The NED-2 Forest Ecosystem Management DSS: The Integration of The Stand Visualization System, The Loftis REGEN Model, and Other Extensions

by

Zhiyuan Cheng

(Under the direction of Walter D. Potter)

Abstract

NED-2 is a sophisticated, intelligent, goal driven, and integrated multi-agent decision support system for forest ecosystem management. NED-2 currently integrates various forest mangement tools and models, including vegetation growth and yield models, wildlife models, management models for timber, ecology, water, and visual quality goals, GIS reporting tool, HTML report generating tool, etc..

This thesis describes recent work on the integration of the Stand Visualization System and the Loftis REGEN Model, as well as other extensions to NED-2. These tools and models provide functionalities of visualizing forest stands and modeling the regeneration process in forest ecosystems, respectively.

Index words: ecosystem management, decision support system, multi-agent system, knowledge-based system, blackboard architecture, the Loftis REGEN Model, the Stand Visualization System

The NED-2 Forest Ecosystem Management DSS: The Integration of The Stand Visualization System, The Loftis REGEN Model, and Other Extensions

by

Zhiyuan Cheng

B.Eng., Huazhong University of Science and Technology,
Wuhan, China, 2003

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2005

The NED-2 Forest Ecosystem Management DSS: The Integration of The Stand Visualization System, The Loftis REGEN Model, and Other Extensions

by

Zhiyuan Cheng

Approved:

Major Professor: Walter D. Potter

Committee: Donald Nute
Charles Cross

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2005

Dedication

This thesis is dedicated to my parents, Yong'an Cheng and Fengju Zhao.

Acknowledgments

I would like to thank Dr. Walter D. Potter, my major professor, Dr. Donald Nute, the Principal Investigator of the NED-2 project, and Dr. Charles Cross for being my committee members and advising my study in the AI Program.

I would like to thank Drs. H. Michael Rauscher, Mark J. Twery, Scott Thomasma, and Pete Knopp-members of the USDA Forest Service-for their help and support on my work on the NED-2 project.

Special thanks to the Graduate School of the University of Georgia and the USDA Forest Service for providing financial support for my study in the AI Program.

Special thanks to Dr. Stephen Fennell for his medical care, and Mr. Blaine Norris for his legal assistance.

Also special thanks to my friends from the church-Vanessa, Jeff, Caroline, David, Lynn-for making me feel like at home in the United States.

I would also like to thank my friends and classmates, as well as AI alumni-Astrid, Cy, Sarah, Xunyu, Cheng, Congzhou, Julian, John, Dennis, Arlo, David, Brian, Vineet, Gopika, Tuohy, Chris, Rucen, Reiman, Hajime, Makiko, Bing, Hong, Jing, Haining-for bringing me a pleasant life at UGA.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

In modern forest ecosystem management, forest managers would like to achieve various management goals, such as ecology, wildlife resources, water, forest health, visual quality, and so on. Various software tools and models have been developed and are used to assist forest managers to achieve these management goals. For example, database management systems are used to store and manipulate inventory data, vegetation growth and yield models are used to simulate silvicultural treatment plans, optimization methods (e.g. genetic algorithms) are used to maximize timber production, geographical information systems (GIS) are used to visualize forested lands, etc..

The challenge for forest managers, however, is how to effectively utilize appropriate tools to achieve their management goals. Forest ecosystem management is a process that involves extensive and sophisticated decision making using domain expertise. In order to free forest managers from the details of various decision support tools and enable them to concentrate on the design and analysis of management goals and treatment plans through a single interface that is both intuitive and easy to use, an integrated system is highly desired. The objective of the NED project of the USDA Forest Service is to develop such a system that integrates a number of decision support tools that are the most useful to forest managers into a complete goal-driven decision support process that leads forest managers through the process in a natural and intuitive way [14]. Since its first release, NED-1, the system has

expanded significantly as more and more tools have been incorporated, and the second generation of the system, NED-2, has been developed in close collaboration between the USDA Forest Service and the Artificial Intelligence Center at the University of Georgia.

1.2 The NED-2 Architecture

NED-2 is a sophisticated, intelligent, and integrated multi-agent decision support system. NED-2 is designed to integrate a variety of decision support tools and models so that forest managers can make use of them in a streamlined fashion through a single interface. Moreover, to meet the users' prospective needs the system is expected to be able to integrate new tools whenever they are available or required. To achieve these design objectives, NED-2 is designed, to have a blackboard architecture which serves as the central organizer and coordinator of the integrated agents and directly determines the design and implementation of the agents.

Here the blackboard is not physical but a central repository of data, facts, intermediate results, requests, responses, etc., which shares all the information among the agents. Specifically, the information is stored either in the form of Prolog clauses, such as those for the predicates `request/1` and `fact/4`, or in the NED-2 databases. Each of the agents either handles an external software tool or implements a model. The agents are implemented in Prolog, a high-level logic programming language, and they know how to use the tools or models to solve problems. But they obtain the users' requests for solving problems, as well as necessary data and knowledge, from the blackboard. After they solve the problems, they put the results back on the blackboard.

The predicate `request/1`, which takes a list of the task(s) requested as its argument, handles the interactions between the agents and the blackboard. During the working process, all the agents keep "watching" the blackboard to see if there is any request on it. Once an agent sees that a request for a certain task appears on the blackboard and it's able to do that task, it will go ahead to do it. After the agent finishes it will retract that request from

the task list and write relevant results on the blackboard. The results can also be either in the form of Prolog clauses, or raw data that are written to the database.

It also happens that if an agent finds that it can do a requested task, but some prerequisite task must be done first before it can continue, then it will write new requests on the blackboard and some other agent will take care of the prerequisite task. In this way the work flow of the agents forms a backward chaining process. An advantage of the blackboard architecture is that, the first agent does not have to know how other agents do the prerequisite task, or what information or knowledge other agents need to know in order to do it; so the design and implementation of the agents are independent of each other. This also leads to the fact that it makes it easy to integrate new tools and models. When we want to integrate a new tool or model, we just create a new agent that conforms to the existing formalism.

1.3 The NED-2 Internal Data Model

The internal data model of NED-2 determines the design of the NED-2 databases and the representation and organization of all kinds of information during a NED-2 session.

A management unit can be divided into stands. Each stand has its inventory data that represent its original status, such as location, size, species, tree observations, buildings, and other physical characteristics. Forest managers can develop candidate plans consisting of silvicultural treatments based on a set of management goals. In NED-2, a plan is also referred to as a scenario, and each scenario has a number that uniquely specifies a plan. When developing a plan, the user is able to select available treatments from a list. When the plan is developed, the user can simulate the plan and use the results to conduct goal analysis. As a result of a plan, various characteristics of the management unit can be changed. In order to record changes on the stands explicitly and clearly, a snapshot value is used to uniquely specify a stand in a particular scenario at a particular time. Each snapshot includes a set of plant observations for the overstory, understory, and groundcover components of the stand [16].

So the information we need in a NED-2 session includes: inventory data, available goals, treatment definitions, scenario designs, simulation results, and snapshots. In NED-2, all these data are stored in Prolog knowledge bases and the NED-2 databases. Also, the system must consult a meta-knowledge base regarding the structure of the NED-2 database [16].

Theoretically speaking, in terms of ontology, which means an explicit specification used to share domain knowledge among the agents, the ontology for NED-2 is incorporated into the design of the NED-2 database structure and the Prolog clauses that store data. The robust design of the ontology and the internal data model of NED-2 ensure efficient information exchange between the agents and all the external components integrated.

