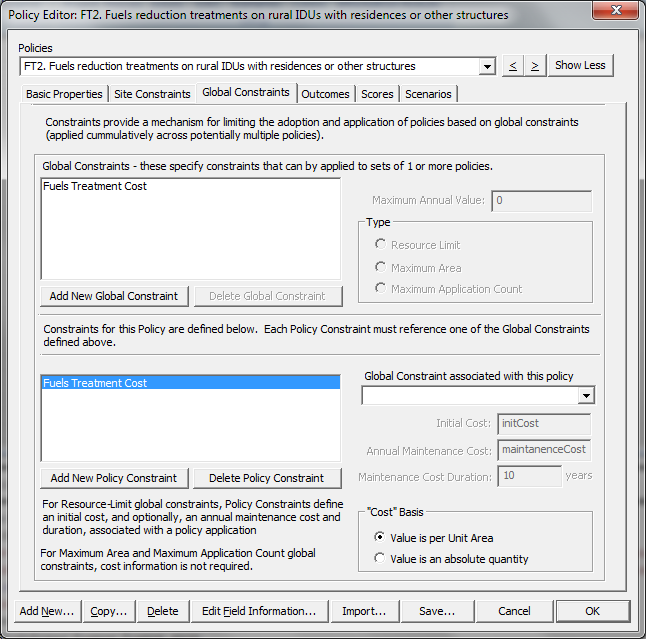
**Site Attributes Tab:** This tab, shown below, allows the construction of spatial queries that specify where in the landscape the policy may be applied. ENVISION has a built-in query language that is used in the Site Attribute Specification. The query allows a range of logical and spatial operators, and is described more fully in Appendix 3. If ENVISION’s “Map” tab is active, the <Update Map> button will execute the current Site Attribute Query and show where on the map the Policy is potentially applicable; this is a very useful feature for determining whether the Site Attribute Query is meeting your requirements for the policy.

To specify a Site Attribute Query, the Policy Editor provides a query builder. Queries can be types directly into the Site Attribute Query field of the Policy Editor, or can be constructed using the builder functions provided. Field/Value attributes are constructed by selecting the desired Field/Operator/Value sets and pushing the “Add” button, which insert the specified triplet at the current insertion point in the query string.



Similarly, a spatial operator can be inserted into the Site Attribute Query by selecting the <Spatial Operators” button, which brings up the Spatial Operators dialog box. The arguments to the spatial operators include a query string that defines the target of the spatial operator, using syntax consistent with any other ENVISION spatial query.

**Global Constraints Tab:** Envision supports the concept of “global constraints” on policy applications, expressed in terms of “costs” of an individual policy application, and one or more budgets (global constraints) defined globally for a set of policies. The basic idea is that if a policy has an associated global constraint, its adoption will be limited when that constraint is met. These constraints are defined on an annual basis. For example, suppose we have a budget for restoration, and a set of policies whose adoption rates are constrained by this budget. We would then define a global constraint (whatever the annual budget is) and policy constraints (defining the cost of adopting the policy). As Actors adopt policies, the cost of those policies are accumulated against the budget defined by the global constraint. One the global constraint is met, the policy is no longer available for adoption in that time step.

Costs consist of two elements: 1) The initial cost, accrued at the time of policy adoption, and 2) an annual maintenance cost, accrued each year for a specified maintenance duration. Cost can be specified on a per unit area basis, or a fixed quantity for each adoption event.

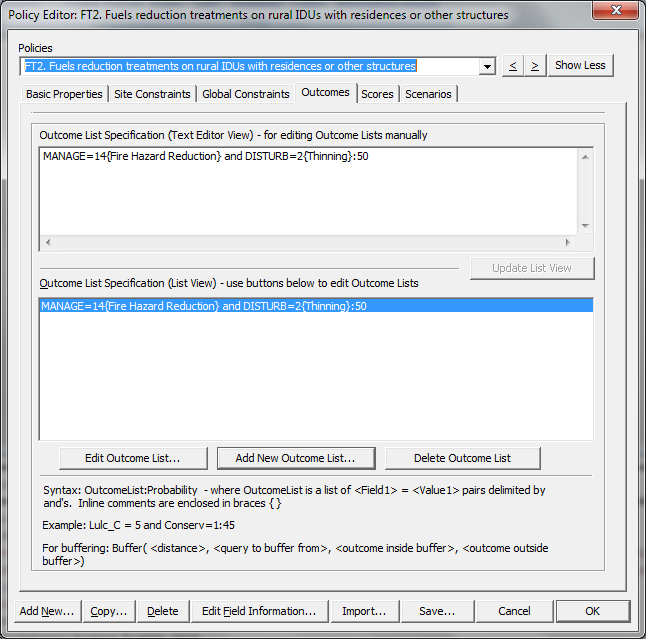
Any number of global constraints can be defined, and a policy can have multiple constraints associated with it, that can reference one or more global constraints.

**Outcomes Tab:** The Outcomes Tab (shown below) specifies what changes to the IDU database can occur if an actor selects and applies a policy to an IDU. ENVISION employs a flexible grammar and syntax to define one or more outcomes for a policy. ***Alternative outcomes*** can be specified, with an associated probability, when more than one distinct set of outcomes can occur when a policy is applied by an actor. For each alternative outcome specified, any number of resulting attribute changes in the underlying IDU data can be defined.

The grammar and syntax for specifying outcomes is as follows: 1) an individual outcome is defined by a sequence of one or more Field=Value pairs, separated by the “and” keyword. A probability of occurrence can optionally be specified by suffixing the outcome string with a colon, followed by a probability value between 0-100. If this is not specified, a probability of 100 percent is assumed. Inline comments (ignored by the outcome compiler) can be specified in braces (“{ } “) to improve readability of the outcomes. For example, assume a riparian policy results in a land use change to “shrublands” (stored in the LULC\_C field in the IDU database with an attribute code of 87) and sets the IDU into conservation status (indicated by setting the CONSERV field to 1), and that this always occurs. The outcome string in this case would be:

LULC\_C=87 {shrub} and CONSERV=1

To specify that this outcome only occurs 50 percent of the time (instead of the 100 percent assumed above), the outcome would be modified to read:

LULC\_C=87 {shrub} and CONSERV=1:50

Note that any number of Field=Value pairs can be specified. Specified fields MUST exist in the IDU database, or the outcome compiler will generate a compilation error. Further note that in this example, nothing happens 50 percent of the time this policy is selected and applied by an actor.

2) To specify that alternative possible outcomes may results from a policy application, individual outcomes as described above are specified, with each individual outcome separated by a semicolon (“;”). For example, to extend the example above to include an additional possible outcome of land use changing to grass buffers instead of shrubs 25 percent of the time, the outcome string becomes:

LULC\_C=87 {shrub} and CONSERV=1:75; LULC\_C=92 {grass} and CONSERV=1: 25

Any number of multiple outcomes can be specified in this manner. If the probabilities sum to less than 100, the difference (100-sum) become the probability of no outcome changes occurring. If the probabilities sum to over 100, they are normalized to sum to 100.

*Expanding the effect of a policy application:*

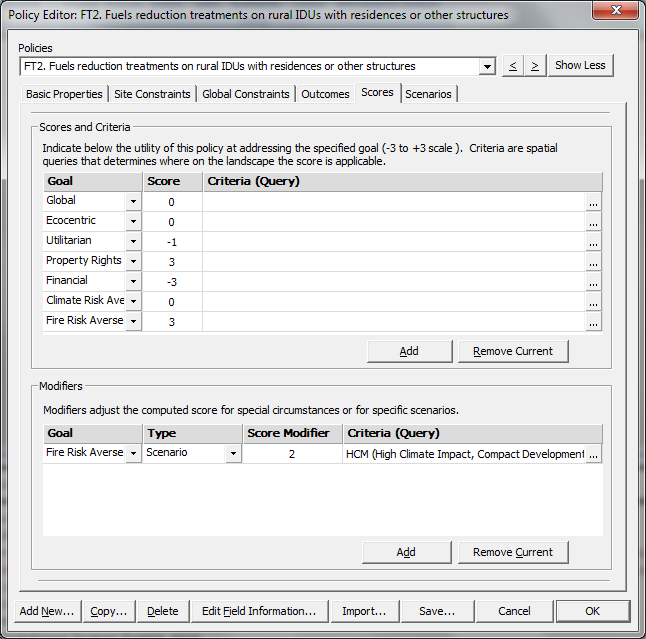
*Generating Buffers using outcome statements:* Buffers are strips of lands, frequently (but not exclusively) along river systems. A powerful capability of ENVISION is the ability to dynamically create buffers during policy application in a run. However, dynamically creating buffers can significantly increase the complexity of results analysis and slow down program execution speed. Incorporating buffering operations into ENVISION is accomplished using the **Buffer** function in place of a Field=Value pair. The **Buffer** function takes four arguments as follows:

***Buffer( width, query, insideOutcome, outsideOutcome)***

where: ***width*** is the width of the buffer, in units consistent with the IDU database.  
***query*** is a spatial query (see Appendix 3 for syntax) that defines what polygon(s) will be used to buffer the current IDU with.   
***insideOutcome*** is the outcome string (see above) defining outcomes inside the newly created buffer. This can be left blank if no change occurs inside the buffer.  
***outsideOutcome*** is the outcome string (see above) defining outcomes outside the newly created buffer. This can be left blank if no change occurs outside the buffer

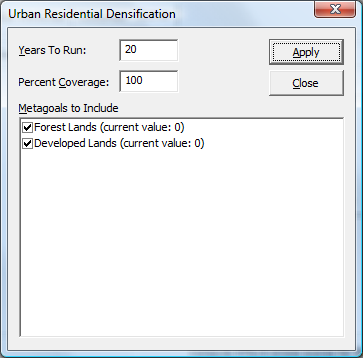
For example, assume a policy creates a 10m riparian buffer in an IDU that is adjacent to a river; inside the buffer, shrubs are planted, while outside the buffer no change is made. The outcome syntax is:

Buffer( 10, LULC\_C=91 {river}, LULC\_C=87 {shrub}, , )

**Effectiveness Scores Tab:** This tab (shown below) allows the specification of the policy effectiveness for the metagoals defined for this application. Metagoals are indicated by flagging the “Decision Use” attribute for Evaluative Models, indicating they are used in self-interested or altruistic decision making in the Project file (see Project file description above). Effectiveness scores indicate roughly the expected utility of the policy at addressing the indicated metagoal(s). Effectiveness values are specified using slider bars: One slider bar is generated for each of the metagoal specified in the application. Effectiveness scores, like all scores in ENVISION, are specified in a -3 to +3 range, where -3 indicated the policy is counterproductive to the goal, 0 indicates the policy is neutral to accomplishing the goal, and +3 indicates the policy is very effective at addressing the goal.

Effectiveness scores can be computed by ENVISION, or set manually based on expert knowledge of the policy effects. To have ENVISION automatically compute the scores, select the “Compute Effectiveness Scores for this Policy” button. ENVISION’s process for this involves running an analysis with and without the policy being applied to the landscape, and assessing the change in the metagoal landscape metrics between these runs. When computing these scores automatically, you can specify the parameters of analysis as follows:

|  |  |
| --- | --- |
| **Analysis Setting** | **Description** |
| Years to Run | The length of the analysis period used to define a parameter. This should be roughly the expected time the effects of the policy take to be realized. |
| Percent Coverage | The portion of the possible sites (as defined by the Site Attribute Query) that the policy is applied to for the analysis. |
| Metagoals to Include | Specifies which metagoals to compute effectiveness scores for. |



This method works reasonably well to get first approximations of policy effectiveness score settings. However, these scores generally need to be reviewed and adjusted to ensure they reflect the policy author’s intentions.

# Chapter 5. Actors and Decision-making

## Basic Concepts

The concept of Actors is fundamental to ENVISION. Actors are the entities in ENVISION that ultimately make decisions about policy selection and land use change that ENVISION propagates through the landscape. Actors are characterized by 1) the *values* the hold about landscape productions, and 2) the *locations* (IDU’s) for which they have responsibilities for making policy selection and implementation. Actors make these decisions using a combination of two primary approaches: 1) *altruistic* decision-making, based on landscape feedbacks that reflect landscape-level scarcities, an 2) *self-interested* decision-making, based on alignment of policy intentions with their own values.

There is a relationship between actor values, landscape scarcity metrics, and policy intentions, as shown in the figure to the right. Each is measured on a -3 to +3 scale, reflecting the strength of the actor value, the scarcity/abundance of the landscape production, and expected efficacy (intention) of the policy, respectively. These are collectively referred to as “goals”, and are specified in the Project file under the **<models>** section in the **Decision Use** field when specifying the model. Usage of this field requires understanding the actor decision-making process; this is described below.

**Actor Decision Frequency:** Actors make decisions only at certain times during a simulation. This timing is determined by two factors: 1) is the IDU being considered available for a decision (e.g. is it not “locked up” by a previous policy application that precludes further policy application at the current time), and 2) is the Actor “ready” to make a decision. The latter is determined using the Actor’s *decision frequency* setting, measured in years. This setting is used to probabilistically test, using a uniform sampling distribution, whether at any given time a policy selection decision on a particular IDU is to be made. For example, a decision frequency of 10 means that on average an Actor will make a policy selection decision about a particular IDU once every ten years.

**Actor Decision Making Algorithm**: Actors make decisions about policy selection based on several elements of decision-making. Each decision element can be enabled or disabled based on the “actorDecisionElements” setting in the <settings> section of the Project file. Additionally, any active decision element can be weighted to reflect the relative importance of that element compared to the other elements – these are combined into an overall weighted score that reflects the influence of each decision element on an actor’s decision-making. The decision elements are summarized in the table below:

|  |  |  |
| --- | --- | --- |
| **Decision Element** | **Description** | **Method of Calculation** |
| Actor Weights | Reflects the alignment of policy intentions with actor values. Self-interested actors tend to select policies that are well aligned with their value systems, regardless of emerging scarcities in the landscape. | Compute the Euclidean distance in Goal Space between the Actor Value vector and the Policy Intention vector. Normalize to the diagonal of the goal space hypercube. Each dimension of the goal space is defined by an evaluative model whose **Decision Use** flag is 1 or 3 (indicating it is used in self-interested decision-making.) |
| Landscape Feedbacks | Reflects the alignment of the policy intentions with landscape scarcities. Altruistic actors tend to select policies that are responsive to landscape scarcities as defined by the evaluative models | For each dimension in goal space, normal the Policy Intention value (e) and the Landscape Metric Abundance value (a) to scale between 0-1. Compute an unbiased score using the following formula:  score = e \* (((e-a)+1)/2) \* (1-a))  Bias that score using the metagoal weights specified for the scenario for each goal using simple weighted averaging. Each dimension in goal space is defined by an evaluative model whose **Decision Use** flag is 2 or 3 (indicating it is used in altruistic decision-making.) |
| Policy Preference | Reflects global, goal-specific, site-specific, and scenario-specific policy scores, defined for each policy in the policy’s “scoring” settings. | Compute a policy preference score based on global policy preferences, and modify that score based on modification defined in the policy. |
| Utility | Reflects the concept that available policies provide some level of “utility”, in the economic sense, to an actor. This utility is determined by an application-specific autonomous process. | Assumes an external process populates a field in the IDU coverage called “UTILITYID” with the policyID of the policy that provides the maximum utility to the actor. |
| Social Network | Reflects the influence of an actor’s social network on an actor’s decision choice. Requires the definition of social network – see section below | See section below on social networks. |

The decision-making process for each IDU at each time step then becomes:

Determine whether an Actor is going to make a decision, based on the availability of the IDU and the Actor’s Decision Frequency. If a decision is to be made, proceed as follows:

1. Select all relevant policies that are applicable to the site using the policy’s site attribute specification.
2. Score each policy on self-interest, altruistic and policy preference aspects.
3. Weight the resulting scores using the global actors weights/landscape feedbacks/policy preference/utility/social network settings in the scenario (or set in the Model Setup dialog box).
4. Probabilistically sample the selected policies, preferentially selecting those policies with higher score, or deterministically select (depending on the “actorDecisionMethod” setting in the <settings> section of the Project file) a single policy for application.
5. Apply the policy to the IDU, propagating that policy’s outcome(s) to the IDU. If the selected policy has alternative outcomes, probabilistically select one of the alternative outcomes based on their likelihoods.

## Social Networks

Social Networks are a mechanism supported by Envision for 1) representing social relationships between actors and external entities that have influence over an actor’s decision-making.

# Chapter 6. Evaluative Models

Evaluative models, at a fundamental level, provide ENVISION with metrics of landscape production. They typically take a landscape as input (in the form of a pointer to ENVISION’s representation of the current IDU coverage), and compute a “scarcity” metric describing how that landscape is performing at providing the metric of interest. To allow these models to be “plugged in” to ENVISION at runtime, ENVISION defines a set of interface functions that developers can implement to allow communication and flow of data to occur between the model and ENVISION. These interface functions, most of which are optional to implement, are provided in the table below.

|  |  |
| --- | --- |
| **Function( args )** | **Description** |
| EMInit( EnvContext\*, CString\* pArgStr ) | This function is called when ENVISION first starts up and loads the model. It is passed the argument string specified for the model in the Project file, which the function can interpret as necessary. This function should perform any necessary one-time initialization (optional) |
| EMInitRun( EnvContext\*, bool useInitialSeed ) | This function is called at the beginning of each run. It should perform any necessary initialization to prepare the model for a new run. The “useInitSeed” argument indicates wheterh the model should reset any random number generators to their original seed values (provides reproducibility of the random streams). (optional) |
| EMRun( EnvContext\* ) | Main Model entry point. This function is called by ENVISION repeatedly during an analysis run, at an interval specified by the user through the model setup dialog box. This function receives a current IDU coverage, and its primary purpose is to evaluate that map for the metric encompassed by the model, returning a scaled result (between -3 to +3) indicating the scarcity (-3) or abundance (+3) of that metric, based on the current ENVISION context. (required) |
| EMEndRun( EnvContext\* ) | Called at the end of a run (optional) |
| EMShowProperties( HWND hParent, int ID ) | This function is called to signal that the Model should pop up a dialog box or property sheet to allow setup of the Model. The ***parent*** argument is a handle to the parent window for the dialog; the ***ID*** is the model ID given in the Project file. (optional) |
| EMInputVar( int ID, MODEL\_VAR\*\* ) | This function is called to allow Models to “expose” input variables to ENVISION for use in defining scenarios. The ***ID*** is the model ID specified in the Project file; the ***modelVar*** argument is a reference to the address of an array of MODE\_VAR structures defined in the Model’s DLL, one MODEL\_VAR for each variable to be exposed. The MODEL\_VAR structure is described below. This function should return the number of variables exposed and the address of the MODEL\_VAR array in the ***modelVar*** argument. (optional) |
| EMOutputVar( int ID, MODEL\_VAR\*\* ) | This function is called to allow Models to “expose” variables to ENVISION for use in collecting output. The ***ID*** is the model ID specified in the Project file; the ***modelVar*** argument is a reference to the address of an array of MODE\_VAR structures defined in the Model’s DLL, one MODEL\_VAR for each variable to be exposed. The MODEL\_VAR structure is described below. This function should return the number of variables exposed and the address of the MODEL\_VAR array in the ***modelVar*** argument. (optional) |

Evaluative models in ENVISION must be written in C++ and compiled into a DLL. The Interface functions must be exposed as “exported” functions in the DLL. A single DLL can contain one or more evaluative models; if a single DLL defines more than one model, each model is assigned a unique ID (in the ENVISION Project file) that passed to the DLL when the interface functions are invoked by ENVISION. The function signatures are provided in <EnvContext.h> in the ENVISION SDK. A skeleton source code file is also included in the SDK. This is available on the developer’s section of the web site.

Central to the model functions is the EnvContext structure. It contains a variety of information that describes the current ENVISION state when the model function is invoked. It provides information that the model may need to compute a landscape scarcity metric.

class EnvContext

{

public:

INT\_PTR (\*ptrAddDelta)(EnvModel \*pModel, int cell, int col, int year, VData newValue, int handle );

int startYear; // year in which the simulation started

int endYear; // last year of simulation

int year; // current year

int runID; // In a multirun session, this is incremented after each run.

bool showMessages;

int logMsgLevel; // see flags in evomodel.h

const MapLayer \*pMapLayer; // pointer to the IDU layer. This is const because

// extensions generally shouldn't write to this.

ActorManager \*pActorManager; // pointer to the actor manager

PolicyManager \*pPolicyManager; // pointer to the policy manager

DeltaArray \*pDeltaArray; // pointer to the current delta array

EnvModel \*pEnvModel; // pointer to the current model

LulcTree \*pLulcTree; // pointer to the lulc tree used in the simulation

int id; // id of the module being called

int handle; // unique handle of the module

int col; // database column to populate, -1 for no column

INT\_PTR firstUnseenDelta; // models should start scanning the deltaArray here

bool \*targetPolyArray; // array of polygon indices with true/false to process (NULL=process all)

int targetPolyCount; // number of elements in targetPolyarray

ENV\_EXTENSION \*pExtensionInfo; // opaque ptr to ENV\_EVAL\_MODEL, ENV\_AUTO\_PROCESS, ENV\_VISUALIZER

// these values are set and returned by models, ignored by autonomous processes

float score; // overall score for this evaluation, -3 to +3 (unused by AP's)

float rawScore; // model-specific raw score ( not normalized )

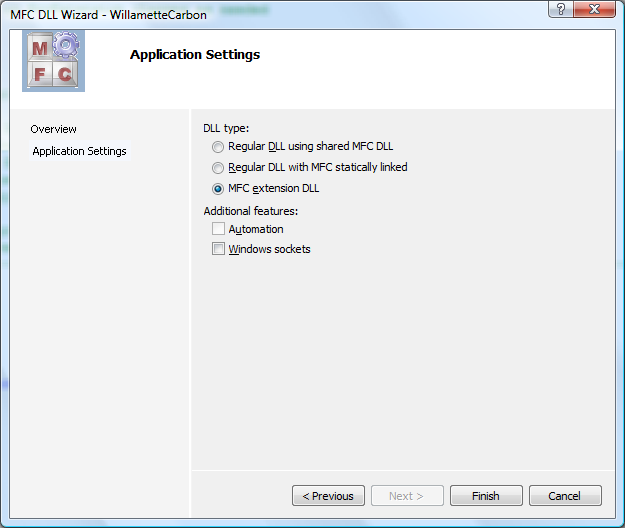
FDataObj \*pDataObj; // data object returned from the model at each time step (optional, NULL if not used)

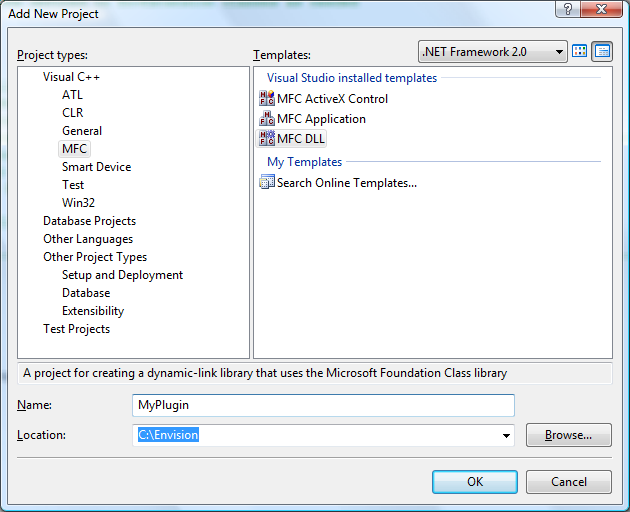
};

The EnvContext class members are defined as follows:

|  |  |
| --- | --- |
| **EnvContext Member** | **Description** |
| ptrAddDelta | This is a pointer to a function used to add Delta’s to ENVISION’s DeltaArray. See the discussion on the DeltaArray below for more information. It can be invoked from model code as:  ptrAddDelta( pEnvContext->pModel, cell, col, pEnvContext->year,  newValue, type ); |
| year | This is the current year of the run when the model function is invoked. It is used to add any Delta’s the model wants ENVISION to record, and for any other function that requires the model to know what the current simulated year is. The first year of the run is always year 0, and increments as the simulation proceeds. |
| runID | This is the ID of the current run. Useful to distinguish individual runs in a multirun situation. The first run has a runID=0; it increments for each run after that. Models generally can ignore this parameter. |
| showMessages | A Boolean value that indicates whether the model should report error/warning message through popup message boxes. During multirun situations, this is set to 0 so that models can run repeatedly without user interaction. During single run situations, it is set to 1 to allow user interaction with the model if necessary. |
| logMessageLevel | Indicates the current level for logging messages when EnvErrorMsg() or EnvWarningMsg() is called. 0=log everything, 1=log critical events, 2= log nothing |
| pMapLayer | Pointer to the IDU coverage. This is generally needed by models to get access to the underlying IDU dataset for the current point in simulated time. The MapLayer class is defined in <maplayer.h>. |
| actorManager | Pointer to the ActorManager. This class, defined in <actor.h>, manages the actors in ENVISION. Generally, models can ignore this unless they need access to Actor information not available in the IDU coverage. |
| policyManager | Pointer to the Policy Manager. This class, defined in <policy.h>, manages the policies in ENVISION. Generally, models can ignore this unless they need access to Policy information not available in the IDU coverage. |
| deltaArray | Pointer the current deltaArray. See the discussion below under the DeltaArray section. This class is defined in <deltaArray.h>. |
| pLulcTree | Pointer the LulcTree loaded by ENVISION. See the discussion above under the LulcTree for more information. This class is defined in <lulctree.h> |
| pModel | Pointer the currently executing EnvModel instance. Generally, models can ignore this unless they need access to information about the current execution context not passed in this EnvContext structure. The EnvModel class is defined in <envmodel.h>. |
| id | The ID of the model to be executed. Useful if a single DLL contains multiple models. The model ID is defined in the Project file for the application. |
| col | The database column reserved for this model (if defined in the Project file), or -1 if no column was reserved. |
| firstUnseenDelta | The first Delta in the current deltaArray this model has not seen yet, and may want to process. See the discussion below under the DeltaArray section. |
| pTargetPolyArray | An array of polygon indexes (array offsets) to be considered by this model. Not currently implemented, so this can be ignored. |
| score | This should be filled in by the model. It is the landscape metric (scarcity) score calculated by the model, expressed along a scale of -3 (scarce) to +3 (abundant) |
| rawscore | This should be filled in by the model. It is the landscape metric score calculated by the model, expressed in appropriate units (unscaled) |
| pProcessInfo | Information about the current Evaluative Model that is being invoked. It is defined in the <envmodel.h> file. |

To facilitate writing Envision plug-ins, an SDK is available on the Envision website – <http://envision.bioe.orst.edu> in the “Downloads” section. This SDK includes required include and library files, as well as a set of example files and C++ classes designed to make it relatively easy to write plug-ins. The steps required to create a basic plug-in are as follows:

**Step 1. Create an MFC Extension DLL Project.**

**** In Visual Studio, create a “New Project” of type MFC DLL. When you click OK, you will be asked to set the “Application Settings”. Indicate “MFC Extension DLL” and click Finish. The Visual Studio App Wizard will create a skeleton DLL for you. In particular, note the following files: dllmain.cpp, *myproject*.cpp, and *myproject*.def, where *myproject* is the name of the DLL you are creating We will modify these files next.

**Step 2. Modify dllmain.cpp.**

The next step involves modifying the files generated by the Visual Studio App Wizard. First, we start with the dllmain.cpp file. Start by replacing the generated dllmain.cpp with the one from the SDK. The replacement file contains default implementations and prototypes for the Envision interface functions. It also contains a number of comments starting with TODO: - these indicate places where you should modify this file, depending on your specific needs. These basically boil down to: 1) Replacing the “EnvExtExample” string with the name of the DLL you are creating, 2) including/deleting references to the model(s) and/or autonomous process(es) you are implementing in the DLL, and commenting out any EMxxx() or APxxx() functions and prototypes you don’t want to use..

**Step 3. Add exports to *myproject*.def.**

Add interface export statement to you .def file included in the project. The simplest way to do this is to copy any needed exports out of the EnvExtExample.def file included in the SDK.

**Step 4. Modify Your Project Settings.**

Right-click on your project in Visual Studio’s Solution Pane and select <Properties>. This allows you to set various project settings. Change the following, being sure to select <All Configurations>.

Required Project Settings (<All Configurations>)

1) General->Character Set: ***Not Set***

2) C/C++ ->General->Additional Include Directories: ***C:\Envision\SDK\include;*** (Note: the location may vary depending on where you installed the SDK)

3) C/C++ ->Preprocessor->Preprocessor Definitions: add ***"\_\_EXPORT\_\_=\_\_declspec( dllimport )";***  Be sure to include the quotes!

4) Linker->General->Additional Library Directories: ***C:\Envision\SDK\libs;*** (Note: the location may vary depending on where you installed the SDK)

5) Linker->Input->Additional Dependencies: ***libs.lib***

6) Build Events->Post Build Events: ***copy $(TargetPath) C:\Envision*** (Note: the location specified in the second path may vary depending on where you installed the Envision)

Note: Your setup may vary slightly based on our directory structure.

**Step 5. Implement your model(s) and process(es).**

The SDK provides two C++ classes, EnvEvalModel and EnvAutoProcess, to facilitate the creation of Envision Plug-ins. They subclass from EnvExtension. The classes provide several capabilities: 1) The provide default implemenations for all interface functions, 2) the manage input and output variables exposed by the models/processes to Envision, and 3) the provide a wrapper to facilitate making changes to the underlying map layers.

To create a model:

1. Derive a subclass from EnvEval Model
2. In the constructor of the class, call EnvExtension::AddVar() for all input and output variables;
3. Override any of EnvExtension::Init(), InitRun(), Run(), and ShowProperties() as needed.
4. When accessing the IDU map layer during processes, get the MapLayer pointer from the EnvContext object passed to the DLL; When making changes to the Map, DO NOT call the MapLayer directly; instead, use EnvExtension::AddDelta() (documented below) to make modifiction to the map.

To create an autonous process:

1. Derive a subclass from EnvAutoProcess
2. In the constructor of the class, call EnvExtension::AddVar() for all input and output variables;
3. Override any of EnvExtension::Init(), InitRun(), Run(), ProcessMap(), and ShowProperties() as needed.
4. When accessing the IDU map layer during processes, get the MapLayer pointer from the EnvContext object passed to the DLL; When making changes to the Map, DO NOT call the MapLayer directly; instead, use EnvExtension::AddDelta() (documented below) to make modifiction to the map.

**Implementation Guidelines.**

1) The IDU overage is contained in an object of class *MapLayer*. MapLayer is a class that embodies a single layer of GIS data, and provides numerous GIS operations on that data. IT also contains a pointer to a *Map* object that is a container of multiple MapLayers (depending on how many are loaded) and provides for the visual display of the maps. Ther are several common tasks involving MapLayer that are described below.