1.4 The NED-2 Agents

The agents of NED-2 are implemented in Prolog, a high-level logic programming language. The list of functions currently supported by NED-2 include inventory, inventory analysis, goal selection, treatment definition, baseline generation, plan development, plan simulation, goal analysis, GIS display, and report generation[12]. Each of these functions is supported by one or more agents. The current list of NED-2 agents, which are all written in Prolog, includes the following [8]:

- NED-2 Interface Agent
- NED-2 Treatment Development Agent
- NED-2 Treatment Set Selection Agent
- NED-2 Simulation Agent
- NED-2 Goal Analysis Agents
- NED-2 GIS Agent
- NED-2 Report Writer Agent

- NED-2 Planning Agent

This thesis describes the integration of two new agents: the SVS Agent and the Regeneration Agent, as well as other extenstions to NED-2. The SVS Agent integrates the Stand Visualization System software that visualizes forested stands for visual goal analysis. Chapter 2 gives an introduction to the Stand Visualization System software and describes the implementation of the SVS agent.

The Regeneration Agent integrates the REGEN Core Engine, which implements the Loftis REGEN Model. Since the Regeneration Agent needs to collaborate with the Simulation Agent extensively, the Simulation Agent has also been extended to support resimulation on a previously simulated stand from any scheduled time point to the end of a treatment plan. Chapter 3 gives an introduction to regeneration, the Loftis REGEN Model, and the REGEN for Excel software, and describes the implementation of the Regeneration Agent. Chapter 4 describes the communication between the Regeneration Agent and the Simulation Agent as well as the implementation of resimulation.

Chapter 5 covers miscellaneous extensions to NED-2.

Chapter 2

The Integration of the Stand Visualization System

2.1 the Stand Visualization System

2.1.1 Overview

The Stand Visualization System (SVS) is a product of the USDA Forest Service, Pacific Northwest Research Station. SVS can generate graphic images to display the visual appearance of forested lands and it helps forest managers analyze visual management goals. The Stand Visualization System (SVS) generates images depicting stand conditions represented by a list of individual stand components, e.g., trees, shrubs, and down material, using detailed geometric models [13]. SVS can display an individual tree using either a pre-defined or user-defined image for it, which helps the user verify or understand the nature of the tree. When displaying all the trees of a forested stand, SVS uses different colors and various marks for different species. SVS also provides overhead, profile, and perspective, three views, which allow the user to have a better understanding of the visual appearance of the stand. Multiple images can also be generated, either to display a stand at different years under a certain treatment plan, or under several treatment plans at a single year. This flexibility is quite useful in that the user can compare different treatment plans in visual goal analysis through a single interface after a single run.

2.1.2 Data Requirements of SVS

SVS uses two types of data: the treelist and the plant form definitions. A treelist contains tree records of a stand, which includes various parameters, such as species, size, location,

density, etc., which are used by SVS to generate appropriate images of the stand. Plant form definitions contain the necessary information that defines the appearance of individual plant species. There are default plant form definitions for plant species in the system and most of the time we use these default settings. If the user would like to change the default definitions he is allowed to do that when in SVS.

The user can either create a new stand or display an existing one. To construct appropriate SVS treelists, the user can either enter new stand data using the ASCII editor or read data from other external resources, such as a simple stand table or an FVS treelist file, in which case data format conversion is usually required. The structure of an SVS treelist may vary, depending on how many cycles it contains. A cycle is a time interval during which the trees on a stand either grow or receive certain treatment plans. If an SVS treelist only contains one cycle, the tree records of the stand are stored in a single `.svs` file. In such case, a typical `.svs` file may look like this:

```
;         |
;         |    Plant      Class   Tree
; Species |     ID     |Plnt|Crwn|Stat|  dbh |height| lang|
;-------------------------------------------------------...
EH         3011L000003     0    0    1  10.70  58.30   0.0
EH         3011L001003     0    0    1  10.70  58.30   0.0
              Crown      Crown       Crown       Crown
     end  Radius Ratio RadiusRatio RadiusRatio RadiusRatio
fang| dia |   1  |   1 |   2  |  2 |   3  |  3 |   4  |  4 |
-------------------------------------------------------...
 0.0  0.00    7.0 0.56    7.0 0.56    7.0 0.56    7.0  0.56
 0.0  0.00    7.0 0.56    7.0 0.56    7.0 0.56    7.0  0.56
Expans Mark  X Coor-  Y Coor-
Factor|Code| dinate | dinate |     Z
---------------------------------------
   1.0   0     32.47   105.17     0.00
   1.0   0    152.78    84.65     0.00
```

Figure 2.1: Sample `.svs` file with one cycle

If multiple cycles exist, the headers describing the treatment details of the cycles are stored in an `.svs` file, whereas the tree records of these cycles are stored in subsidiary files with the same name as the `.svs` file and ordinal extensions. For example, if an SVS treelist

has two cycles, and the headers of these cycles are stored in a file called `stand1.svs`, which may look like this:

```
#TREELISTINDEX
;
"001   NONE   Cycle:1   Year:2011" "...\test.001"
"001   NONE   Cycle:2   Year:2021" "...\test.002"
"001   NONE   Cycle:2   Year:2011" "...\test.003"
"001   NONE   Cycle:3   Year:2031" "...\test.004"
"001   NONE   Cycle:4   Year:2041" "...\test.005"
"001   NONE   Cycle:5   Year:2051" "...\test.006"
"001   NONE   Cycle:6   Year:2061" "...\test.007"
```

Figure 2.2: Sample `stand1.svs` file

Then the tree records of the two cycles could be stored in two subsidiary files called `stand1.001` and `stand1.002`, respectively. The two subsidiary files will have the same structure as an `.svs` file that contains the tree records of an SVS treelist that has a single cycle. Then SVS will read `stand1.svs` and use the records in the subsidiary files to generate an image for each of the cycles.

2.1.3 FVS2SVS

SVS provides three data conversion engines to convert external data files to SVS treelists. The TBL2SVS engine allows the user to convert simple stand tables to SVS treelists. The ORG2SVS engine can convert output files from the ORGANON growth model into a series of SVS treelists. What we are mainly concerned with is the FVS2SVS engine, which converts FVS treelists to SVS treelists.

An FVS2SVS executable is also available to allow the user to manually perform the conversion and with this facility we are able to make FVS and SVS work in a streamlined fashion. Our key task is then to implement a scheme that constructs an appropriate FVS treelist of the stand that we want to display and submits it to FVS2SVS. After FVS2SVS converts the FVS treelist to an SVS treelist, SVS can generate images of the stand requested.

It might seem more efficient to generate the input files for SVS directly rather than generate files in FVS format and convert them to SVS format. However, notice that the tree

information in an SVS treelist file includes $X-Y$ coordinates. This information is not stored in NED-2. Apparently, FVS2SVS has a method for computing reasonable $X-Y$ coordinates for a stand of trees. Rather than try to duplicate this, we decided to let FVS2SVS do this task for us.

2.2 The Integration of SVS into NED-2

The integration of SVS into NED-2 required a new agent that can process requests for displaying a stand using FVS2SVS and SVS. The behavior of this agent must conform to the existing blackboard and multi-agent architecture of NED-2. Moreover, it provides a graphical user interface embedded into NED-2 to allow users to use SVS within the NED-2 interface framework.

2.2.1 The SVS Agent

The new member of the NED-2 family of agents-the SVS Agent-is expected to work efficiently with its peers. During a NED-2 session, like all the other agents, the SVS Agent keeps watching the blackboard to see if there is any request for tasks that it can perform. Requests for displaying a stand are usually posted on the blackboard after the simulation of treatment plans. The user may want to see what a stand looks like at a certain year under different plans, or under a certain plan at different years, in order to analyze his visual goals. When such a request appears on the blackboard, the SVS Agent is triggered.

The SVS Agent consists of two parts, one of which is responsible for generating a graphical user interface when the agent is triggered and the other part collects necessary data from the NED-2 database and executes FVS2SVS and SVS to display the requested stand. The work flow of the SVS Agent is shown in Figure 2.3 on the next page.

There is an entry “Generate SVS Display” under “Analysis” in the NED-2 main user interface (A Pane). When the user clicks this entry, a request will be written on the blackboard to trigger the SVS Agent. As shown in Figure 4.3, when the agent is triggered, a user

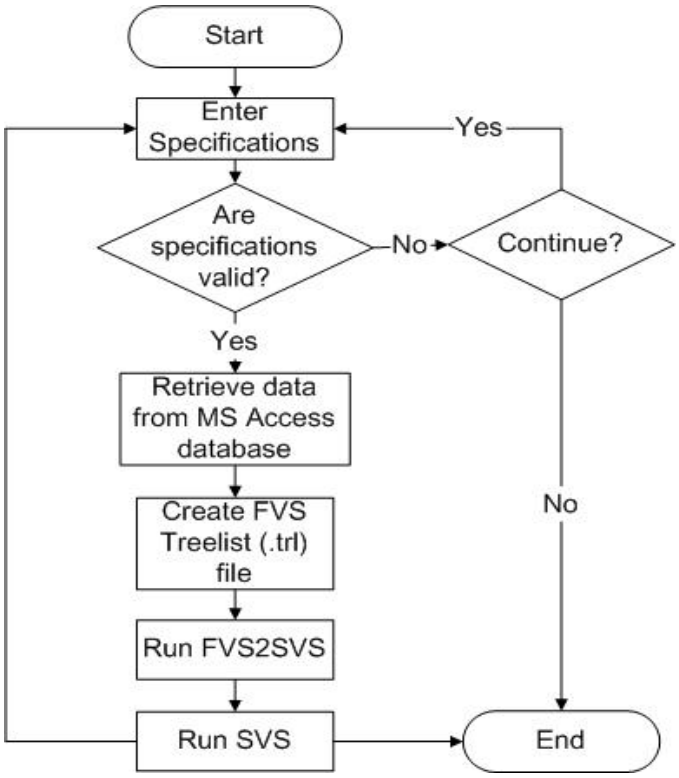

Figure 2.3: Work Flow of the SVS Agent

interface will pop up and prompt the user to explicitly specify his request, such as exactly which stand he wants to display, and whether he wants to display the stand at a certain year under different plans (by year) or under a certain plan at different years (by plan). After that the agent will collect relevant data for the user's specification from the NED-2 database. When all the data are ready they will be used to construct an FVS treelist. Then FVS2SVS is called to convert this FVS treelist to an SVS treelist, which is then used to generate (an) image(s) for the requested stand and scenario.

2.2.2 Interface Design

When the SVS Agent is triggered, it will create a graphical user interface (GUI) which is shown in Figure 2.4 on the next page.

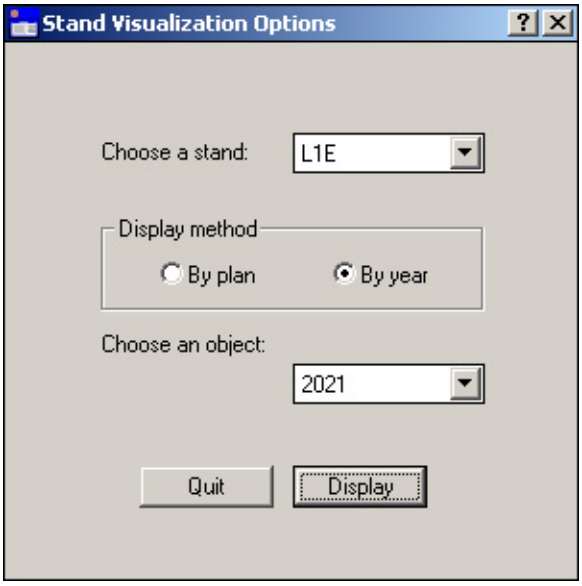

Figure 2.4: The SVS GUI

The request for SVS written on the blackboard is a `request/1` fact which may look like

```
request([svs|Rest]).
```

This request does not specify which stand the user wants to display or how it should be displayed. After the SVS Agent creates the GUI it will read from the NED-2 database the names of all the stands, plans, and treatment years into three lists. These lists are filled into the corresponding list boxes on the GUI so that the user can specify which stand he wants to display and how to display it. After the user chooses a stand, he can choose to display this stand either "by year" or "by plan". If he chooses "by year", he can specify a year in the list box and SVS will generate (an) image(s) of the stand under all treatment plans at that year. If he chooses "by plan", he can specify a treatment plan in the list box and SVS will generate (an) image(s) of the stand under that plan at all the projected treatment years. The user is allowed to choose all the scheduled treatment years, including the baseline year, but baseline generation itself is not treated as a plan here and it will not show up in the list of plans. All the underlying work to run FVS2SVS and SVS will be triggered if the user

clicks “Display”. Necessary bullet-proof routines are also provided to avoid invalid display specifications.

2.2.3 Data Acquisition

All the data needed by the SVS Agent can be obtained from the NED-2 database, either directly or through some manipulation. A NED-2 database is loaded into a NED-2 process before the user can start working on a management unit. At the beginning, this database contains inventory data, available goals, treatment definitions, scenario designs, and initial snapshots. The user then defines management goals, develops treatment plans, and runs simulations for the plans. During this process, for each of the plans and treatment years, a new snapshot is created and is associated with the plant records generated by FVS for the previous cycle under this plan. The SVS Agent is expected to allow the user to generate images for a target stand at any time and under any plan; so the key is to find the snapshot or list of snapshots that represents the user’s specifications.

However, the process of finding the target snapshot(s) depends on the data that we want to retrieve from the database. To construct an FVS treelist that can be converted to an SVS treelist that directs SVS to generate correct images for the target stand, for each of the tree observation we need to get the following variables: Cluster Number, Observation Number, Species Code, Trees per Acre, Mortality per Acre, DBH, Tree Height, Crown Ratio, and Maximum Crown Width. The variables Cluster Number and Observation Number are used to uniquely identify a plant record. Species Code is used to consult the species knowledge base to get the NED alpha species code of the plant that can be recognized by FVS and SVS. Trees per Acre and Mortality per Acre represent the population of this tree record. DBH, Tree Height, Crown Ratio, and Maximum Crown Width are needed to determine the shape of the individual trees.

When FVS simulates a treatment plan, part of the Simulation Agent’s work is to save some, but not all of the simulated data to the database. These data include values of most

of the variables we need to construct FVS treelists, specifically Trees per Acre, DBH, Tree Height, Crown Ratio, Maximum Crown Width. Cluster Number, Observation Number, and Species Code can be obtained from either the inventory data or the simulated data. The only variable whose values are not directly available is Mortality per Acre.

Mortality per Acre is the decrease of Trees per Acre of a tree observation during a treatment cycle, which represents the dead trees of the observation. The mortality value of a tree observation identified by a certain snapshot can be calculated by subtracting its Trees per Acre value at the snapshot from that of its previous snapshot so we need a list of consecutive snapshots pairs. In the approach of representation, the second element of each pair is the "target" snapshot, which is the one to be displayed, and the first element is the target snapshot's previous snapshot. The ways of obtaining such a list of pairs are slightly different, depending on whether the user chooses to display "by year" or "by plan". If the user chooses to display "by year", then after he specifies a stand and a year in the GUI, the SVS Agent will find the snapshots that identify all the treatment plans scheduled on the stand at that year. If the user chooses to display "by plan", then after he specifies a stand and a treatment plan in the GUI, the SVS Agent will find the snapshots that identify all the scheduled treatment years of the plan for the stand. In both cases, the agent will also find the previous snapshot for each of the snapshots as well.