**Getting a pointer the IDU *MapLayer***: A pointer to the IDU *MapLayer* is passed in the EnvContext structure during each invocation of an interface. To access this pointer:

const MapLayer \*pLayer = pContext->pMapLayer;

**To iterate over the IDU coverage**: Use a *MapLayer::Iterator* as follows

const MapLayer \*pLayer = pContext->pMapLayer;

for ( MapLayer::Iterator idu=pLayer->Begin(); idu < pLayer->End(); idu++ )

{

float value; // int, CString, bool, etc.depending on col datatype

pLayer->GetData( idu, m\_colOfInterest, value );

// do something with the idu’s value of the specified column

}

**To iterate over the DeltaArray**: DeltaArray’s provide a complete history of every change that has been made to the IDU coverage. Further, Envision tracks the last time each model/process has “seen” the DeltaArray. This means that if a model/process want to see anychanges that have been made either from a particular years, form the beginning of the run, or since it was last “seen” by the model/process, that is easily accomplished.

DeltaArray \*deltaArray = pContext->deltaArray;// get a ptr to the delta array

INT\_PTR daSize = deltaArray->GetSize(); // use static size so deltas added

// by this process this cycle don’t

// get looked at!!!

// iterate through deltas added since last “seen”

for ( int i=pContext->firstUnseenDelta; i < daSize; ++i )

{

DELTA &delta = deltaArray->GetAt(i);

// do something with the delta

}

**To make a modification to the IDU layer**: Models/Processes should not modify the map directly – instead, they should add a DELTA to the map indicating the change to be made. Envision then modifies the map with all DELTA’s added when the model/process returns control to Envision when exiting the call. The EnvContext contains a pointer to a function for adding DELTA’s. To simplify use, the EnvExtension class provides a member function makes it very easy to add DELTA’s.

const MapLayer \*pLayer = pContext->pMapLayer;

for ( MapLayer::Iterator idu=pLayer->Begin(); idu < pLayer->End(); idu++ )

{

float fValue = 10.0f;

int iValue = 20;

**AddDelta( pContext, idu, m\_col0, fValue ); // note: these are polymorphic**

**AddDelta( pContext, idu, m\_col1, ivalue );**

}

**To define input/output variables for models/processes:** Models can optioanlly expose input and output variables to Envision. Exposed input variables can be set differently across different scenarios; exposed output variables are used by Envision to collect data from the model/process during a run. The EnvExtension parent class manages these variables. All that is needed it to indicate the number of each in the constructor, allocate each variable as a class member, and call EnvExtension::DefineVar() for each exposed variable specified in the constructor. Alternatively, variables can be defined dynamically using the EnvExtension::AddVar() and related methods, in which case the number is NOT specified in the constructor – the default constrctor with no arguments is used instead.

myprocess.h

class MyModel : public EnvEvalModel

{

protected:

int m\_input1; // declare input vars

float m\_input2;

int output; // declare output var

public:

MyModel(); // constructor

}

myprocess.cpp

MyModel::MyModel()

: EnvEvalModel()

, m\_input1(10)

, m\_input2(2)

, m\_output( 5 )

{

AddInputVar ( \_T(“Input 1”), m\_input1, \_T(“this is an input”) );

AddInputVar ( \_T(“Input 2”), m\_input2, \_T(“this is another”) );

AddOutputVar( \_T(“Output”), m\_output, \_T(“this is an output”) );

}

# Chapter 7. Autonomous Processes

Autonomous processes in ENVISION are defined as those processes that “run” independent of or in conjunction with the human-decisionmaking processes embodied in actor policy selection and application. They can be used to incorporate a wide range of behaviors into ENVISION; for example, simulating vegetative succession, population growth, or flood events. ENVISION comes with a variety of “standard” autonomous process modules, for representing common situations with no programming required. Set the “Standard Plug-ins” chapter for details.

Autonomous processes conform to a “plug-in” architecture similar to the Evaluative Models described above – see that section for details on writing an autonomous process using the EnvAutoProcess class provided in the Envision SDK. Autonomous processes have a similar interface specification, described in the table below:

|  |  |
| --- | --- |
| **Function( args )** | **Description** |
| APInit( EnvContext\*, CString\* pArgStr ) | This function is called when ENVISION first starts up and loads the autonomous process. It is passed the argument string specified for the autonomous process in the Project file, which the function can interpret as necessary. This function should perform any necessary one-time initialization (optional) |
| APInitRun( EnvContext\*, bool useInitSeed) | This function is called at the beginning of each run. It should perform any necessary initialization to prepare the autonomous process for a new run. The “useInitSeed” argument indicates whether the model should reset any random number generators to their original seed values (provides reproducibility of the random streams). (optional) |
| APRun( EnvContext\* ) | Main entry point. This function is called by ENVISION each year of an analysis run. This function receives a current IDU coverage, and its primary purpose modify the map by adding Delta’s to the map’s delta array (see Delta Array discussion below) (required) |
| APEndRun( EnvContext\*) | Called at the end of a run (optional) |
| APShowProperties( HWND parent, int ID ) | This function is called to signal that the autonomous process should pop up a dialog box or property sheet to allow setup of the autonomous process . The ***parent*** argument is a handle to the parent window for the dialog; the ***ID*** is the autonomous process ID specified in the Project file. (optional) |
| APInputVar( int ID, MODEL\_VAR\*\* )  APOutputVar( int ID, MODEL\_VAR\*\* ) | These functions is called to allow autonomous process to “expose” input variables to ENVISION for use in defining scenarios. The ***ID*** is the autonomous process ID specified in the Project file; the ***modelVar*** argument is a reference to the address of an array of MODE\_VAR structures defined in the autonomous process’s DLL, one MODEL\_VAR for each variable to be exposed. The MODEL\_VAR structure is described below. This function should return the number of variables exposed and the address of the MODEL\_VAR array in the ***modelVar*** argument. (optional) |

Similar to evaluative models, the Envision SDK provide high-level C++ wrappers that significantly simplify the process of developing autonomous process plug-ins. The process is virtually identical to that described in the Evaluative Models chapter – but instead of deriving your class from “EnvEvalModel”, use”EnvAutoProcess” instead.

# Chapter 8. Visualizers

ENVISION supports the concept of “Visualizers” – extension modules that can be “plugged in” the Envision to provide visualization and display services. Like all plug-ins, these are required to be packaged as DLLs and expose conformant functions that are invoked by Envision. Visualizers come in three flavors: 1) **Input Visualizers**, designed to provide inputs into model runs, 2) **Runtime Visualizers**, designed to provide a visualization of outputs during a simulation run, and 3) **PostRun Visualizers**, design to be viewed on the Envision PostRun tab after the completion of a run. A single visualizer can expose functionality that allows it to be any or all of these types. Additionally, a given DLL can contain one or more visualizers, each identified by a unique ID.

The Visualizer interface consists of four functions, defined in EnvContext.h:

BOOL PASCAL VInit (EnvContext\*, LPCTSTR initInfo ); // for initialization

BOOL PASCAL VInitRun(EnvContext\* ); // for init of run

BOOL PASCAL VRun (EnvContext\* ); // for runtime views

BOOL PASCAL VShowProperties(EnvContext\*, HWND ); // displaying properties for the Visualizer

A given visualizer may implement one or more of these interfaces, depending on its type, as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **VInit()** | **VInitRun()** | **VRun()** |
| Input | Required | Optional (but likely useful) | Optional |
| Runtime | Optional | Optional (but likely useful) | Required |
| PostRun | Optional | Optional | Optional |

In all cases, the visualizer is called with an EnvContext structure providing access to Envision internals and current context, which contains a window handle (HWND) stored in the “extra” field of the context structure that corresponds to the window the visualizer “owns”. This handle remains constant throughout the life of the visualizer. It provides the device context on which the visualizer operates.

To include a visualizer in an application, you must either add it through the Envision <Options> dialog (which stores the necessary information in the Windows registry) or include an appropriate entry in the project (.envx) file, as described in Chapter 2.

The Envision SDK provides a class, EnvVisualizer, that simplifies development of visualizers for Envision. Its use is similar to that described above under the “Evaluative Models” section.

# Chapter 9. The Delta Array and Dynamic Maps

Documentation for the Delta Array is currently being developed.

Chapter 10. Standard ENVISION Plug-ins

ENVISION is distributed with a number of “standard” plug-ins that are easily modified to provide site- and application-specific functionality. These are not required for ENVISION, but they do provide significant, commonly used functionality to facilitate application development. These plug-ins and their input requirements are described below.

## Simple State Transition Model (SSTM) (Autonomous Process)

The Simple State Transition Model (SSTM)is a very simple autonomous process plug-in that “ages” a landscape by providing for succession from different land use/land cover types (or any other state-transition variable). Succession is modeled using a transition table that defines transition probabilities and periods until transitions occur to any number of other classes. These transitions are specified in an Xml file, and the name of the file is specified in the Project file as the initialization string for SSTM.

An example SSTM input file (for the NLCD 2001 classification) is given in below:

<?xml version="1.0" encoding="utf-8"?>

<!-- Transition Class defintions for ALPS -->

<transitions>

<!-- Decidous Forest-->

<trans start="41" end="43" period="25" prob="25" /> <!-- to Evergreen Forest-->

<!-- Evergreen Forest-->

<trans start="42" end="43" period="25" prob="10" /> <!-- to Mixed Forest-->

<!-- Mixed Forest-->

<trans start="43" end="41" period="25" prob="25" /> <!-- to Decidious Forest-->

<trans start="43" end="42" period="15" prob="10" /> <!-- to Evergreen Forest-->

<!-- Shrub Scrub -->

<trans start="52" end="41" period="25" prob="10" /> <!-- to Deciduous Forest-->

<!-- Grassland -->

<trans start="71" end="52" period="15" prob="40" /> <!-- to Shrub/Scrub -->

<!-- Palustrine Shrub/Scrub-->

<trans start="92" end="52" period="10" prob="5" /> <!-- to Deciduous Forest-->

<trans start="92" end="71" period="10" prob="5" /> <!-- to Grassland -->

<trans start="92" end="91" period="10" prob="10" /> <!-- to Palustrine

Forested -->

</transitions>

The format of this file consists of a <**transitions**> tag containing individual transition descriptors. Individual transition descriptors are defined by a <**trans**> tag with the following attributes:

|  |  |
| --- | --- |
| **<trans> Attribute** | **Description** |
| Start | The Lulc C class code for which this transition is defined. This is defined in the lulcTree input file. (required) |
| End | The Lulc C class code which this transition will result in. This is defined in the lulcTree input file. (required) |
| Period | The average duration for the ‘start’ class before a transition occurs (years) (required) |
| prob | The probability that a transition will occur over the period (0-100) (required) |

To include SSTM in an application, include the <**autonomous\_process**> entry shown below in the <**autonomous\_processes>** section of the Project file. Note that the SSTM Xml input file is given as the last argument. Note that specified files must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<autonomous\_processes>

<autonomous\_process

name ='State Transition Model'

path ='sstm.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='sstm\_nlcd.xml'

/>

</autonomous\_processes>

## DynamicVeg (Autonomous Process)

**DynamicVeg** is an Envision Extension that provides a very flexible framework for deterministic and probabilistic vegetation state class transition modeling. Users create State and Transition models that are considered at each Envision time step, and for each IDU in the study area. If a vegetation state class has been selected for a transition to a new state class, that transition will be updated in the VEGCLASS attribute in the IDU layer.

There are two different types of state class transitions models that DyanmicVeg handles, deterministic and probabilistic. A deterministic transition is considers at each time step by retrieving the current value of AGECLASS in IDU layer, comparing it to an age class threshold in the deterministic transition model, and updating the VEGCLASS attribute in the IDU layer if the current age class is equal to the value in the deterministic model. Probabilistic transitions are also considered at each time step for each IDU. For each IDU, DynamicVeg will query the user defined probabilistic transition models and return a suite of potential transitions state classes with their corresponding probabilities. A Monte Carlo algorithm is then employed and based on a pseudo random number a probabilistic transition may, or may not be selected. Again, any state class transitions will be updated in the VEGCLASS attribute in the IDU layer.

DynamicVeg is dependent on 6 different attributes that must exist in the IDU layer, they are VEGCLASS, DISTURB, REGION, PVT, AGECLASS, and TSD. DynamicVeg will also utilize 5 different attributes if present in the IDU layer. They are FUTSI, LAI,REGEN, VARIANT, and FORESTC (table 1).

Table 1. DynamicVeg and the IDU layer

|  |  |  |  |
| --- | --- | --- | --- |
| IDU layer attribute | TYPE | DESCRIPTION | REQUIRED IN IDU LAYER |
| VEGCLASS1,3,4 | INTEGER | A 7 digit number representing individual vegetation state classes. VEGCLASS must be greater than or equal to 2000000. | YES |
| DISTURB2,3 | INTEGER | Enum DISTURB Envconstants.h. Also used to filter deterministic and probabilistic models. DISTURB less than or equal to zero is interpreted as no disturbance. | YES |
| REGION3,4 | INTEGER | An IDU layer may have more than one “modeling region” for similar VEGCLASSes. This attribute is used to filter the deterministic and probabilistic models | YES |
| PVT3,4 | INTEGER | Potential Vegetation Type. An IDU layer may have more than one PVT. This attribute is used to filter the deterministic and probabilistic models | YES |
| AGECLASS3 | INTEGER | The number of time steps a VEGCLASS is in it’s current state. | YES |
| TSD | INTEGER | Time since any type of disturbance. See DISTURB | YES |
| REGEN3 | INTEGER | A VEGCLASS may be in a regeneration state. This attribute is used to filter the probabilistic models | OPTIONAL |
| FUTSI3 | INTEGER | Future Site Index. If present this attribute is used to filter the probabilistic models | OPTIONAL |
| LAI4 | FLOAT | Leaf Area Index. If present this attribute is updated in the IDU layer if VEGCLASS changes | OPTIONAL |
| FORESTC4 | FLOAT | Forest Carbon. If present this attribute is updated in the IDU layer if VEGCLASS changes | OPTIONAL |
| VARIANT3 | INTEGER | An index to represent variants in fuel models for fire modeling | OPTIONAL |

1 – see defining a VEGCLASS attribute

2 – DynamicVeg handles all DISTURB VEGCLASS transitions, see disturbance transitions

3 - A corresponding column must exist in the user defined Probabilistic models. See creating a Probabilistic model.

4 – A corresponding column must exist in the user defined Deterministic models. See creating a Deterministic AGECLASS model.

***Defining a VEGCLASS attribute*** - VEGCLASS is a successional vegetation classification within a Potential Vegetation Type (PVT) that are defined by combinations of Cover Types and Structural Stages. VEGCLASS is represented by a seven digit number defined by the user. The first three digits represent the Cover Type, or a vegetation classification depicting a genus, species, group of species, or life form of tree, shrub, grass, or sedge. The second three digits represent the Structural Stage, or a stage of development of a vegetation community that is classified on the dominant processes of growth, development, competition, and mortality. The last digit is a place holder that may be utilized in later versions of DynamicVeg, and is currently set equal to zero. Table 2 gives an example of how to create a VEGCLASS seven digit code.

Table 2. Example of how to create a VEGCLASS seven digit code (VEGCLASS code must be greater than or equal to 2000000)

**Cover Type**

GrassShrub

Grass / Shrub plus NLCD wetlands and veg

GS

meadow/shrub

200

SubAlp parkland

Subalpine parkland

PK

meadow/shrub

205

Alder

Red alder

Al

broadleaf

250

Asp\_Willow

Trembling aspen / Willow

AW

broadleaf

255

Oak

Oregon white oak

Oa

Oak

300

OakPine

Oregon white oak / Ponderosa pine

OP

Oak

305

DougFir

Douglas-fir

DF

mixed conifer

400

DFirMix

Douglas-fir mix

DFmx

mixed conifer

405

**Structural Stage**

1st digit

2nd digit

3rd digit

Barren

0

None

0

None

0

Development

1

Low (open, 10-40%)

1

Single

1

Meadow

2

Medium (40-60%)

2

Multi

2

Shrubs

3

High (closed, >60%)

3

Seedling/sapling

4

Post-disturbance

4

Pole (5-10" dbh)

5

Small tree (10-15" dbh)

6

Medium tree (15-20" dbh)

7

Large tree (20-30" dbh)

8

Giant tree (>30" dbh)

9

**VEGCLASS (last digit zero)**

Description

4005210

Douglas Fir:Pole:medium canopy closure:single layer

***Creating a Probabilistic Model*** - For each Envision time step, and for each IDU, DynamicVeg will retrieve current values of the following attributes in the IDU layer, REGION, PVT, REGEN, TSD, DISTURB, and VEGCLASS. With these six values DynamicVeg will create a “key” that is used to retrieve the proper State Transition Model(s) (STM) from a comma delimited text file referred to as the “probability-lookup-table”. Each record, or line in this file is a STM. The first line of the file must contain a “column name” that is exactly like those found in Table 3.

The probability lookup-table that is used by DyanamicVeg is modeled after the probability table from a state and transition modeling effort put forth by Integrated Landscape Assessment Project (<http://oregonstate.edu/inr/sites/default/files/project_ilap/ILAP_final_report_sep9_2013.pdf>). The ILAP project has created STMs for many VEGCLASSes in the Western United States.

Table 3. DynamicVeg and the Probability lookup table

|  |  |  |  |
| --- | --- | --- | --- |
| Lookup table attribute or column name | TYPE | DESCRIPTION | REQUIRED IN LOOKUP TABLE |
| VEGCLASSfrom1,3,4 | INTEGER | A 7 digit number representing the starting vegetation state classes. VEGCLASS must be greater than or equal to 2000000. | YES |
| VEGCLASSto1,3,4 | INTEGER | A 7 digit number representing the vegetation state classes transition to. VEGCLASS must be greater than or equal to 2000000. | YES |
| DISTURB2,3 | INTEGER | Enum DISTURB Envconstants.h. Also used to filter deterministic and probabilistic models. DISTURB equal to 0 is interpreted as successional STM. Greater than zero is interpreted as a disturbance | YES |
| REGION3,4 | INTEGER | An IDU layer may have more than one “modeling region” for similar VEGCLASSes. This attribute is used to filter the deterministic and probabilistic models | YES |
| PVT3,4 | INTEGER | Potential Vegetation Type. This attribute is used to filter the deterministic and probabilistic models | YES |
| PVTto | INTEGER | For each VEGCLASS transition, a transition of PVT is possible | YES |
| MINAGE | INTEGER | If a successional probabilistic VEGCLASS transition is to occur, AGECLASS must be greater than or equal to MINAGE | YES |
| MAXAGE | INTEGER | If a successional probabilistic VEGCLASS transition is to occur, AGECLASS (IDU layer) must be greater than or equal to MAXAGE | YES |
| TSD | INTEGER | If a successional probabilistic VEGCLASS transition is to occur, Time Since Disturbance (TSD IDU layer) must be greater than or equal to TSD | YES |
| P | FLOAT | The probability a VEGCLASS will transition to a new VEGCLASS | YES |
| REGEN3 | INTEGER | A VEGCLASS may be in a regeneration state. This attribute is used to filter the probabilistic models | YES |
| KEEPRELAGE | INTEGER | If KEEPRELAGE is equal to 1 then RELATIVEAGE is subtracted from AGECLASS (IDU Layer). If zero, then AGECLASS (IDU layer) is unaffected. | YES |
| RELATIVEAGE | INTEGER | If KEEPRELAGE is 1 then RELATIVEAGE is subtracted from AGECLASS (IDU layer) | YES |
| FUTSI3 | INTEGER | Future Site Index. If present this attribute is used to filter the probabilistic models | OPTIONAL |
| VARIANT4 | INTEGER | If a VEGCLASS (IDU layer) transitions occurs, this value will update VARIANT (idu layer). An index to represent variants in fuel models for fire modeling | OPTIONAL |

1 – see defining a VEGCLASS attribute

2 – DynamicVeg handles all DISTURB VEGCLASS transitions, see disturbance transitions

3 - A corresponding attribute must exist in the IDU layer

4 – A corresponding attribute must exist in the IDU layer

***Optional probability multiplier file*** - Each time a probabilistic STM is considered for a given IDU at a given Envision time step, the user has the option to multiply the existing probability of the STM by a multiplier. The probability multiplier file (comma delimited) is specified in the DynamicVeg input file (see ***DynamicVeg input file***). The attributes, or column names in the file, must match the names that are listed in table 4

Table 4. Probability Multiplier file

|  |  |  |  |
| --- | --- | --- | --- |
| Lookup table attribute or column name | TYPE | DESCRIPTION | REQUIRED IN LOOKUP TABLE |
| VEGCLASS1,3,4 | INTEGER | A 7 digit number representing the starting vegetation state classes. VEGCLASS must be greater than or equal to 2000000. | YES |
| DISTURB2,3 | INTEGER | Enum DISTURB Envconstants.h. Also used to filter deterministic and probabilistic models. DISTURB equal to 0 is interpreted as successional STM. Greater than zero is interpreted as a disturbance | YES |
| REGION3,4 | INTEGER | An IDU layer may have more than one “modeling region” for similar VEGCLASSes. This attribute is used to filter the deterministic and probabilistic models | YES |
| PVT3,4 | INTEGER | Potential Vegetation Type. This attribute is used to filter the deterministic and probabilistic models | YES |
| PROBMULTIPLIER | FLOAT | If a successional probabilistic VEGCLASS transition is to occur, AGECLASS must be greater than or equal to MINAGE | YES |
| FUTSI3 | INTEGER | Future Site Index. If present this attribute is used to filter the probabilistic models | OPTIONAL |
| TIME | INTEGER | Envision’s zero based time step | yes |

1 – see defining a VEGCLASS attribute

2 – DynamicVeg handles all DISTURB VEGCLASS transitions, see disturbance transitions

3 - A corresponding attribute must exist in the IDU layer

4 – A corresponding attribute must exist in the IDU layer

***Creating a deterministic AGECLASS Model*** - For each Envision time step, and for each IDU, DynamicVeg will retrieve current values of the following attributes in the IDU layer; REGION, PVT, and, VEGCLASS. With these three values DynamicVeg will create a “key” that is used to retrieve the proper age-class deterministic State Transition Model (STM) from a comma delimited text file referred to as the “deterministic-lookup-table”. Each record, or line in this file is an age class deterministic STM. The first line of the file must contain a “column name” that is exactly like those found in Table 5.

Once retrieved from the deterministic lookup table, each STM will be associated with a start age, and end age. For the given IDU, the AGECLASS of that IDU is compared to the end age class of the STM. If the AGECLASS of the IDU is equal to the end age in the STM, then a VEGCLASS transition will occur.

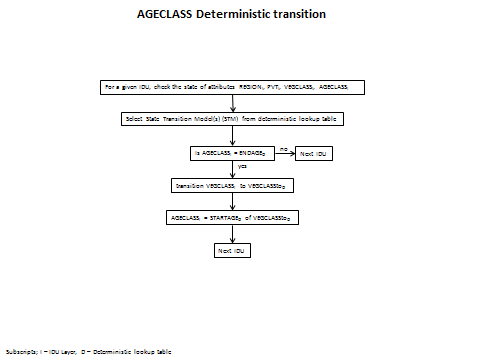
Table 5. DynamicVeg and the deterministic AGECLASS lookup table

|  |  |  |  |
| --- | --- | --- | --- |
| Lookup table attribute or column name | TYPE | DESCRIPTION | REQUIRED IN LOOKUP TABLE |
| VEGCLASSfrom1,2,3 | INTEGER | A 7 digit number representing the starting vegetation state classes. VEGCLASS must be greater than or equal to 2000000. | YES |
| VEGCLASSto1,2 | INTEGER | A 7 digit number representing the vegetation state classes transition to. VEGCLASS must be greater than or equal to 2000000. | YES |
| REGION2 | INTEGER | An IDU layer may have more than one “modeling region” for similar VEGCLASSes. This attribute is used to filter the deterministic and probabilistic models | YES |
| PVT2 | INTEGER | Potential Vegetation Type. This attribute is used to filter the deterministic and probabilistic models | YES |
| STARTAGE | INTEGER | The start age of the AGECLASS | YES |
| ENDAGE | INTEGER | The end age of the AGECLASS | YES |

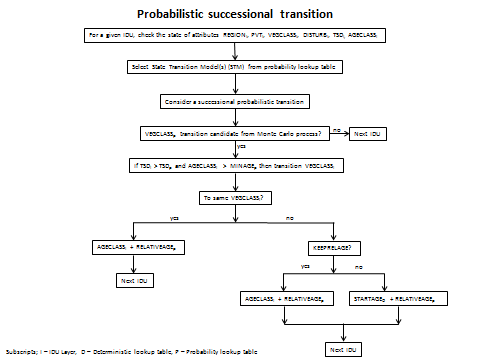
1 – see defining a VEGCLASS attribute

2 - A corresponding attribute must exist in the IDU layer

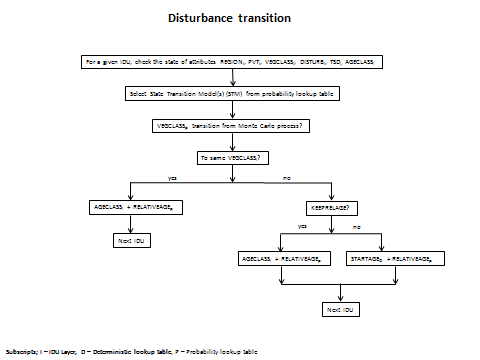
***Deterministic AGECLASS transitions*** - At each Envision time step DynamicVeg checks that status of the IDU layer (see ***Creating a Determinsitic Model)***. Among the attributes it checks is the value of AGECLASS. If the AGECLASS of the IDU layer is equal to the attribute ENDAGE in the deterministic lookup table then VEGCLASS is changed in the IDU layer (figure 1)



***Probabilistic successional transitions*** - At each Envision time step DynamicVeg checks that status of the IDU layer (see ***Creating a Probabilistic Model)***. Among the attributes it checks is the value of DISTURB. If the value of DISTURB is less than or equal to zero (less than zero indicates past disturbance of that value) a successional probabilistic transition is considered (DISTURB=0 in probability lookup table). More than one possible VEGCLASS trajectory may be possible for any give IDU, including no selection of all the possible VEGCLASS transitions (figure 2).

******

***Disturbance transitions*** - DynamicVeg is an autonomous process that does not create a disturbance (wildfire, prescribed burning, harvests, thinnings, etc…) on the IDU landscape, but handles the VEGCLASS transitions as a result of a disturbance created by either an Envision policy, or another autonomous process. Disturbance transitions are State and Transitions Model(s) (STM) that are cataloged in the probability lookup table (see ***Creating a Probabilistic Model)***. For each disturbance type there may be one or more possible VEGCLASS transition trajectory. For all STMs the probabilities must be greater than 0, and will be normalized between 0 and 1 before being compared to a pseudo random between 0 and 1 in the Monte Carlo process. Normalization of the probability will insure that a disturbance transition will take place (figure 3).



***DynamicVeg input file*** - Users can take advantage of options available in DynamicVeg by setting values in an Extensible Markup Language file (.xml file). The input file is specified in the Envision start-up document prior to run time (see ***Running DynamicVeg in Envision)*** . An example file with explanations of settings is listed in Figure 4. The user is encouraged to cut and paste figure 4 into a file used as input into DynamicVeg.

Figure 4. Example DyanmicVeg input file

<?xml version="1.0" encoding="utf-8"?>

<!--

Set "dynamic\_update" to "1" for dynamic site index and pvt over time.

Set "dynamic\_update" to "0" for static site index and pvt over time.

dynamic (1) will use every year of data for the selected site index and pvt input files, and increase in step

with current Envision time step.

Static (0) will use the first year of data for the selected site index and pvt input files,

for all time steps.

if filenames are not specified ("") then site index and PVT will not vary over time.

Currently, this option is under developement in DynamicVeg

-->

<pvt dynamic\_update="1">

<mc1\_outputs>

<mc1\_output id="0" name="hadleyA2"

site\_index\_file=""

pvt\_file=""

/>

<mc1\_output id="1" name="mirocA2"

site\_index\_file=""

pvt\_file=""

/>

<mc1\_output id="2" name="csiroA2"

site\_index\_file=""

pvt\_file=""

/>

</mc1\_outputs>

<!--

transition deltas -is a binary switch that when set equal to 1 will create an attribute

in the IDU layer named "VEGTRANTYPE" during Init. During run time a delta will be

sent to the delta array everytime DynamicVeg implements a VEGCLASS transition. If set

equal to 0, nothing will happen

Values for the IDU Layer attribute VEGTRANTYPE:

0 - is the default set before a run

1 - Successional AGECLASS deterministic transition

2 - Successional probabilistic transition

3 - Disturbance transition

-->

<transition\_deltas id="1"/>

<!--if either the AGECLASS or TSD attributes in the IDU layer need to be

initialized before first Envision time step, then set equal to 1. If

set equal to 0, then it is assumed that the user has initialized these

attributes with another method.

if 1 is set for age\_class\_initializer, then AGECLASS will be intialized in the IDU layer

with a uniformally distributed random AGECLASS value between the STARTAGE and

ENDAGE in the Deterministic Lookup file.

if 1 is set for tsd\_initializer, then TSD will be intialized in the IDU layer

by the following equation. Values for this equation are found in the Probability

lookup table. Random TSD is calcualted based on a uniform distribution between

STARTAGE and ENDAGE (from deterministic lookup table) minus STARTAGE.

TSD = (random AGECLASS) - STARTAGE as found in determinsitic lookup table-->

<initializers>

<initializer

age\_class\_initializer="1"

tsd\_initializer="1"

/>

</initializers>

<!--DynamicVeg requires two lookup tables, a probabilistic state transition model file, and an AGECLASS

deterministic file. The probMulitiplier file is optional. If not used empty quotes should be used. -->

<vegtransfiles>

<vegtransfile

probability\_filename="\envision\StudyAreas\CentralOregon\CO\_probability\_transition\_lookup.csv"

deterministic\_filename="\envision\StudyAreas\CentralOregon\CO\_deterministic\_transition\_lookup.csv"

probMultiplier\_filename=""

/>

</vegtransfiles>

<!--Currently DynamicVeg does not generate any default charts or graphs, however if a user wanted to add

and output variable use this form.-->

<outputs>

<output name="Douglas-fir" query="VEGCLASS &gt; 4001990 and VEGCLASS &lt; 4008330" />

</outputs>

</pvt>

***Running DynamicVeg in Envision*** - The **DynamicVeg** process and individual **DynamicVeg** elements are defined in an Xml input file that is specified in the Project (.ENVX) file (see **Project (.envx) File**). In some cases the timing and sequence of autonomous processes may require more than one instance of DynamicVeg to be called during each Envision time step, and at certain times in the Envision process. The order in which the DynamicVeg processes are listed in the .envx file, and the use of the elements “id” and “timing”, control how and when DynamicVeg is called. Setting the “id” element equal to zero will invoke DynamicVeg to consider successional probabilistic, or an age class deterministic transitions. Setting “id” equal to 1 will invoke DynamicVeg to consider any possible disturbances created by other autonomous processes on the landscape. Setting “id” equal to 2 will increment the AGECLASS attribute in the IDU layer by one if no VEGCLASS transitions for a given IDU have been recorded for the current Envision time step.

In the example listed in Figure 5 the syntax that might be included in an .envx file might have multiple instances of DynamicVeg being called to handle different types of VEGCLASS transitions on the landscape. The first call handles any successional probabilistic, or age class deterministic transitions. Notice the element “id” is set equal to zero, and the element “timing” is set equal to zero (called before evaluative models, or actor decisions/policies). The second call to DynamicVeg handles a disturbance (“id” set equal to 1) in response to the “Example wildfire disturbance creator” , both with “timing” element s set equal to 0. The third call to DynamicVeg’s disturbance handler is called with a “timing” element set equal to 1. “Timing” elements set equal to 1 are called after evaluative models and actor decisions/policies (e.g. timber harvest). Lastly, DyanmicVeg is called to increment AGECLASS on the IDU layer for each IDU if no VEGCLASS transitions have taken place. Thus, the final call to DynamicVeg at the end of the autonomous processes has the element “id” set equal to 2, and the element “timing” set equal to 1.