Several situations in finding the previous snapshots are considered. First, if a snapshot identifies the baseline generation, then its previous snapshot is the one that identifies the input of inventory data for the stand. Second, if a snapshot identifies a year when there is only the default growth treatment, its previous snapshot is either the one that identifies the previous treatment, or the one that identifies baseline generation if this growth-only treatment is the first in the treatment sequence after baseline generation, and in such case the previous snapshot is not associated with the same scenario as the current one. Third, if there are more than one treatments at a single year, then only the snapshot of the last treatment is used to represent this year, and its previous snapshot is the one that identifies

its previous treatment year. All the above operations are implemented using appropriate SQL queries. A list of consecutive snapshots pairs that we obtain after the above process may look at this:

```
[[0,10],[42,43],[43,44],[44,45]].
```

With the list of consecutive snapshots pairs available, we can retrieve the Trees per Acre values associated with both snapshots in a pair to calculate the values of Mortality per Acre for each of the tree observations identified by the target snapshot. This way only works when the two snapshots in a pair identify the same tree observations. When the previous snapshot of a pair identifies a Clearcut treatment that cuts all trees on a stand, the target snapshot actually identifies new trees grown by regeneration, which are totally different from the previous trees. It doesn't make sense to calculate Mortality per Acre values here; so in such case the previous snapshot is not used and the Mortality per Acre values are set to be 0. Besides a Clearcut treatment, if in any case the target snapshot identifies a tree observation that is not also in its previous snapshot, the Mortality per Acre value of this observation is set to be 0.

To improve efficiency, at first all the tree observations identified by both the previous snapshot and the target snapshot are retrieved from the database and are stored in a previous treelist and target treelist. If an observation in the target list is also in the previous list, then the corresponding mortality value is calculated. Otherwise the mortality value is set to be 0. All the other required values can be directly obtained from the target treelist. Hence after processing the target treelist, we will obtain all the observation data required to construct an FVS treelist. An observation data tuple may look like this:

```
['Plan 1',[0,0,'LITU',5.769,-0.769,20.5,83,47,38.2]].
```

If the user chooses to display "by year", each of the tuples has a control field to indicate what plan it represents; otherwise it has a field to indicate what year it is at. These fields are used to write appropriate headers.

Values of some other control variables are also needed to generate headers for FVS treelists. The headers distinguish different treatment cycles or different plans. The data fields of a standard FVS treelist include: Header Indicator, Number of Rows, Cycle Number, Year, Stand ID, Management ID, FVS Variant and Version number, Date, Time, Type of List, Length of Cycle, Stand Sampling Weight, FVS Variant Revision Date, and Parallel Processing Code. To construct a valid FVS treelist for FVS2SVS, all of these fields are required. But SVS only needs Header Indicator, Number of Rows, Cycle Number, Year, Stand ID, and Management ID. Hence we only need to provide correct values for these fields and dummy values for the others.

Another concern is that SVS was only designed to generate multiple images for a stand at different years under a treatment plan, but not for a stand under different plans at a single year. In fact, the images generated depend only on what data an SVS treelist contains. When writing tree records of a stand under different plans at a single year to an FVS treelist, if we consider the Cycle Number as an indicator of different plans, the generated images will still make sense. In such case, we use the value "PLAN" for the Management ID field instead of the default value "NONE" in order to remind the user what he is viewing, because there is no facility available in SVS to tell this difference.

2.2.4 FVS Treelist Generation

The generation of an FVS treelist by the SVS Agent is actually an inverse process with respect to the generation of such a list by the FVS software. When FVS runs a simulation, it generates a treelist file (`.trl` file) containing all the relevant simulation results as its output. In NED-2, the Simulation Agent is responsible for saving some of these results in the treelist to the NED-2 database. A treelist output by FVS after simulation only contains tree observations of a stand at different years under a treatment plan. What the SVS Agent does is that it uses the data in the NED-2 database to reconstruct such an FVS treelist. The flexibility of this design is that the agent can also gather the data for a stand under different

treatment plans at a single year, into a centralized treelist which can be displayed using the SVS software so that the user can more conveniently compare the effects of different plans and perform his visual goal analysis.

When all the tree observation data tuples are retrieved from the NED-2 database, they are written to temporary files in the format of FVS treelists. A typical FVS treelist looks like this:

```
-999   4    1 2011 001                        NONE NE 6.21 10-02-2005 17:21:49 T 10
    3011   29 EH 16  3  0   3    6.793    0.207  10.7  1.31  58.3  4.1 56 14.0  0  26.46
    2005   14 RM 26  3  0   2   14.003    1.997   6.8  0.66  52.7  3.6 49 15.4  0  17.31
    2006   15 RM 26  3  0   2   99.322   41.678   2.7  0.58  30.8  7.7 42  7.3  0   4.62
    1005    4 AB 40  3  0   1   28.637    7.363   4.9  0.75  45.0  4.8 43 10.4  0  13.22
-999   4    2 2011 001                        NONE NE 6.21 10-02-2005 17:21:49 C 10
    3011   29 EH 16  3  0   3    6.793    0.000  10.7  1.31  58.3  4.1 56 14.0  0  26.46
    2005   14 RM 26  3  0   2   14.003    0.000   6.8  0.66  52.7  3.6 49 15.4  0  17.31
    2006   15 RM 26  3  0   2   99.322    0.000   2.7  0.58  30.8  7.7 42  7.3  0   4.62
    1005    4 AB 40  3  0   1   28.637    0.000   4.9  0.75  45.0  4.8 43 10.4  0  13.22
-999   5    2 2021 001                        NONE NE 6.21 10-02-2005 17:21:49 T 10
ES020021   21 RM 26  2  0   2   14.003    0.000   1.0  0.00   5.5  0.0 63  3.9  0  52.90
ES020022   22 RM 26  2  0   2   14.003    0.000   1.0  0.00   5.5  0.0 63  3.9  0  61.02
ES020023   23 RM 26  2  0   2   99.322    0.000   1.0  0.00   5.5  0.0 63  3.9  0  47.68
ES020024   24 RM 26  2  0   2   99.322    0.000   1.0  0.00   5.5  0.0 63  3.9  0 100.00
ES020005    5 WO 55  2  0   1    0.322    0.000   1.0  0.00   5.5  0.0 63  4.5  0  56.14
-999   5    3 2031 001                        NONE NE 6.21 10-02-2005 17:21:49 T 10
ES020021   21 RM 26  2  0   2    9.836    4.167   2.5  1.41  13.9  8.4 57  7.5  0  74.92
ES020022   22 RM 26  2  0   2    9.839    4.164   2.5  1.40  13.8  8.3 57  7.5  0  37.14
ES020023   23 RM 26  2  0   2   69.757   29.565   2.5  1.38  13.8  8.3 57  7.4  0  32.49
ES020024   24 RM 26  2  0   2   69.872   29.450   2.5  1.40  13.8  8.3 57  7.5  0  70.21
ES020005    5 WO 55  2  0   1    0.215    0.107   2.7  1.55  13.7  8.2 57  6.8  0  79.62
```

Figure 2.5: Part of an FVS Treelist

The first line of the treelist is the header for the first cycle, which contains the control information and distinguishes the tree records of this cycle from one another. In the construction of an FVS treelist, every cycle starts with a similar header line, although the "cycles" here may actually indicate different plans instead of different years, which depends on what values are in the control fields of the data tuples. The following process is straight-forward. All the values in the data tuples are written in corresponding fields of the FVS treelist,

except that values for the Tree Number field are obtained by combining the Cluster Number and the Observation Number (OBS) and the Tree Indices are arbitrarily set from 1 to the size of the set of data tuples for a cycle. Any field in such an FVS treelist that is not needed by FVS2SVS is set to 0, which serves as a dummy value. When a new value in the control field is encountered, a new header line is written to indicate a new cycle, which indicates either a new year or a new plan.