<autonomous\_process

name ='SuccessionTransitions'

path ='DynamicVeg.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='DynamicVeg\_input\_file.xml'

dependencies =''

initRunOnStartup ='0'

/>

<autonomous\_process

name ='Example wildfire disturbance creator'

path ='Example.dll'

id ='6'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='Example.xml'

dependencies =''

initRunOnStartup ='0'

/>

<autonomous\_process

name ='DisturbanceTransitions'

path ='DynamicVeg.dll'

id ='1'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='DynamicVeg\_input\_file.xml'

dependencies =''

initRunOnStartup ='0'

/>

<autonomous\_process

name ='DisturbanceTransitions'

path ='DynamicVeg.dll'

id ='1'

use ='1'

timing ='1'

freq ='1'

sandbox ='0'

fieldName =''

initInfo = 'DynamicVeg\_input\_file.xml'

dependencies =''

initRunOnStartup ='0'

/>

<autonomous\_process

name ='TimeInAgeClass'

path ='DynamicVeg.dll'

id ='2'

use ='1'

timing ='1'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='DynamicVeg\_input\_file.xml'

dependencies =''

initRunOnStartup ='0'

/>

</autonomous\_processes>

## Flow (Autonomous Process)

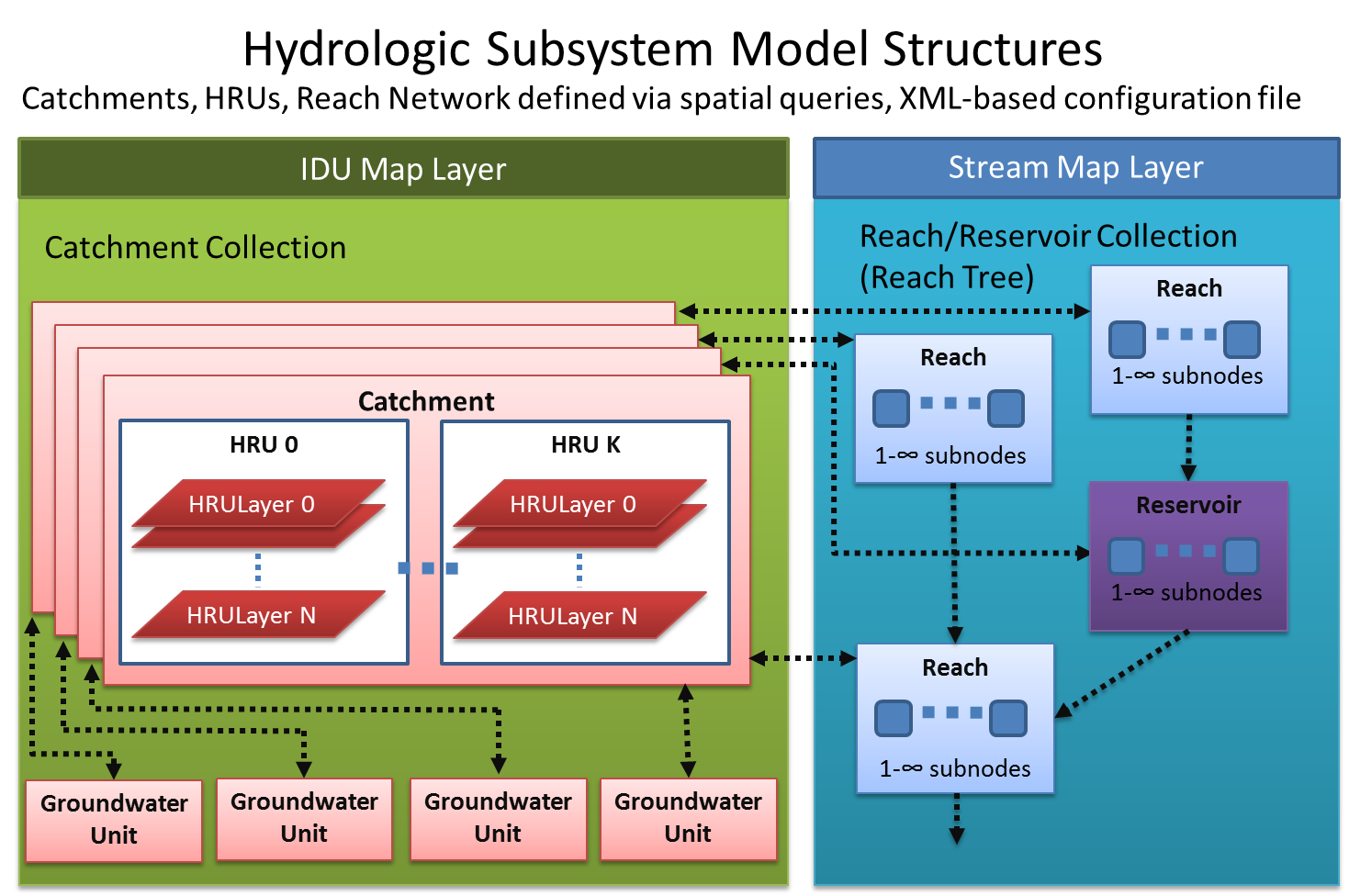
**Flow** is a Envision Extension that provides a very flexible framework for hydrologic modeling. Similar to Envision, is supports the idea of “plug-ins”, DLL’s that extend functionality. **Flow** provides a number of elements of hydrological process representation, including geometric representation of terrestrial and aquatic (riverine) datasets used in the hydrologic model, topology, simulation control, data management, and default implementations of important hydrologic processes. While providing for basic hydrologic connectivity (described in detail below), the framework supports the concept of externally defined fluxes for representing sources or sinks at various points in the hydrologic system, i.e. “straws” that the can add, remove, or transfer water at specific locations. In addition, if the various internal implementations of hydrologic processes are insufficient for a given application, the can be globally or locally overridden. Input is an XML file specifying definitions of catchments, hydrologic response units (HRU), and river/stream features, externally-define fluxes and process overrides, and other information needed to define a specific hydrological model. These various framework elements are defined in detail below.

**Spatial Representation**. **Flow** defines (in very flexible ways) a spatial geometry and set of geometric objects that represent spatial elements of hydrologic system (see figure below). Terrestrial elements are organized as Hydrologic Response Units (HRUs) that are represented by collections of polygons in a source layer (typically the IDU layer) with common attributes. HRUs are a fundamental unit of terrestrial hydrologic representation in **Flow**. They can consist of one or more layers (*HRULayer* objects) that maintain water content information and any number of additional state variables (e.g. heat, nitrogen level.) HRU Layers come in three flavors: snow, vegetation, and soil; any number (including zero) of any of these layer types can be defined, but they are always sequenced top to bottom with snow on top, vegetation in the middle and soil layers on the bottom. HRUs are collected into catchments (*Catchment* objects) with a single drainage point. These are assembled by **Flow** from a GIS layer (*MapLayer* object) managed by Envision, typically the IDU layer, that has relevant spatial data used to define or model the hydrologic system. Similarly, the stream/river network is represented by a linear feature network coverage (*MapLayer* object), also managed by Envision, that represents the river system network layer that resides “on top” of the IDU coverage. Reaches are subdivided into “subnodes” that can be spaced as needed for a particular model construct. *Catchments*, *HRUs* and *Reaches* are defined by “spatial aggregates”, collections of polygons that have similar attributes. *HRULayers* always nest within an *HRU*, HRU’s should always nest in a catchment. The spatial domain of these aggregates can be limited, meaning a subset of base layer terrestrial polygons or stream layer lines can by selected into the aggregates; any excluded polygons/lines are simply ignored by the model.

[NEED TO ADD INFO ON RESERVOIRS, GROUNDWATER]

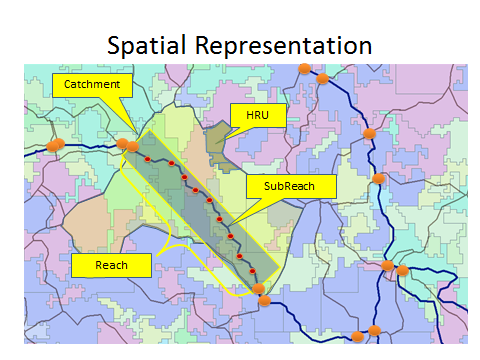
State variable information is maintained at the *HRULayer* level and at the Reach subnode level. By default, a single state variable related to water content is maintained by these containers. Additional state variables, representing transported constituents, can be easily added by specifying them in the input XML file.

Topologically, **Flow** defines and manages hydrological connectivity between *Catchments* and *Reaches* – catchments “know” what reach they are connected to, and vice versa. **Flow** assumes that terrestrial/riverine connectivity is between catchments and reaches only; i.e. all lateral inflow/outflow from a catchment’s *HRULayers* is collected at the catchment level and transferred to the reach corresponding to a catchment. Connectivity is defined through a join column that must be present in the catchment coverage, with a related join column in the reach coverage; catchments are joined to reaches that share common values in their join fields; this is assumed to be a 1:1 relationship; one catchment can be associated with no more than one reach, and one reach can be associated with no more than one catchment.



**Internal Hydrologic Processes**. **Flow** provides several internal representations of important hydrologic processes, including vertical transport among HRULayers, transfer of water/constituents between catchments and reaches, and instream routing. Developers can specify which internal method to use, or override them completely and provide an externally defined method. In this latter case, a DLL and function entry point must be specified (a “global handler”), and appropriate algorithms provided in that entry point. This external function has full access to the internal representations provided by flow. Details are providing in the “Writing a Global Handler” section below.

**Defining External Fluxes**. Central to **Flow** is the idea that external fluxes (“straws”) can be added to hydrological models easily and flexibly. Fluxes can represent wells, surface water extraction/additions, precipitation, processes like ET, or any other hydrology-related flux. Fluxes can be defined via an external plugin, an external data source, or, in simple cases, by specification of evaluable expression in the **Flow** input file. Fluxes can be “sources” or “sinks”, and can be one-way or two-way straws. The straw “ends” can be inserted into a *HRULayer,* a *Reservoir*, a *Groundwater unit* or a *Reach*.



The **Flow** process and individual **Flow** elements are defined in an Xml input file that is specified in the Project (.ENVX) file. An example input files is given below. .

To include a Flow-based mode in an application, include the **<autonomous\_process>** element shown below in the **<autonomous\_process>** section of the Project file. Note that the **Flow** Xml input file is given as the initInfo. This Flow entry corresponds to the Flow file given above. Note that the specified file must be either in the ENVISION executable directory, or specified as a fully-qualified path.

<autonomous\_processes>

<autonomous\_process

name ='My Hydro Model'

path ='flow.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='flow\_input.xml'

/>

</autonomous\_processes>

## WaterMaster (a Plug-In for the Flow Autonomous Process )

The WaterMaster plugin for the Flow autonomous process is a framework that utilizes spatially explicit place-of-use, and point-of-diversion input data to simulate water right diversions as governed by a prior appropriated water-right system. Currently, WaterMaster simulates diversions from Flow’s stream layer, and ground water sources for irrigation, municipal, and in-stream uses. Other water-right use, and diversion sources can be easily added to the WaterMaster framework as needed.

## Modeler (Evaluative Model and/or Autonomous Process)

The Modeler extension allows for definition of autonomous processes and evaluative models at a very high level. It has support for a variable of variable and expression types that can be incorporated into models. Modeler allows application developers to define evaluative models and autonomous processes using a simple map-based modeling language, expressed in an Xml format file. Multiple models can be defined in a single Xml file. Variables in the model(s) correspond to fields in the IDU database or can be defined using an expression or a lookup table. A variety of mathematical and logical operators, described below, are available to the modeler.

Models specified using the Modeler extension are calculated as follows. At each time step the model is evaluated, the model expression is evaluated for each IDU on the map. If a column is specified for the model, the results of the expression evaluation are stored in a column in the IDU database.

For evaluative models, the overall model metric is then calculated (representing a landscape value of the model expression), based on the values of the **evalType** and **method** specifications (see table for definitions), and returned to ENVISION as an evaluative model scores, scaled to the range [ -3 to +3 ]. Typically this is computed as an area-weighted average score, scaled between a lower bound and upper bound specified in the model.

Modeler also supports the concept of “Network” models and processes. These are similar to regular models and process, except that that run on a linear network coverage superimposed on top of the IDU layer. There are particularly useful for representing riverine processes that interact with the IDU representation of the landscape.

Basic Concepts

Models/processes are specified as a set of constants, variables and expressions that are computed at runtime by the Modeler extension. Models/processes can be specified in a number of ways, described below. Modeler input files are XML files with the outermost containing element being the <modeler> tag. Within this tag, any number of constants, variables, evaluative models and autonomous processes can be defined using XML syntax described below.

Variable/Constant Scoping: Variables and constants can be defined within the <modeler> tag, giving them global scope (available to all models/processes) or within the scope of given model/process tag, making them “visible” only to that model or process.

Constants: Constants are values that do not change during a simulation. They are

<?xml version="1.0" encoding="utf-8"?>

<!--

NOTE: ORDER OF MODELS IS IMPORTANT AND MUST MATCH ID's in the correspondign INI File!

evalType : specifies how this evaluation is made. One of the following (required)

HIGH - want to maximize the area within the range defined by the bounds

LOW - want to minimize the area within the range defined by the bounds

method : specifies how the calcualtion is made. One of the following (required)

TOTALAREA - compute total area that satifies the query

AREAWTAVG - compute an area-weighted average value

lowerBound: the lower bound for this attribute, expressed in total area or percent area, depending on the valueof the "area" attribute (required)

upperBound: the upper bound for this attribute, expressed in total area or percent area, depending on the valueof the "area" attribute (required)

-->

<modeler>

<lookup name="NitrogenCoeff" col="lulc\_c">

<map attrValue="1" value="0.13" />

<map attrValue ="2" value="0.16" />

<map attrValue ="3" value="0.19" />

</lookup>

<lookup name="EconomicCoeff" col="lulc\_c">

<map attrValue ="1" value="0.44" />

<map attrValue ="2" value="0.66" />

<map attrValue ="3" value="0.84" />

</lookup>

<lookup name="Impervious" col='Area'>

<histogram category='StateClass' value='336' bins='20' obs='1337'>

<bin left='8.864e-008' right='0.000128685' count='1253'/>

<bin left='0.000128685' right='0.000257282' count='38'/>

<bin left='0.000257282' right='0.000385879' count='18'/>

</histogram>

<histogram category='StateClass' value='337' bins='20' obs='1347'>

<bin left='9.2e-010' right='0.000351896' count='1333'/>

<bin left='0.000351896' right='0.00070379' count='6'/>

<bin left='0.00070379' right='0.00105568' count='4'/>

<bin left='0.00105568' right='0.00140758' count='3'/>

</histogram>

</lookup>

<model name="Nitrogen" id="0" col="Nitrogen" value="NitrogenCoeff" evalType="LOW"

lowerBound="0" upperBound="50" method="PERCENTAREA"/>

<model name="Economic" id="1" col="Economic" value="EconomicCoeff" evalType="HIGH"

lowerBound="0" upperBound="50" method="PERCENTAREA"/>

<model name="ImpervP" id="2" col="Imperv" value="Imperv" evalType="HIGH"

lowerBound="0" upperBound="50" method="PERCENTAREA"/>

</models>

The format of this file consists of a **<models>** tag containing both lookup table definitions (if used) and model specifications. Lookup table variables are defined by a **<lookup>** tag with the following attributes:

|  |  |
| --- | --- |
| **<lookup> Attribute** | **Description** |
| name | The name of the Lookup variable. Blanks (spaces) in the name are not allowed. (required) |
| col | The IDU database col index that this variable references. The lookup table will map values from this column into corresponding values in the lookup table. (required) |
| **<map**  **attrValue=”column**  **attribute *value*”**  **value=”*returned***  ***variable value*”**  **>** tags | Map tags provide the lookup table mapping for the variable. “attrValue” values are the source values (coming from the IDU database for the column referenced by this lookup variable), and “value” values are the value returned for the variable for that IDU. (required) |