FVS2SVS is then called by the predicate `exec/3` to convert a constructed FVS treelist to SVS treelists. And finally SVS is also called by `exec/3` to generate images for the requested stand. A typical SVS display for a stand under different plans at a single treatment year looks like this:

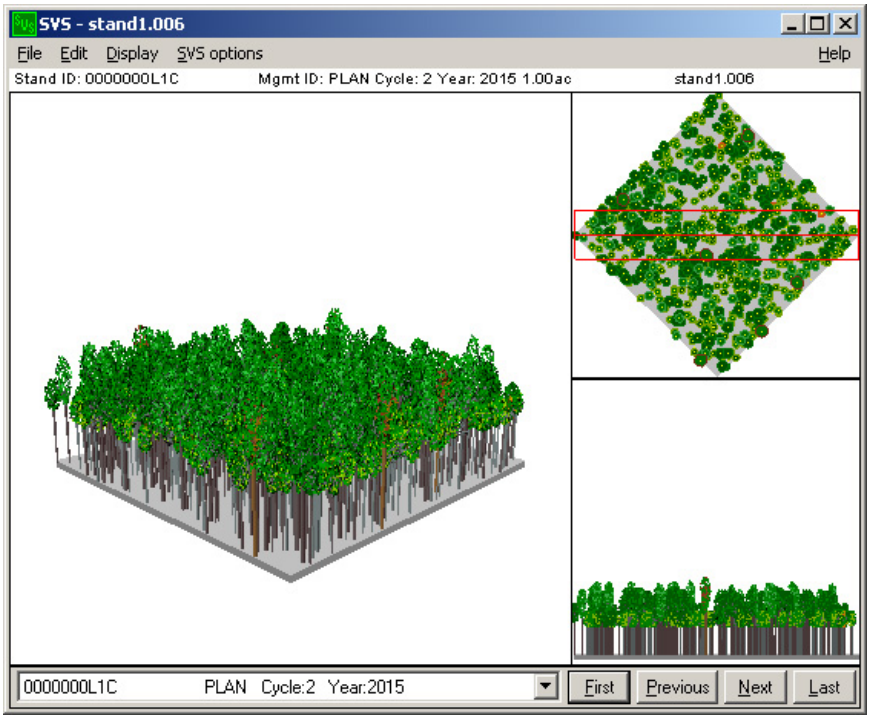

Figure 2.6: Typical SVS Display By Year

Chapter 3

The Integration of the Loftis REGEN Model

3.1 Regeneration and Regeneration Models

Regeneration is the ecological process where plants species emerge and grow into the overstory on forest stands that have experienced major disturbance, such as catastrophic natural fire or a planned clearcut. During the regeneration process, forest managers would like to know particularly what species can be expected to appear on a stand, as well as their expected densities and populations. To help the managers predict the outcome of regeneration, corresponding models are needed to simulate this process.

There are many theoretical models developed for regeneration, such as the ESTAB model [7], the SYSTUM-1 model [17], the ACORn model [4], and the RVM Model developed by USDA Forest Service and Oregon State University. Some well known forest simulation and management software packages also include regeneration models. For example, SILVAH [6] uses regeneration guidelines to offer prescriptions, and FVS includes the ESTAB Model. What will be mainly discussed in this chapter is the Loftis REGEN Model [9]. The Loftis REGEN Model is applied for the most part to silvicultral systems to describe post-harvest or post-disturbance performance of a given stand of trees [1]. Section 3.2 gives more details of the Loftis REGEN Model, and the REGEN Core Engine, which is a stand-alone Excel software module that implements the Loftis REGEN Model. Section 3.3 briefly describes the user interface for the REGEN Core Engine. Section 3.4 details the integration of the REGEN Core Engine into NED-2.

3.2 The Loftis REGEN Model and The REGEN Core Engine

The Loftis REGEN Model can be used to predict the expected composition of species on forest stands that have experienced major disturbance. The most typical "major" disturbance could be events like a catastrophic natural forest fire or a clearcut treatment, in which cases regeneration will be triggered and simulated. The triggering conditions as proposed by David Boucugnani are: (1) basal area is less than or equal to 50 $ft^2/acre$ or (2) opening size is greater than 0.25 acres and the base area within the opening is between 0 and 5 $ft^2/acre$ for localized events [1]. The triggering conditions, however, do not affect how the REGEN Core Engine works and we can define other triggering conditions as needed.

An important theoretical basis of the Loftis REGEN Model is that it assumes the floristic composition of a stand before a major disturbance has effects on the species composition of the stand after the disturbance. The model takes a number of plots as its input, each of which is a region with an area of either $\frac{1}{100}$ acre or $\frac{1}{250}$ acre on a stand. The input plot size can be changed. Before being submitted to the REGEN Core Engine, in each plot, the density of each individual species should be transformed from a per acre representation to a per plot-size representation. After being submitted to the REGEN Model, the species in each plot will "compete" with each other based on a ranking scheme for the opportunity to regenerate. Each species will be assigned a rank and species with a higher rank have a higher chance to win.

According to the Loftis REGEN Model, the species can regenerate from stumps, small stems, or seedlings. The ranking scheme is also based on species' size classes and some stochastic functions. A species' probability of winning the competition is proportional to its ranking. If a constant probability threshold is available, the model can generate a random real number and determine the rank of the species by comparing the number with the constant threshold. The rank can also be determined by plugging random numbers into a logistic regression model, if all the explanatory variables are known. After all species' ranks are determined, the process of selection by competition begins. Depending on whether stump

sprouts exist in a given plot or not, the ranks are used in different ways to select winning species. The model consults the corresponding management unit knowledge base for the number of individuals of the winning species to regenerate. Once the process is finished, the regeneration results of each plot will be tranformed back to a per acre representation and can be used for further simulation and analysis. For more details of the above process and other specifics of the REGEN Model, refer to [9] and [1].

A stand-alone software module, the REGEN Core Engine that implements the Loftis REGEN Model, has been developed by David Boucugnani. The REGEN Core Engine could serve as a "plug-in" for any number of custom human-computer interfaces or as a module in larger forestry applications such as NED-2 [16]. A major feature of the REGEN Core Engine is that it separates the knowledge base representing a management unit from the procedural model implementation. For different management units and stands, different knowledge bases can be loaded and used by by the Core Engine.

A management unit knowledge base contains species information needed by the REGEN Core Engine, such as rankings, probabilities, numbers of individuals to regenerate, explanatory variables for logistic regression model, and so on. The knowledge and information are organized and stored in Prolog clauses. For example, the unary predicate

```
regen_kb_name(KBID).
```

stores an atom that uniquely specifies the identification number of a knowledge base, the 4-ary predicate

```
regen_ranking(CodeA,SizeA,RankN,KBID).
```

stores the triple `<species,size,ranking>` that contains the size class and ranking of a species, and the 8-ary fact

```
regen_establishment_logistic(SourceA,CodeA,SEP1N,SEP2N,SEP3N,SEP4N,SEP5N,KBID).
```

stores the values of 5 explanatory variables used by the logistic regression model.

Being separated from the declarative knowledge stored in the knowledge base, the procedural knowledge of the REGEN Model is applied to implement the functionalities of the REGEN Core Engine. The REGEN Core Engine is divided into three sub-modules: *preserve plots, regenerate stand*, and *consolidate runs* [1]. The *preserve plots* sub-module collects all `plot/4` facts provided by the user and preserves them at a protected space on the REGEN blackboard. The *regenerate stand* sub-module then works on these `plot/4` facts to simuate the regeneration process. At last, the *consolidate run* sub-module compiles and analyzes the regeneration results, as well as calculates various statistics. For more details of the REGEN Core Engine, refer to [1].

3.3 REGEN for Excel

REGEN for Excel is a client application utilizing the REGEN Core Engine to provide a stand-alone application for modeling post-disturbance stand composition [1]. It is written in Visual Basic for Applications scripting language and can work with Microsoft Excel 97 to 2002. This application consists of the Excel user interface, the Knowledge Editor, the Stand Editor, and the Report Generator. The Knowledge Editor allows the user to enter knowledge and information about a management unit and compile them into a REGEN knowledge base file (a file with the extension "rkb") that can be recognized by the Core Engine to perform regeneration. The Stand Editor can be used to estimate the species composition of a stand by analyzing several representative plots defined in an existing knowledge base. The Report Generator creates reports for a finished regeneration process with graphs and charts relying on Microsoft Excel's graphing system.

As stated earlier, the REGEN Core Engine itself can be used as a "plug-in" and integrated into larger systems. REGEN for Excel provides a means for users to directly interact with the model implementation. In our work of integrating the REGEN Core Engine into the NED-2 system, the Excel user interface is no longer used as the REGEN Core Engine works directly with the Simulation Agent.

3.4 The NED-2 Regeneration Agent

The NED-2 Regeneration Agent, which was developed by Julian Bishop, is designed to integrate the REGEN Core Engine into NED-2 and make it work closely with the Simulation Agent to apply regeneration whenever necessary. This section describes the implementation details of the agent; the mechanism of coordinating with the Simulation Agent is covered in Chapter 4.

As other agents of the NED-2 system, the Regeneration Agent keeps watching the blackboard for requests for regeneration. When such a request is posted, the agent will first collect necessary information, including the Scenario Number, the list of recently simulated stands, and the starting year, and then search for triggering conditions in the NED-2 working database. After extensive communication with the professionals from the Forest Service, three prelimimary triggering conditions are defined in the Regeneration Agent.

1. The total basal area of a stand is larger than or equal to 50 but smaller than 60, but falls below 20 within 5 years.

2. The total basal area of a stand is larger than or equal to 20 but smaller than 50, but falls below 20 within 10 years.

3. The total basal area of a stand is below 20.

If one of these conditions is satisfied by a snapshot (representing a stand at a particular year), a triple containing the corresponding stand, the triggering snapshot, and the triggering year will be returned and used to trigger regeneration. There may be more than one stand satisfying the triggering conditions and a single stand may also satisfy the conditions at different times within a treatment plan (see Figure 4.2), but the Regeneration Agent will only return the triple representing the earliest triggering every time.

After locating the target, the following work flow of the Regeneration Agent is shown in Figure 3.1.

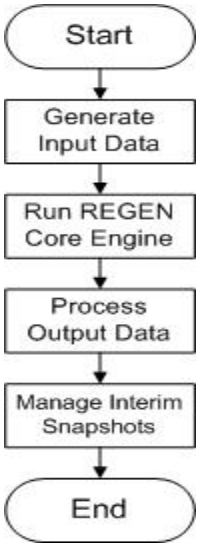

Figure 3.1: Work Flow of Regeneration

3.4.1 Input Data Generation

The first step is to generate the input data for the REGEN Core Engine, as well as other specifications needed by the Regeneration Agent. The previous and post-regeneration snapshots of the triggering snapshot are required. The previous snapshot is needed by the agent to construct the species composition before regeneration as the input to the REGEN Core Engine. An entire plan (or scenario) is always simulated without regeneration before the Regeneration Agent becomes active. Thus, there will already exist a post-regeneration snapshot with trees that have been grown during simulation. The Regeneration Agent will later modify this snapshot to represent the regeneration results. The agent will also write "LOFTIS-REGEN-TRIGGERED" to the triggering snapshot's scenario_designs_notes column in the [Scenario_designs] table so that this snapshot will not trigger regeneration again.

For simplicity the size of a REGEN input plot is set to be $\frac{1}{10}$ acre so that each cluster of a stand has exactly one regeneration plot. The REGEN Core Engine uses FIA codes to specify species and the Regeneration Agent will translate NED species codes to corresponding FIA

codes whenever necessary. Then the agent checks if this stand or any of its adjacent stands has been previously marked as a "bad" stand. A stand is marked as a "LOFTIS-REGEN-BAD-STAND" if for some reason regeneration fails on this stand given all valid input.

According to the Loftis REGEN Model, species can regenerate from stumps, small stems, or seedlings. In order to find out from what source the species of a stand should regenerate, the Regeneration Agent consults the REGEN knowledge base (`.rkb` files associated with the stand) to create three lists containing the species that should regenerate from the above three sources, respectively. Certain species' seedlings are superior sources of regeneration when compared with other species [1], such as yellow poplar, sweet birch, black cherry, and so on, which will be treated differently by the REGEN Model. A list of special species is created based on the list of the species that regenerate from seedlings by querying the database for its member species that have non-zero stem counts on the stand. Then a `regen_special_species_list/1` fact that contains the list of special species is asserted. For example, the asserted fact may be

```
regen_special_species_list([‘yellow poplar‘,‘sweet birch‘,‘black cherry‘]).
```

The `regen_special_species_list/1` fact is taken as part of the input to the REGEN Core Engine.

Next the agent creates the plot list by asserting a `plot_list/1` fact. For example, the asserted fact may be

```
plot_list(['10','9','8','7','6','5','4','3','2','1']).
```

Because the plot size is set to be the same as the cluster size, the length of the plot list is the same as the number of clusters and in this way we create one REGEN input plot per cluster.

For each plot, a `regen_plot_associated_rkb/2` fact that contains the ID of the plot and the name of its associated REGEN knowledge base is asserted. For example,

an asserted fact may be `regen_plot_associated_rkb('5','SAPP_sub_int_int').` The `regen_plot_associated_rkb/2` facts are also needed by the REGEN Core Engine.

Next the Regeneration Agent generates the input data that represent other species for the REGEN Core Engine. These species either regenerate from stumps or small stems. The input data for these species are organized by plot. Each plot will have a series of facts containing the species that regenerate from stumps, and a series of facts containing the species that regenerate from small stems.

For the species that regenerate from stumps, a list of [FIAStemCode, DBH, StemCount] triples is created. For each tree record in the triggering snapshot, its tree_stems_per value is subtracted from the corresponding value in the previous snapshot. The result is then converted to a per $\frac{1}{10}$ acre representation (by being divided by 10) and rounded to the nearest integer and is taken as the StemCount value for this record. The NED species code of each tree record is also retrieved and is translated to the corresponding FIA code, which is taken as the FIAStumpCode value for this record. After all the triples are generated, the total StemCount values of each species are added together and all non-zero DBH values are put into a list. A predicate that contains a species in a plot that regenerates from stumps may look like:

```
regen_potential_sp(RegenPlotID,StumpCount,DBHList,FIAStumpCode).
```

For the species that regenerate from small stems, a list of [FIAStemCode, SizeClass, RoundedStemCount] triples is created. Small stems means the trees that are either in the [Ground_obs] table, or in the NED-2 [Overstory_obs] or [Understory_obs] table with DBH less than 1.5. SizeClass is determined based on Tree Height or HeightClass. When Tree Height (H) is available, SizeClass is determined using the following rules:

1. $0ft < H < 2ft \Rightarrow$ SizeClass = 's' (small);
2. $2ft \leq H < 4ft \Rightarrow$ SizeClass = 'm' (medium);
3. $H \geq 4ft \Rightarrow$ SizeClass = 'l' (large).

When Tree Height is not available, HeightClass may be used instead to determine SizeClass. In this case, the agent will first try to find a mapping from the NED-2 HeightClass for a stem to the Loftis SizeClass in the facts `height_class_ned_loftis/2`. If these facts are not available, the agent will determine itself by querying the [Inventory_parameters] table. The query returns a list of Tree Height intervals as lower and upper bounds of height classes. Loftis SizeClass values are then determined based on these NED-2 HeightClass values and the mapping relations are encoded in `height_class_ned_loftis/2` facts that are asserted later. Then given the HeightClass values corresponding SizeClass values can be determined. If both Tree Height and HeightClass are missing, the SizeClass is set to be "unknown". Any tree that is classified as "l" is also wrapped in either a `large_sapling_no_dbh/4` fact or a `large_sapling_with_dbh/4` fact, depending on whether its DBH is 0 or not.

In the list of species that regenerate from small stems, each species may have more than one triple. In such case the StemCount values of these triples are added together to form a new triple for the species so that each species will have exactly one triple containing its FIA code, size class, and rounded total stems count. At last each triple of the list is wrapped in a `plot/4` fact which is asserted into the working memory as part of the input to the REGEN Core Engine. A typical `plot/4` fact may look like:

```
plot(RegenPlotID,species(FIASpeciesCode),size(SizeClass),count(StemCount)).
```

The plot motality coefficient is arbitrarily set to be 0 by asserting the fact

```
regen_plot_mortality(0).
```

Thus a complete input dataset is created, which includes:

1. A `regen_special_species_list/1` fact.
2. A `plot_list/1` fact.
3. For each input plot, a `regen_plot_associated_rkb/2` fact.