Individual Evaluative Models are specified with a **<model>** tag with the following attributes:

|  |  |
| --- | --- |
| **<model> Attribute** | **Description** |
| name | The name of the Evaluative Model. This should match the name of this model specified in the corresponding Project file. (required) |
| id | ID of this model, as defined by the developer. It should be unique, and should match the ID value giving in the corresponding Project file. (required) |
| col | The IDU database column name that is populated with the evaluation result. If the evaluation result spatial information does not need to be retained, this attribute can be omitted. (optional) |
| value | A mathematical expression describing this model. Many operators and functions are available to the model developer, including boolean, trigonometric, and numerical approximation functions. Additional plug-ins can be loaded to extend the parser's language. See Appendix 4 for details on Envision Map Expression language(required)  Lookup variables can be referenced in the model expression. IDU fields can be referenced by their field name.  An example expression is: F0 + F1 \* sin( F2) / sqrt( 1.5 \*L0 ) where F0, F1, andF2 are IDU field names, and L0 is a lookup variable name. |
| evalType | Specifies how this evaluation is made. One of the following (required):  HIGH - want to maximize the area within the range defined by the bounds  LOW - want to minimize the area within the range defined by the bounds |
| method | Species how the evaluative model scoring calculation is made. One of the following (required):  SUM – computes the cumulative sum of scores across IDUs  AVG – computes the average score across IDUs  AREAWTAVG - compute an area-weighted average score |
| lowerBound | The lower bound for this attribute (required) |
| upperBound | The upper bound for this attribute (required) |

To include a Modeler model in an application, include the <model> entry shown below in the <**models**> section of the Project file. Note that the Model Xml input file is given as the initInfo argument. This Project entry corresponds to the program file given above. Note that specified files must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<models>

<model

name ='Impervious Surfaces'

path ='formulaEvaluator.dll'

id ='0'

use ='1'

freq ='1'

decisionUse ='0'

showInResults='1'

fieldName ='EM\_IMPERV'

initInfo ='Bainbridge\_models.xml'

gain ='1.000000'

offset ='0.000000'

/>

</models>

## Program Evaluator (Evaluative Model)

A “Program” is a target or set of targets representing desired landscape condition. For instance, a community may want to preserve a certain percentage of the landscape in agricultural uses, or ensure some level of available buildable lands, or some other similar landscape statistic. ENVISION’s **Program** plug-in provides an easily used capability for defining Program Elements. These are used by ENVISION as Evaluative Models, reporting back to ENVISION the abundance or scarcity to the element in the landscape.

Program Elements are defined in terms of 1) a query that defines the IDU qualities that are included in the program element, 2) a minimum and maximum target that defines when the Program Element is considered abundant or scarce. These targets can be represented as either 1) area of the coverage satisfying the Program Element query, or 2) as the percentage of the study area satisfying the Program Element query. Further, the Program Element can be defined in terms of an absolute level or as a trend (trajectory). Any IDU field or fields can be used to define a program element.

The Program and individual Program Elements are defined in an Xml input file that is specified in the Project file. An example input files is given below. This example specifies two Program Elements, one for Ag Conservation Lands, calculated as a percentage of the landscape where 20 percent is consider scarce and 40 percent is considered abundant; one for Developed lands (LULC\_A=20), calculated as a trend (rate of change) on a total (absolute) area basis, where no change is considered “scarce” growth, and a 10 percent/year growth rate is considered “abundant” growth.

<?xml version="1.0" encoding="utf-8"?>

<program>

<progElement name="Ag Cons Lands" id="0" query="LULC\_A=80 AND CONSERV=1"

weight="0.5" evalType="HIGH" lowerBound="20" upperBound="40"

useArea="PERCENT" />

<progElement name="Developed" id="1" query="LULC\_A=20" weight="0.5"

evalType="INCREASING" lowerBound="0" upperBound="10" useArea="TOTAL"/>

</program>

The format of this file consists of a **<program>** tag containing individual Program Element descriptors. Individual Program Elements are defined by a **<progElement>** tag with the following attributes:

|  |  |
| --- | --- |
| **<progElement> Attribute** | **Description** |
| name | The name of the Program Element. This should match the name in the ENVISION Project file. (required) |
| id | The Model ID for this Program Element. This must match the ID in the ENVISION Project file. (required) |
| query | A spatial query that defines the Program Element. See Appendix 3 for a complete description of the query language. Any legal query can be used to define the Program Element. (required) |
| weight | If the Evaluative Model defines multiple Program Elements, this is the relative weight (0-1) of this Element. (required) |
| evalType | Specifies how this Program Element is calculated. For absolute quantities, use HIGH or LOW. For trends (trajectories), use INCREASING OR DECREASING. These are defined as follows:   * HIGH: Abundance is defined by areas of the Program Element near the maximum target. Used when the goal is to maximize the Program Element. * LOW: Abundance is defined by areas of the Program Element near the minimum target. Used when the goal is to minimize the Program Element. * INCREASING: Abundance is defined by the trend, expressed as percent change/year of the Program Element near the maximum target. Used when the goal is to maximize the trend line of the Program Element. * DECREASING: Abundance is defined by the trend, expressed as percent change/year, of the Program Element near the minimum target. Used when the goal is to minimize the trend line of the Program Element.   (required) |
| lowerBound | A numerical value defining the lower target of the Program Element (required) |
| upperBound | A numerical value defining the upper target of the Program Element (required) |
| useArea | Indicates whether the Program Elements is defined be total area or percent of the landscape. Possible values are:   * PERCENT: Specifies that the Program Element is defined as a percent of the landscape area. * TOTAL: Specifies that the Program Element is defined as total (absolute) area. |

To include a Program in an application, include the **<model>** element shown below in the **<models>** section of the Project file. Note that the Program Xml input file is given as the initInfo. This Project entry corresponds to the program file given above. Note that specified files must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<models>

<model

name ='Forest Lands'

path ='program.dll'

id ='0'

use ='1'

freq ='1'

decisionUse ='1'

showInResults='1'

fieldName =''

initInfo ='Bainbridge\_program.xml'

gain ='1.000000'

offset ='0.000000'

/>

</models>

## Spatial Allocator (Autonomous Process)

The Spatial Allocator allows you define global allocation processes that attempts to allocate specific “Allocations” on the landscapes to meet define targets. Allocations are organized into “Allocation Sets” that represent collections that are represented by a single field in the IDU database. An Allocation is defined by a set of target values that can be defined in a number of different ways (see attribute table below). Additionally, temporal “sequences” of the allocated item can be defined, e.g. to represent a crop rotation sequence. An item defined by an allocation is assigned to the landscape using a scoring systems defined by “constraints” and “preferences”. Constraints are spatial queries that are required to be satisfied before an item can be allocated to an IDU. Preferences define scores that additively accumulate if the underlying IDU satisfies the spatial query associated with the IDU.

Multiple allocations within an allocation set “compete” to be assigned to an IDU based on their preference scores. The Spatial Allocator process attempts to allocate items within an allocation set to meet the allocation target(s), using the accumulated preference scores to determine where on the landscape allocations should be assigned.

When an allocation is assigned to an IDU, the effect of the allocation can optionally be “expanded” to adjacent IDUs by specifying a spatial query that must be satisfied for the allocation to expand, and/or a maximum expansion area. The Spatial Allocator processes will attempt to expand the allocation to adjacent IDUs until either the maximum area is achieved or no more adjacent polygons satisfy the associated expand query.

Any number of Allocation Set and Allocations can be defined In a given input XML file.

An example Spatial Allocator input file is given in below:

The format of this file consists of a <**spatial\_allocator**> tag containing individual Allocation Sets. Individual Allocation Sets are defined by a <**allocation\_set**> tag with the following attributes:

<?xml version="1.0" encoding="utf-8"?>

<!--

<spatial\_allocator>

area\_col: name of column with IDU areas

method: "score priority" (default) or "best wins"

-->

<spatial\_allocator area\_col="AREA" method="score priority">

<allocation\_set name="Fire" col="DISTURB" >

-->

<allocation name="High Severity Fire" id="22" target\_type="file"

target\_values="fire.csv" expand\_query="LULC\_B=3"

expand\_area="U(2000,50)" >

<constraint name="only forest" query="LULC\_A=3" />

<preference name="slope" query="Slope>20" weight="0.4" />

<preference name="global" query="AREA &gt; 0" weight="0.1" />

</allocation>

<allocation name="Stand Replacing Fire" id="23" target\_type="file"

target\_values="fire.csv" expand\_query="x" expand\_max\_area="X">

<constraint name="only existing ag" query="LULC\_A=2" />

<preference name="Existing Rotation" query="ROTATION=101" weight="0.4" />

<preference name="global" query="AREA &gt; 0" weight="0.1" />

</allocation>

<allocation name="BugA" id="22" target\_type="file" target\_values="fire.csv"

expand\_query="x" expand\_max\_area="X" expand\_type="distribution" >

<constraint name="only forest" query="LULC\_A=3" />

<preference name="slope" query="Slope>20" weight="0.4" />

<preference name="global" query="AREA &gt; 0" weight="0.1" />

</allocation>

<allocation name="BugB" id="23" target\_type="file" target\_values="fire.csv"

expand\_query="x" expand\_max\_area="X">

<constraint name="only existing ag" query="LULC\_A=2" />

<preference name="Existing Rotation" query="ROTATION=101" weight="0.4" />

<preference name="global" query="AREA &gt; 0" weight="0.1" />

</allocation>

</allocation\_set>

</spatial\_allocator>

|  |  |
| --- | --- |
| **<spatial\_allocator> Attribute** | **Description** |
| area\_col | Name of column with IDU areas (optional, defaults to “AREA”) |
| method | "score priority" (default) or "best wins" |

|  |  |
| --- | --- |
| **<allocation\_set> Attribute** | **Description** |
| name | Name of column with IDU areas (required) |
| col | IDU field associated with this AllocationSet, populated with Allocation IDs (optional, if not provide, allocation IDs are not written to the IDU coverage) |
| sequence\_col | IDU field populated with sequence ID's (optional, required if using sequences) |
| use | 0=turn this allocation off, 1=turn this allocation on. Can be overridden by scenario variables (optional, default='1') |

Within an <**allocation\_set**> tag, individual allocations are defined using the <**allocation**> tag as follows:

|  |  |
| --- | --- |
| **<allocation> Attribute** | **Description** |
| name | Name of the item to be allocated (required) |
| id | Unique attribute identifer for the item (required)  Notes:  1) if the item is NOT a sequence, then the ID is the attribute code for the item.  2) if the item IS a sequence, then the ID is not used to identify initial |
| target\_type | 'rate' - indicates the value specified in the 'target\_values' attribute is a growth  rate. This requires that the target column (specified in the Allocation Sets ‘col’ attribute be populated in the initial IDU coverage)  'file' - load from a conformant CSV file, filename specified in 'target\_values'. Conformat are text files, first row containing column headers, remaining rows containing /target value pairs, where time=calendar year or 0-based, depending on whether a start year is specified in the envx file), and target value= landscape target for the iteming being allocated  'timeseries' - use specified time series value pairs specified in the 'target\_values' attribute  (required) |
| col | IDU field used to calculate targets. This is what is accumulated during the allocation process, measured in units of the target. This can be thought of as an IDU field that stores what is being specified by the allocation target. For example, if the target is measured in $, this would specified a field that has the cost of the item being allocated for each IDU. If the targets are areas, then the field containing the area of the IDU would be specified  (optional, defaults to "AREA" ). |

|  |  |
| --- | --- |
| target\_values | for rate types, indicates the rate of growth of the item being allocated (decimal percent)  for file types, indicates full path to file containing actual year-based time series values for the allocation target variable. NOTE: if target\_domain=’pct\_area’, these should be defined using decimal percents.  for timeseries type, a list of (time,value) pairs giving an actual year-based time series values for the allocation target variable  eg. '(2007,500), (2010,600), (2040,800)' . NOTE: if target\_domain=’pct\_area’, these should be defined using decimal percents.  (required) |
| target\_domain | Specifies the domain for which target values are defined. Valid values include:  'area' - allocation target is an area (default)  'pct\_area' - allocation targets are percent areas. iarget\_value’s should be expressed as decimal percentage values. See 'target\_query' and “target\_value’ for additional information.  If target\_type is ‘rate’, target\_domain is ignored.  (optional, default is 'area') |
| target\_query | Spatial query that specifies that the area that the target values apply to. Only used if target\_domain='pct\_area'  (optional) |
| score\_col | IDU column holding calculated score for the allocation (optional) |
| sequence | For sequences, a comma-separated list of ID's defining the sequence (e.g. a crop rotation) |
| init\_pct\_area | For sequences, the (decimal) percent of the initial land area that is in a sequence (required if sequence is specified) |
| expand\_query | Indicates an allocation should be expanded when applied, to neighbor cells that satisfy the query. Can use kernal references! |
| expand\_area | Indicates the maximum area allowed for an expansion. can be a fixed value (eg 400) or a distribution. Distribution can be specified as:   * Uniform: 'U(min,max)' * Normal: 'N(mean,stdev)' * Weibull: 'W(a,b)' * LogNormal: 'L(mean,stdev)' * Exponential: 'E(mean)' |
|  |  |
|  |  |

To include Spatial Allocator in an application, include the <**autonomous\_process**> entry shown below in the <**autonomous\_processes>** section of the Project file. Note that the Spatial Allocator Xml input file is given as the last argument. Note that specified files must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<autonomous\_processes>

<autonomous\_process

name ='Allocations'

path ='spatialAllocator.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='myalloctions.xml'

/>

</autonomous\_processes>

## Sync (Autonomous Process)

**Sync** is an Envision Extension that allows changes in one IDU column’s attributes to cascade into changes into other IDU columns as attributes dynamically change during the course of an Envision run. It is useful for synchronizing related data fields during the course of a run. For example, suppose a particular Envision application models landscape disturbances, expressed in a column called “DISTURB”. Further suppose that a policy sets the IDU’s DISTURB value to “2” to represent a fire, and that any time this occurs, an IDU field named “STAND\_AGE” should be set to “0”. **Sync** provides an easy to use mechanism to propagate these changes without writing C++ code. Instead, the application developer creates an XML file that describes the necessary mappings between sources of the change and resulting outcomes to be propagated whenever that change is encountered. Once the XML file is created, an autonomous process is added to the applications project (.ENVX ) file that references sync.dll and includes the path of the Xml file as its initialization string.

Sync Elements are defined in terms of mappings between changes in “source” values (specified for a particular column in the IDU database ) and corresponding changes in the “target” value(s) that are propagated in the target IDU column. Multiple target outcomes (values) can be specified probabilistically for a given source value. Any IDU field or fields can be used to define a sync element.

The method of propagation is defined with the <sync\_map> *method* attribute. Two methods are supported: “useDelta”, which propagates changes by scanning the delta array at each invocation of the autonomous process, looking for changes to the source value, and “useMap” which scans the entire map at each invocation of the autonomous process, looking for source values for each IDU. For both cases, the timing (pre-year or post-year) for the invocation of the autonomous process is indicated in the appropriate entry in the project (.ENVX) file defining the autonomous process.