4. For each input plot, `regen_potential_sp/4` facts that represent the species that regenerate from stumps.

5. For each input plot, `plot/4` facts that represent the species that regenerate from small stems.

6. A `regen_plot_mortality(0)` fact.

A sample input dataset is listed in Appendix C.

The REGEN Core Engine is run by calling the predicate `regenerate_stand_multi/1`. Because the REGEN Core Engine implements the stochastic REGEN Model, every time it may generate different output, to get more reliable results the REGEN Core Engine is run 10 times. The reason why we run 10 times is that by adding the 10 regenerated stems count values of an individual species together, it automatically gives us a per acre representation of the result so that we don't have to do any more unit conversion again.

3.4.2 Output Data Processing

As the REGEN Core Engine applies regeneration, it generates output data and asserts facts into the working memory. When regeneration finishes, the Regeneration Agent is responsible for writing these regeneration results into the post-regeneration snapshot.

First the regenerated trees of each species of the same size class on each plot are aggregated by adding the 10 simulated stems counts together. After 10 runs of the REGEN Core Engine there are 10 `regen_plot_stat/5` facts for each spcies-size-plot combination. Such a fact looks like

```
regen_plot_stat(PlotID,Species,SizeClass,count(TotalPlotCount),NoOfRun).
```

Since the input stems count values are in per $\frac{1}{10}$ acre representation, by adding the 10 TotalPlotCount values together we can get the total stems count value in per acre representation, which is compatitable with the data requirement of the NED-2 database. After the

aggregation, each 10 `regen_plot_stat/5` facts are turned into one `regen_cluster_stat/4` fact, which may look like

```
regen_cluster_stat(ClusterID,Species,SizeClass,count(TotalClusterCount)).
```

The TotalClusterCount is taken as the TPA (Trees per Acre) increment value for this combination.

The data wrapped in `regen_cluster_stat/4` facts represent the winning candidate species and are used to update records in or add new records to the post-regeneration snapshot. These facts are processed one by one and, depending on whether the species regenerates from large saplings or not, are treated differently.

If a winning species regenerates from large saplings, the agent will first generate a list of records of this species in the previous snapshot from the NED-2 database tables and get its Observation ID's. This information can be obtained from the input predicates `large_sapling_with_dbh/4` and `large_sapling_no_dbh/4`. Then we determine which of these records that represent large saplings prior to regeneration are chosen to represent this winning species in the post-regeneration snapshot based on the regenerated TPA of this species.

If the regenerated TPA of a species is less than 10, exactly one record will be affected. One of the records is picked randomly from the list and if it had a DBH value before regeneration, the corresponding record in the originally simulated post-regeneration snapshot is located and the TPA value of this record is increased by an amount that is equal to the regenerated TPA. If the randomly picked record had no DBH value before regeneration, a new record with an appropriate Observation ID and Cluster Number will be created and added to the post-regeneration snapshot in the [Overstory_obs] table with the value of the regenerated TPA written in the `tree_stem_per` column. These records are given default DBH values of 1.5 inches, and all the other records in the list that are not chosen will be treated as losers.

If the regenerated TPA of a species is larger than 10, the regenerated TPA is divided by 10 and the number of records that will be affected is equal to the quotient plus one, because

each winner record is set to represent at most 10 trees per acre, and possibly less if it is the last one selected. Each time a random number is generated to pick a record from the list, then either a corresponding record in the originally simulated post-regeneration snapshot is updated or a new record is written to the post-regeneration snapshot. Again, all the other records in the list that are not chosen will be treated as losers.

The losers here, however, are not those species that lost in the regeneration competition, but are those that were not chosen in the above random picking process. If this species has a record in the previous snapshot in the [Overstory_obs] table, the TPA of the corresponding record in the post-regeneration snapshot is set to 0; if this species has a record in the previous snapshot in the [Understory_obs] table, all its records in the originally simulated post-regeneration snapshot are deleted.

For the species that do not regenerate from large saplings, the Regeneration Agent follows a similar procedure but the agent only adds new records to the post-regeneration snapshot in a way that each winner record is set to represent at most 10 trees per acre, and possibly less if it is the last one selected.

The Loftis REGEN Model only predicts the results 10 years after regeneration is triggered. Most of the time it is assumed that when regeneration happens, there is a previously simulated snapshot 10 years later. If for any reason there is not, the output data from regeneration will be written to the snapshot that represents the earliest year that is at least 10 years after regeneration is triggered. If there are other previously simulated snapshots between the triggering snapshot and the post-regeneration snapshot, the data in between will be erased and these interim snapshots are set to be -997 in the [Scenario_designs] table. A snapshot value of -997 indicates that regeneration was in progress during this year and no data is available.

Chapter 4

Extensions to the Simulation Agent for Regeneration

4.1 Communication Protocol

The Regeneration Agent will trigger the REGEN Core Engine to perform regeneration on a stand if the triggering conditions are satisfied. The information needed to test the triggering conditions is obtained from the NED-2 working database. Usually regeneration will take place after a clearcut treatment that is scheduled in a treatment plan. Hence after a plan that contains a clearcut treatment is simulated, the Regeneration Agent is expected to find out that the triggering conditions are satisfied and execute the REGEN Core Engine. Then after the regeneration on a stand is finished, the Simulation Agent is expected to notice that and resimulate the plan from the end of the regeneration to the end of the plan on that stand. Next the Regeneration Agent will look at the working database again to see if any other regeneration is necessary. This process is shown in Figure 4.1.

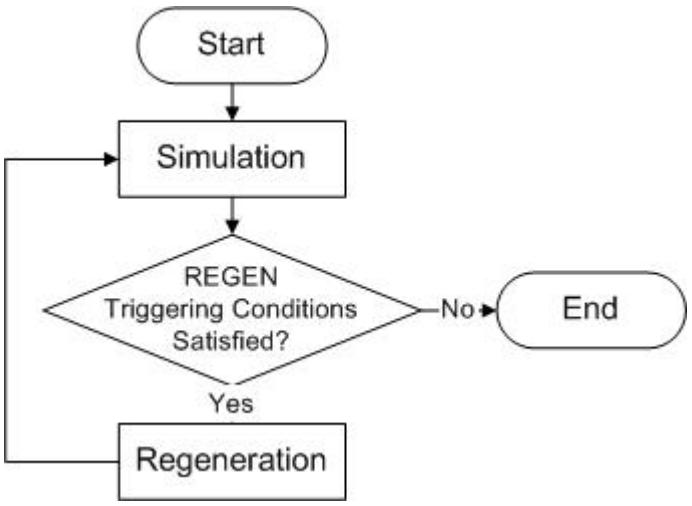

Figure 4.1: Work Flow of the Simulation and Regneration Agents

This design relies heavily on the communication and coordination between the Simulation Agent and the Regeneration Agent. The Regeneration Agent should be able to know when to look at the database to check triggering conditions, whereas the Simulation Agent should be able to know when the regeneration is over and it is expected to resimulate the stand. Moverover, after simulation the Simulation Agent will have to pass some information to the Regeneration Agent to instruct it to work on the database and after regeneration the Regeneration Agent will also need to pass some information to the Simulation Agent to tell it where to start the resimulation. To realize these functionalities a protocol was designed by Julian Bishop and me to help the two agents communicate and coordinate with each other.