The Sync process and individual Sync elements are defined in an Xml input file that is specified in the Project (.ENVX) file. An example input files is given below. This example specifies one <**sync\_map**> element specifying mappings for one source column; multiple <**sync\_map**> elements may be included within a <**sync**> tag to specify multiple columns to be synchronize. In the example below, the source column triggering changes is LULC\_A; the target column receiving synchronized changes is STANDCLASS. For LULC\_A values of “1”, two possible outcomes are defined for STANDCLASS – “4” gets populated in the STANDCLASS column twenty percent of the time, and “5” gets populated eighty percent of the time.

<?xml version="1.0" encoding="utf-8"?>

<!--

Sync specifies a set of mappings between variables in one column to variables in another column.

method: "useDelta" (default) or "useMap"

-->

<sync>

<sync\_map source\_col="LULC\_A" target\_col="STANDCLASS" method="useDelta">

<map source\_value="1">

<outcome target\_value="4" probability="20" />

<outcome target\_value="5" probability="80" />

</map>

</sync\_map>

</sync>

The format of the Xml input file consists of a **<sync>** tag specifying individual sync mapping (described within <**sync\_map**> tags. Within a <**sync\_map**> tag, possible outcomes for each source value are described using <**map**> and <**outcome**> tags.

|  |  |
| --- | --- |
| **<sync\_map> Attribute** | **Description** |
| source\_col | The name of the source column in the IDU coverage. (required) |
| target\_col | The name of the target column in the IDU coverage. (required) |
| method | Specifies the method used to scan for changes in the IDU coverage at run-time. Options include:   * useDelta: (default) scan the delta array for changes. * useMap: iterate through the entire IDU map, propagating changes to the target column when an idu’s srouce column attribute matches the value specified in the sync\_map   (optional) |
| <**map**> Attribute |  |
| source\_value | Specifies the value of the source column that will trigger propagation of an outcome. This can be a single value, a range of values (specified as ‘minVal-maxVal’, e.g. 5-8, or a wild card character (\*) that matches any value. (required) |
| <**outcome**> Attribute |  |
| target\_value | Specifies the value of the target column that will be propagated when a corresponding source value is encountered. (required) |
| probability | The probability that this outcome will be propagated, expressed as a percent (0-100) (required if more than one outcome specified, or if for a single outcome it is only propagated part of the time) |

To include a Sync in an application, include the **<autonomous\_process>** element shown below in the **<autonomous\_process>** section of the Project file. Note that the Sync Xml input file is given as the initInfo. This Sync entry corresponds to the Sync file given above. Note that the specified file must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<autonomous\_processes>

<autonomous\_process

name ='Sync for LULC'

path ='sync.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='sync\_input.xml'

/>

</autonomous\_processes>

## Target (Autonomous Process)

The Target process allows for defining “capacity surfaces” and “existing density surfaces” and move the “existing density surface” towards the “capacity surface” according to some growth rate in the attribute being modeled. It is typically used to model the spatial allocation of human population growth on a landscape, but can be applied to any other phenomena that conforms to this conceptual model.

A “target” defines a set of allocation sets that define the capacity surface and an associated growth rate. Any number of targets and allocation sets can be defined, depending on the needs of the model. An “allocation set” is a collection of individual allocations that represent a complete capacity surface definition for the specified target variable, with each allocation define capacity for particular regions of the landscape. Typically, if there are differences across scenarios in an allocation set, a distinct allocation set is defined for each scenario in the target specification.

The capacity surface is defined by specifying an equation for the capacity of portions of the IDU coverage defined by a given allocation’s spatial query. The equation can reference underlying IDU attributes if needed. For example, if the capacity surface is defined in terms of underlying zoning, then the capacity of each zone would specified in the equation, and the allocation’s spatial query would specify which zone that equation applied to.

When a Target model runs, the Target process attempts to move the “existing capacity surface” representing some spatially distributed variable of interest (e.g. population density), toward the capacity surface at a rate controlled by the Target’s grow rate specification. The allocation of new increments of population are spatially allocated proportionally to difference between the existing density and the capacity surface, resulting in the model moving the existing density surface towards the capacity surface. This function can be modified by introducing preference factors into the allocations; these preference factors in effect modify the differences between the existing density surface and capacity surface based on underlying IDU values, defined via a spatial query associated with the preference.

Target-based models are defined in terms of an Xml-based input file. The format of this file consists of a <**target\_process**> tag containing individual Target models. Individual Target models are defined by a <**target**> tag with the following attributes:

|  |  |
| --- | --- |
| **<target> Attribute** | **Description** |
| Name | Name of model (required) |
| description | A text description of the model (optional) |
| method | Method used to define growth rates for the model. Valid values include "rateLinear” or “rateExponential” See the ‘value’ attribute for additional information (required) |
| value | For method = ‘rateLinear’ or ‘rateExp’, indicates the growth rate, as a decimal percent. (required) |
| col | Specifies which IDU field contains the existing density surface values. This must be present and populated with initial values in the IDU coverage (required) |
| capCol | Specifies which IDU field Target will use to populate capacity values (absolute, not densities). If it is not present in the IDU coverage, it will be automatically added. (required) |
| availCol | Specifies which IDU field Target will use to populate available capacity values (absolute, not densities), e.g. the difference between the capacity and the existing density. If it is not present in the IDU coverage, it will be automatically added. (required) |
| pctAvailCol | Specifies which IDU field Target will use to populate percent available capacity values (decimal percent), e.g. the (difference between the capacity and the existing density)/capacity. If it is not present in the IDU coverage, it will be automatically added. (required) |
| areaCol | Specifies which field in the IDU coverage stores the area of the IDU, in units consistent with the coverage. If not present, defaults to “AREA”. (optional) |
| prefsCol | Specifies which IDU field Target will use to populate preference calculations. If not specified, preferences are calculated, but not stored in the IDU coverage. (optional) |
| query | A spatial query that determines the extent of application of the specified model. If not specified, the model is applied everywhere an associated allocation is defined (optional) |
| estimate\_params | Target can be run in “parameter estimation” model by setting this to ‘1’. In this mode, successive runs of the model within an Envision session will adjust any preference factors to meet “targets’ defined for each allocation (see below). For each run, adjusted preference values for each allocation are written to the output window. Typically, the model is run successively until the preference settings converge to provide good estimated between the allocation targets and modeled values. Think of this as a manual iteration process a la Newtons’ method.(optional) |

Within a <target> tag, a number of children tags are allowed, including any number of <report> tags, <const> tags, and <allocation\_set> tags.

The <report> tag generate output summaries of the target variable.

The <const> tag defines constants variables that can be used in allocation equations.

The <allocation\_set> tag defines a scenario-specific set of allocations.

Definitions of these tags are given below:

|  |  |
| --- | --- |
| **<report> Attribute** | **Description** |
| name | Name of report variable (shows up in the Post-Run Results window) (required) |
| query | Spatial query associated with the report (required). The output generated by the report indicates the percentage of growth allocated to the area defined by the spatial query. (required) |

|  |  |
| --- | --- |
| **<const> Attribute** | **Description** |
| name | Name of constant (required) |
| value | integer of floating-point values assigned to the constant (required) |

|  |  |
| --- | --- |
| **<allocation\_set> Attribute** | **Description** |
| name | Name of allocation set. Generally, this corresponds to the scenario the allocation is used in. (required) |
| Id | Unique identifier (integer) used to define a particular allocation set (required) |

Within an <**allocation\_set**> tag, capacities for individual allocations are defined using the <**capacity**> tag as follows:

|  |  |
| --- | --- |
| **<capacity> Attribute** | **Description** |
| name | Name of the capacity definition, generally related to the location on the landscape in which this allocation is operative. (required) |
| query | Spatial query defining where in the landscape this capacity definition applies (required) |
| value | A mathematical expression defining the capacity for the IDUs satisfying the query above. This expression can reference <const> variables defined above, as well as IDU field names. See Appendix 4 for more details on the expression language used in Envision. (required) |
| multiplier | reserved for future use (optional) |

The “capacity” surface can be “warped”, or biased, based on underlying spatial information using “preferences” – these in effect tend to bias the allocation process towards or away from particular locations, depending on the value associated with the preference. Preference tags are nested inside <allocation\_set> tags, similar to <capacity> tags. Preference tags have the following attributes:

|  |  |
| --- | --- |
| **<preference> Attribute** | **Description** |
| name | Name of the preference, generally related to the location on the landscape in which this allocation is operative. (required) |
| query | Spatial query defining where in the landscape this preference is defined (required) |
| value | A numeric value specifying the multiplier used to warp the capacity surface. (required) |
| multiplier | reserved for future use (optional) |

To include Target in an application, include the <**autonomous\_process**> entry shown below in the <**autonomous\_processes>** section of the Project file. Note that the Target Xml input file is given as the last argument. Note that specified files must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<autonomous\_processes>

<autonomous\_process

name ='Population'

path ='Target.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='TargetPopulation.xml'

/>

</autonomous\_processes>

An example input file for target follows is available at <http://envision.bioe.orst.edu/StudyAreas/Tutorials/TargetPopulation.xml>

## Trigger (Autonomous Process)

**Trigger**, like Sync, is an Envision Extension that allows changes in one IDU column’s attributes to cascade into changes into other IDU columns as attributes dynamically change during the course of an Envision run. It is useful for synchronizing related data fields during the course of a run. If differs from Sync in the following ways:

|  |  |  |
| --- | --- | --- |
| **Function** | **Sync** | **Trigger** |
| Iteration Method | Delta array or Map | Map only |
| Query Method | Single Column, Single Value | Spatial Query |
| Multiple Possible Outcomes | Yes | Yes |
| Multiple Possible Outcomes Can Reference Different Fields | No | Yes |
| Probabilistic Outcomes | Yes | Yes |
| Outcomes Value | Constant | Map Expression |

The Trigger process and individual triggers are defined in an Xml input file that is specified in the Project (.ENVX) file. An example input files is given below. This example specifies one <**trigger**> element specifying a spatial query indicating which IDUs will fire the trigger. When an trigger fires for an IDU, an outcome is selected probabilistically from the associated outcome list. The **target\_value** can be any valid Map Expression (see Appendix 4.) In the example below, the source query triggers an event whenever the STANDE\_AGE field is less than 10. The target column receiving changes is STAND\_AGE in one case, LULC\_C in the second case.

<?xml version="1.0" encoding="utf-8"?>

<!--

Sync specifies a set of mappings between variables in one column to variables in another column.

method: "useDelta" (default) or "useMap"

-->

<triggers>

<trigger query="STAND\_AGE < 10" >

<outcome target\_col="STAND\_AGE" target\_value="STAND\_AGE+1" probability="20" />

<outcome target\_col="LULC\_C" target\_value="4" probability="80" />

</trigger>

</triggers>

The format of the Xml input file consists of a **<triggers>** tag specifying individual trigger definitions (described within <**trigger**> tags. Within a <**sync\_map**> tag, possible outcomes for each source value are described using <**map**> and <**outcome**> tags.

|  |  |
| --- | --- |
| **<trigger> Attribute** | **Description** |
| query | A spatial query identifying IDUs that will trigger the event. (required) |
| <**outcome**> Attribute |  |
| target\_col | Specifies the value of the target column that will be propagated when a corresponding source value is encountered. (required) |
| target\_value | Map Expression for computing value to populate the IDU triggering the event |
| probability | The probability that this outcome will be propagated, expressed as a percent (0-100) (required if more than one outcome specified, or if for a single outcome it is only propagated part of the time) |

To include a Trigger in an application, include the **<autonomous\_process>** element shown below in the **<autonomous\_process>** section of the Project file. Note that the Trigger Xml input file is given as the initInfo. This Sync entry corresponds to the Trigger file given above. Note that the specified file must be either in the ENVISION executable directory, or a fully-qualified path with directories must be provided.

<autonomous\_processes>

<autonomous\_process

name ='Trigger'

path ='trigger.dll'

id ='0'

use ='1'

timing ='0'

freq ='1'

sandbox ='0'

fieldName =''

initInfo ='trigger\_input.xml'

/>

</autonomous\_processes>

Chapter 11. Running ENVISION from the Command Line

ENVISION can be executed in “batch” mode by running from a command line using command line arguments. To do this, open a command window and navigate to the Envision directory (usual C:\Envision). Supported command line parameters and switches are described in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Command line switch** | **Arguments** | **Examples** | **Notes:** |
| <envxfile> | The name of the project (envx) file to load – must be a fully qualified filename or on the Window PATH to be found | **Envision.exe \TestDir\test.envx**  **Envision.exe “\My Dir\test.envx”** | If the path to the envx file contains spaces, it must be enclosed in quotes |
| /r:<scnIndex> – Run a given scenario and exit | The one-based index of the scenario to run, or 0 to run all scenarios | **Envision.exe MyProj.envx /r:0** – run all scenarios and exit  **Envision.exe MyProj.envx /r:2** – run the second scenario and exit | Requires envx file be specified on the command line in addition to the /r switch if run in batch mode |

## envcmd Utility

The envcmd utility provides capabilities for running commands in batch files.

**Usage 1: envcmd /c:cmdFile <-- run commands in specified file**

**Usage 2: envcmd <switches>**

/i:inputShapeFile -- specifies path to shape file to use for input

/o:outputFile -- specifies output file

/s:"query" -- subset and export based on the specified query

/t:ReachIDField -- fix topology (line shape files only) - ReachIDField is optional

Example: > envcmd /i:"\envision\studyareas\WW2100\idu.shp" /s:"SUB\_AREA=1"  
 /o:"\envision\studyareas\ww2100\mckenzie\idu.shp"

# Appendix 1. Example LulcTree File

<?Xml version="1.0" encoding="utf-8"?>

<!-- Land Use Land Cover input file for ENVISION -->

<!-- Note: classifications sections must be order from high to low levels -->

<lulcTree cols="LULC\_A,LULC\_B,LULC\_C">

<!-- top level classification -->

<classification name="LULC\_A" level="1">

<lulc id="10" name="Water" rgb="0,0,189" />

<lulc id="20" name="Developed" rgb ="20,221,0" />

<lulc id="30" name="Barren" rgb="127,127,127" />

<lulc id="40" name="Forested Upland" rgb="0,128,0" />

<lulc id="50" name="Shrubland" rgb="128,64,0" />

<lulc id="70" name="Grassland/Herbaceous Upland" rgb="0,183,0" >

<lulc id="80" name="Agriculture" rgb="96,96,32" >

<lulc id="90" name="Wetlands" rgb="64,128,128" >

</classification>

<!-- second tier classification comes next -->

<classification name="LULC\_B" level="2">

<lulc id="10" parentID ="10" name="Water" rgb="0,0,189"/>

<lulc id="20" parentID ="20" name="Developed" rgb ="20,221,0" />

<lulc id="30" parentID ="30" name="Barren" rgb="127,127,127" />

<lulc id="40" parentID ="40" name="Forested Upland" rgb="0,128,0" />

<lulc id="50" parentID ="50" name="Shrubland" rgb="128,64,0" />

<lulc id="70" parentID ="70" name="Grassland/Herbaceous Upland" rgb="0,183,0" />

<lulc id="80" parentID ="80" name="Agriculture" rgb="96,96,32" />

<lulc id="90" parentID ="90" name="Wetlands" rgb="64,128,128" />

</classification>

<!-- third tier classification comes next -->

<classification name="LULC\_C" level="3">

<!-- Water -->

<lulc id="11" parentID ="10" name="Open Water" rgb="0,0,235" />

<lulc id="12" parentID ="10" name="Perennial Ice/Snow" rgb="255,255,255" />

<!-- Developed -->

<lulc id="21" parentID ="20" name="Developed - Open Space" rgb="127,255,127" />

<lulc id="22" parentID ="20" name="Developed - Low Intensity" rgb="127,127,127" />

<lulc id="23" parentID ="20" name="Developed - Medium Intensity" rgb="63,63,63" />

<lulc id="24" parentID ="20" name="Developed - High Intensity" rgb="30,30,30" />

<!-- Barren -->

<lulc id="31" parentID ="30" name="Barren Land" rgb="255, 160,160" />

<lulc id="32" parentID ="30" name="Unconsolidated Shore" rgb="235,235,235" />

<!-- Forested -->

<lulc id="41" parentID ="40" name="Deciduous Forest" rgb="0,92,0" />

<lulc id="42" parentID ="40" name="Evergreen Forest" rgb="0,196,0" />

<lulc id="43" parentID ="40" name="Mixed Forest" rgb="0,128,0" />

<!-- Shrublands -->

<lulc id="51" parentID ="50" name="Dwarf Scrub" rgb="128,128,64" />

<lulc id="52" parentID ="50" name="Shrub/Scrub" rgb="128,128,64" />

<!-- herbaceous Upland-->

<lulc id="72" parentID ="70" name="Grassland/Herbaceous" rgb="0,183,60" />

<lulc id="73" parentID ="70" name="Lichens" rgb="60,183,0" />

<lulc id="74" parentID ="70" name="Moss" rgb="60,183,60" />

<!-- Planted/Cultivated-->

<lulc id="81" parentID ="80" name="Pasture/Hay" rgb="96,96,32" />

<lulc id="82" parentID ="80" name="Cultivated Crops" rgb="154,171,58" />

<!-- Woody Wetlands -->

<lulc id="90" parentID ="90" name="Woody Wetlands" rgb="100,128,128" />

<lulc id="91" parentID ="90" name="Palustrine Forested Wetland" rgb="100,128,128" />

<lulc id="92" parentID ="90" name="Palustrine Shrub/Scrub Wetland" rgb="64,128,100" />

<lulc id="93" parentID ="90" name="Estuarine Forested Wetland" rgb="64,100,128" />

<lulc id="94" parentID ="90" name="Estuarine Forested Wetland" rgb="64,128,128" />

<lulc id="95" parentID ="90" name="Emergent Herbaceous Wetland" rgb="80,140,150" />

<lulc id="96" parentID ="90" name="Palustrine Emergent Wetlands (Persistent)" rgb="100,140,150" />

<lulc id="97" parentID ="90" name="Palustrine Emergent Wetlands" rgb="80,140,100" />

<lulc id="98" parentID ="90" name="Palustrine Aquatic Bed" rgb="120,140,150" />

<lulc id="98" parentID ="90" name="Estuarine Aquatic Bed" rgb="80,140,120" />

</classification>

</lulcTree>

# Appendix 2. Field Info Descriptors

Field Info Descriptors are Envision’s mechanism for specifying and storing information about how to display map field (column) information. Envision provides a Field Information Editor (shown below). Internally, Field Info’s are stored as XML files. By default, these correspond to shape file names (with an .xml extension), but any file name can be used.

Field Info files consists of a list of <fieldInfo> tags. Attributes include the following fields:

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| label | Label for this field |
| level | When viewing the Map Tab, indicates the menu level where this field is available. 0=top level, 1= show in “Results” submenu. |
| mfiType | Type of bin:  0 = quantity bins, based on ranges of values between binMin, binMax  1 = category bins, based on unique values (binMax = binMin = 0.0f - ignored) |
| type (Note: DEPRECATED) | An integer flag specifying the data type of the field, as follows: CHAR=1, BOOL=2, UINT=3, INT=4, ULONG=5, LONG=6, FLOAT=7, DOUBLE=8, STRING=10 |
| count | The number of bins to generate for attribute values for this field. 0 implies unique bins. |
| displayFlags | An integer flag indicting how the thematic map for the field should be coded: MIXED=0, RED=1, GREEN=2, BLUE=3, GRAY=4, SINGLECOLOR=5, TRANSPARENT=6, BLUEGREEN=7, BLUERED=8, REDGREEN=9. |
| binStyle | In integer flag indicating how bins for attribute values for this field should be generated: 0=linear bins, 1=equal count bins |
| min | For numeric fields, the minimum value to use in binning. |
| max | For numeric fields, the maximum value to use in binning. |
| showInResults | Indicates whether ENVISION collects dynamic map information and makes those dynamic maps available in the Results tab for this field. 0=no, 1=yes. |
| decadal Maps | Determines whether this attribute is included in the decadal maps generated during a multitrun. |
| useInSiteAttr | Determines whether the policy editor shows this field as a site attribute (0=no,1=yes) |
| useInOutcomes | Determines whether the policy editor shows this field as an outcome (0=no,1=yes) |

Each <field> element contains and embedded **<attributes>** element that further contains individual **<attr>**  elements describing each bin. The **<attr>** tag is defined as follows:

|  |  |
| --- | --- |
| **Attribute Name** | **Definition** |
| label | Label for this bin |
| color | RGB triplet containing the color for this bin (e.g. ‘255,255,255’) |
| value | For category bins only, specifies the value of the attribute for this bin. |
| minVal | For quantity bins only, specifies the minimum value for this bin |
| maxVal | For quantity bins only, specifies the maximum value for this bin |

Appendix 3. ENVISION’s Spatial Query Language

ENVISION includes a simple query language and built in query compiler that can be used to 1) specify site attribute for policies, to indicate where in the landscape the policy may be applied, 2) to run queries on the map to locate IDU’s matching specific attribute criteria, and 3) to specify inputs into certain evaluative models and autonomous processes.

The syntax for this language is very straightforward and consists most fundamentally of *Field* | *operator* | *value* triplets. These may be logically ‘and’ed or ‘or’ed together and arbitrarily nested to create complex queries. Additionally, the Query language supports several spatial operators to incorporate neighborhood information into the query.

The <*Field* | *operator*| *value*> triplets are the fundamental query elements and are defined as follows:

* *Field* indicates a column name in the IDU database, e.g. LULC\_A. Any database column can be referenced using name. This argument is case-insensitive. Fields in the primary IDU database can by referenced simply by their field name (e.g. AREA). Fields in additional layers included in the application’s Project file can be referenced with an extended syntax that uses a period (.) operate to specify the layer name, field name, and database row to reference. The row reference can be an IDU column name, if the column contains row information for the referenced layer, or can be an explicit row reference. An example of the former is Roads.PaveType[ RoadRef ]. This statement indicates the query references a layer named “Roads”, with a field column named “PaveType”, and the query engine will look up a column in the IDU database named “RoadRef” to find the row in the Roads table that should be returned when the current IDU is being processed. This can be very useful when the IDU layer needs to refer to linked information on an additional map layer.
* *operator* is one of the relational operators; these include:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Definition** | **Applicable Types** | **Example** |
| =, != | Equal to, not equal to | Any (numeric, string) | LULC\_A = 3 |
| <, <= | Greater than, greater than or equal to | Any numeric | SLOPE < 10 |
| >, >= | Less than, less than or equal to | Any numeric | POPDENS >= 100 |
| |, & | Logical OR, logical AND: performs bitwise logical operations on the arguments | Integer | DEVELOP\_AS | 2 |
| $, CONTAINS | Searches for substrings within strings. The ‘$’ notation is shorthand for the full keyword | String | ADDRESS contains “Main”  ADDRESS $ “Main” |

Note that strings in an expression must be enclosed in double quotes, e.g. “This is a string”.

* *value* is an expression to be used in the evaluation of the relational operator. This can be a simple expression (e.g. a number or string), a field name (evaluating to the value of the field) or a mathematical expression (e.g. POPDENS\*100). Supported mathematical operations include addition (+), subtraction (-), multiplication (\*) and division (/).

Complex queries can be constructed using the AND and OR operator to connect the *field|operator| value* triplets, with optional nesting with parentheses. These operators are not case sensitive. Examples:

LULC\_A=10 AND DIST\_STR<100

(LULC\_A=20 and DIST\_STR < 100) OR (LULC\_A = 30 and DIST\_HWY > 10)

**Spatial Operators**

In addition to basic *field|operator|value* triplets, several spatial operators can be incorporated into a query. These operators are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Definition** | **Arguments** | **Example** |
| NextTo( *query* ) | Returns True if the IDU is adjacent to an IDU that satisfies the query argument | *query:* Any valid query | NextTo( LULC\_A = 3 ) |
| NextToArea( *query* ) | Returns the area of the region adjacent to an IDU that satisfies the query argument. The area returned is an *extended* area, meaning it finds all IDUs that define a continuous region around the target IDU, even if some of those IDU’s are not directly adjacent to the target IDU. | *query:* Any valid query | NextToArea( LULC\_A=40 ) > 500 |
| Within(*query, distance*) | Returns True is the IDU is within the specified distance of any IDU that satisfies the query argument | *query:* Any valid query;  *distance:* a numeric distance | Within( SLOPE < 10, 100 ) |
| WithinArea (*query,radius,percent)* | Returns True if the percent area of IDU’s satisfying the query argument within the circle around the center of the IDU is greater than the specified percentage. | *query:* Any valid query;  *radius:* a numeric distance;  *percent:* the decimal percent threshold | WithinArea( DENS<100, 1000, 0.20)  This is true if within a radius of 1000 of the centroid of the IDU, the DENS value of the IDU areas falling within that radius is less than 100 for at least 20 percent of the area. |

These spatial operators can be combined into complex queries in a manner similar to the *field|operator|value* triplets, e.g.:

LULC\_A=10 and NextTo( LULC\_A=30 )

**Constants**

One two constants are defined by the query language, **TRUE** (which evaluates to 1), and **FALSE** (which evaluates to 0). These are not case sensitive, so **TRUE**, **true**, and **TRue** all evaluate the same way. These are generally useful for comparing values returned from Boolean database fields.

**Embedding Comments in Spatial Queries**

Comments can be embedded in spatial queries to greatly improve readability and interpretability of the query string. Anything enclosed in matched braces ( { } ) is considered a comment and ignored by the Query compiler. For example:

LULC\_A=10 {Agricultural Lands} is treated identically as LULC\_A=10.

Appendix 4. ENVISION’s Mathematical Expression Language

ENVISION includes a mathematical expression language that can be used to evaluate map-based mathematical expressions. This is used in a variety of “standard” plug-ins, and can be accessed using user-defined plug-ins. This language allows for expressions that can contain:

1. Algebraic operators (e.g. add, subtract, etc.)
2. Logical operators (e.g. logical AND, logical OR, etc. )
3. A variety of mathematical functions (e.g. sin, cos, etc.)
4. Field references for accessing IDU-level field information (e.g. AREA, IDU\_INDEX, POPDENS)
5. Expression precedence grouping using parentheses.

Available operators and functions are specified below.

**Operators:**

|  |  |
| --- | --- |
| + - \* / ^ % & | !  >, >= <, <= !=  == | Addition Subtraction and unary minus Multiplication Division Power Modulo Logical AND Logical OR Logical NOT Greater or equal Smaller or equal Not equal Equal |

**General functions:**

|  |  |  |
| --- | --- | --- |
| abs acos asin  atan avg(x,y,z,…)  bin ceil cos  cosh | fact floor hex if isNaN log log10 max(x,y,z,…)  min(x,y,z,…) | rand() rand(min, max) round  sin sinh  sqrt  sum(x,y,z,…)  tan tanh |

**Numerical Approximation functions (with the Numerical Approximation plug-in):**

|  |  |
| --- | --- |
| derivate(expr, var, point)    trapezoid(expr, var, a, b, [step=0.1])    solve(expr, var, result, [vo=0], [tol=0.01], [maxIter=100]) | Compute a numerical approximation of the derivative of the expression with respect to one variable at the specified point  Compute a numerical approximation to the integral between **a** and **b** using the trapezoid's rule  Find the variable's value that yields the desired result using Newton's numerical approximation method |

**Date/Time functions (with the Date plug-in):**

|  |  |
| --- | --- |
| date datevalue day hour minute month | nowdate nowtime second time weekday year |

**Example of valid expressions include:**

1 + sin( 0.5 )

17 \* AREA + log( POPDENS ) Note: this assume AREA and POPDENS are fields in the IDU database

if ( area > 100, AREA \* 1000, 0 ) Note: syntax is if ( condition, expr when true, expr when false)

For more details on the expression evaluator used in Envision, see <http://www.codeproject.com/Articles/7335/An-extensible-math-expression-parser-with-plug-ins>

Appendix 5. Frequently Asked Questions

***Q: My plug-in wants to communicate information back to Envision.   How do I do this?***

1) For single values coming out of a model during a Run() invocation that you want Envision to know about (for plotting purposes, for example), e.g. variable totals across a study areas, you would want to have an internal (to the model) variable that stores the value during Run() and expose that variable as Appendix 5. Frequently Asked Questions

***Q: My plug-in wants to communicate information back to Envision.   How do I do this?***

1) For single values coming out of a model during a Run() invocation that you want Envision to know about (for plotting purposes, for example), e.g. variable totals across a study areas, you would want to have an internal (to the model) variable that stores the value during Run() and expose that variable as an output variable using EnvExtension::AddOutputVar(), generally invoked in Init() .  
    
2) For spatially explicit values stored in the IDU layer, use one of the EnvExtension::AddDelta() methods.  Envision generally won't let a plug-in set data into the IDU layer directly – AddDelta allows Envision to coordinate these writes.   
    
3) For spatially explicit values NOT represented in the IDU layer, (say, fish counts in a reach represented in the stream network MapLayer), you would just set the value in the layer using one of the MapLayer::SetData()  methods. This is allowed for non-IDU layer writes.

 4) For values that Envision doesn't need to know about, but you want to persist on disk, do that on your own using whatever I/O code you want.   
    
5) Finally, if you have tabular data that you want Envision to manage that isn't necessarily spatially explicit, there is a mechanism for handing this to Envision as a DataObject, but you probably don't need to do that.    
    
  
***Q: My plugin needs to add a column to the IDU database and populate it at run time. Can we create a new column in the IDU map layer by name?  If so, how will that translate into an int for the column?***

You can add columns easily –either externally to Envision with ArcGIS or similar tool, within Envision using the Data Preparation ribbon->Add Field(s) menu option, or from within a plugin using the EnvExtension::CheckCol() method, specifying CC\_AUTOADD.. CheckCol has the following prototype:

static bool EnvExtension::CheckCol( const MapLayer \*pLayer, int &col, LPCTSTR label, TYPE type, int flags );

Example usage (typically used in Init())

// check and store relevant columns  
EnvExtension::CheckCol( pContext->pMapLayer, m\_colFishPop, \_T("FishPop"), TYPE\_FLOAT, CC\_MUST\_EXIST /\*or CC\_AUTOADD \*/ );

Note that using the later approach, the column won't be saved into the IDU database on disk unless you explicitly tell Envision to save it.  Generally, if a column needs to exist for a given application, you should just add it to the IDU coverage prior to an Envision run (options 1 and 2 above).

Columns have a type associated with them. You set this from the UI for options 1 and 2 above, for option three, you pass a type argument (e.g. TYPE\_INT) in CheckCol() – see typedefs.h for a list of supported types and associated constants

When saving databases, keep in mind that the current state of the database is what's saved – so if you save after a run, you will right post-run values of the data out to the database – this is rarely what's intended.

***Q: I want to visualize an IDU column as a dynamic map.  How does one do this?***

1) Bring up the field information editor – it's on several of the ribbons.

2) Select the field of interest.  If it is not checked, check it – that will generate a field information entry for the field.

3) There is an option called "Show In Results".  Check that.  You'll probably want to populate the "attribute Values" stuff as well.

4) Save the field information (be default it will overwrite the original file specified in the envx file for e given map layer).

After a run, the specified field will show up on the "Post Run Results" tab under the "Maps" tree node.

During a run, the "Main Map" will generally be updated with whatever the currently specified field is – that's another way to see the mapped attribute.  The field info entry is not required in that case, but you can only see results during a run.