The protocol contains two parts. The first part uses the existing `request/1` predicate. Suppose before the simulation of a treatment plan there is a request

```
request([fvs_plan|Rest]).
```

on ther blackboard. Then after the simulation of the plan is successfully finished, the Simulation Agent will retract the previous request and put

```
request([regenerate|Rest]).
```

on the blackboard. As the Regeneration Agent keeps watching the blackboard, it will see this request and query the database for valid triggering conditions and perform regeneration if necessary. After the regeneration is successfully finished, the Regeneration Agent will retract the precvious request and put

```
request([resimulate|Rest]).
```

on the blackboard. Then the Simulation Agent will perform appropriate resimulation on a stand. This process will continue until the Regeneration Agent can't find any more stands that require regeneration in the current working database.

The other part of the protocol takes care of the details. The simulation of a treatment plan on a management unit is performed on the stand level, that is, the stands in the unit

will be simulated one by one. Whenever a stand is simulated and the data for this stand are written to the working database, the Simulation Agent will assert a `fact/4` clause on the blackboard. For example, the fact

```
fact(recently_simulated(plan(1),stand(2)),true,system,[]).
```

says that Stand 2 was just simulated under Plan 1. After all the stands are successfully simulated under the plan, another `fact/4` clause is written on the blackboard, which may look like

```
fact(recently_simulated(plan(1),year(2001)),true,system,[]).
```

which says that the first treatment year of Plan 1 is 2001. After the initial simulation this year should be the baseline year which tells the Regeneration Agent where to search from, since this year is definitely no later than any potential regeneration. In the above example, the Regeneration Agent will take year 2001 as the starting point of its search. During later communications between the two agents, the year returned in `fact/4` may be the time when a previous regeneration finished. In such case the Regeneration Agent will search the years after this year for other potential regenerations. If regeneration is needed at the same year in multiple stands, the REGEN Core Engine will be triggered to simulate the first processed stand.

Resimulations are determined by the information placed on the blackboard by the Regeneration Agent. When the Regeneration Agent finds triggering conditions satisfied on a stand, first it will retract the corresponding fact that represents the recent simulation of that stand. Using the same example above, if triggering conditions are found satisfied on Stand 2, the fact

```
fact(recently_simulated(plan(1),stand(2)),true,system,[]).
```

will be retracted from the blackboard. Then the Regeneration Agent triggers the REGEN Core Engine to perform the regeneration. When the regeneration is finished, the Regeneration Agent will write a fact on the blackboard, which may look like

```
fact(recently_regenerated([plan(1),year(2031),stand(2)],result_snapshot(17)),
                          true,system,[]).
```

This fact tells the Simulation Agent that the regeneration on Stand 2 under Plan 1 is just finished, and the finishing year and snapshot are 2031 and 17, respectively. The finishing year and snapshot will be needed by the Simulation Agent to construct appropriate FVS input files for later resimulation on the stand. In this example, resimulation is expected to start from 2031 (but not including 2031).

With the above fact and the request `request([resimulate|Rest])` on the blackboard, the Simulation Agent will first retract the above fact and then go ahead to resimulate the plan on Stand 2 from the end of the regeneration to the end of the plan. After the resimulation is successfully finished, another fact announcing the recent resimulation for the stand will be asserted and a new list of simulated stands will be available for the Regeneration Agent to search for another potential regeneration.

4.2 Implementation of Resimulation

4.2.1 Principal Methodology

Before the Regeneration Agent is integrated into the NED-2 system, the Simulation Agent could only simulate a stand from the baseline year to the last year of a plan. According to the new design, part of the simulated data may be overwritten by regeneration and the Simulation Agent should be able to resimulate the plan after regeneration. A graph illustrating an example process is shown in Figure 4.2.

In this simplified example, a treatment plan is developed from 2001 to 2061 with six 10-year cycles and the year 2001 is the baseline year. Suppose there are two clearcut treatments scheduled in 2011 and 2041 on a stand, respectively. First the Simulation Agent will simulate the plan from the baseline to the end. After that the Regeneration Agent will find that regeneration should be triggered at 2011 and it will perform the regeneration, simulate the 10-year cycle from 2011 to 2021, and modify the original simulated data at 2021. Then the

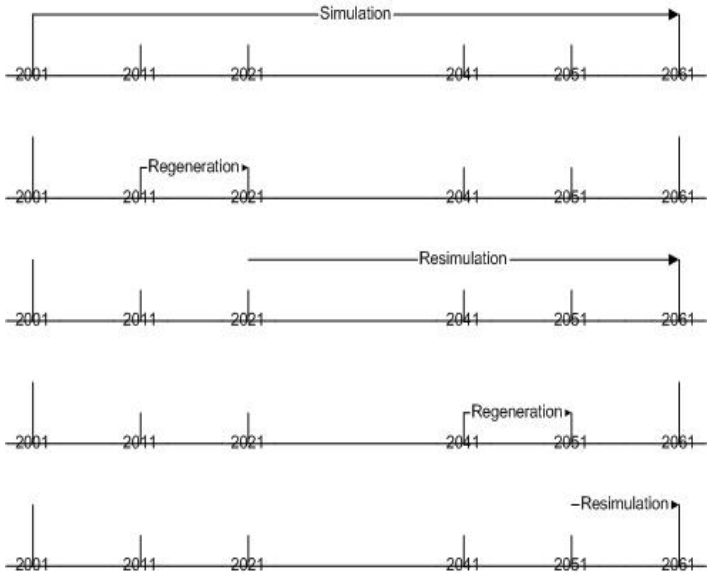

Figure 4.2: Coordination between the Simulation and Regneration Agents

Simulation Agent will resimulate the plan from 2021 to the end, and the Regeneration Agent will trigger the regeneration at 2041. At last, the Simulation Agent resimulates the plan from 2051 to the end.

The original data after the end of a previous regeneration should be erased becasue these data are invalidated. In the previous example, after the first regeneration, the simulated data after the year 2021 should be erased before resimulation. When the Simulation Agent runs a simulation, it creates new snapshots and writes simulated data to a couple of tables in the NED-2 working database. For a complete list of these tables as well as the detailed work flow of the Simulation Agent, refer to [8]. In the implementation, the key is to erase the data in all snapshots for the stand that come after the post-regeneration snapshot returned by the Regeneration Agent. The predicate `fvs_eraseOldData/3` does this job. After erasing these invalidated data, the Simulation Agent will take the ending year of regeneration as the new baseline year.

The next step is to create the input file for FVS. The new baseline year is the ending year of the previous regeneration and the data for this year is already written by the Rengeration Agent, but the treatment list generated by `fvs_getTreatment/4` always includes the new baseline year. To avoid a duplicate simulation of the new baseline year, the first treatment in the list is deleted. For example, if the new baseline year is 2031, then the treatment list given by `fvs_getTreatment/4` may look like

```
[[grow,2031,grow],['Clearcut',2031,treatment],[grow,2041,grow]]
```

The regeneration data is already written in the snapshot for the growth treatment of 2031; so the treatment list we actually need is

```
[['Clearcut',2031,treatment],[grow,2041,grow]]
```

While it is unusual that there are consecutive clearcut treatments, this approach also works for any new baseline year that has a silvicultural treatment that does not necessarily trigger another regeneration.

4.2.2 Resimulation with ESTAB

With the treatment list `mdb2fvs/6` will create the input keyword file for FVS. The ESTAB regeneration model is implemented in FVS and by default, whenever its triggering conditions are met FVS will apply regeneration. Two situations are considered here:

(1) The new baseline year has a growth treatment only.

(2) The new baseline year has another silvicultural treatment.

These two situations should be considered separately because in (1), the first treatment to resimulate is in the next treatment year, which is the usual case, but in (2), there will be a silvicultural treatment in the new baseline year. According to the original design of NED-2, there should never be a non-growth treatment in the baseline year.

In case (1), the keyword file will look exactly like an ordinary one in which each treatment scheduled has the user-defined cycle length. But in case (2), the treatment scheduled in the new baseline year has a one-year cycle and the next treatment's cycle length is equal to the original length minus one. Reflecting on the output treelist, the regenerated trees are listed in the year after the new baseline year, and actually there is not a snapshot for this year in the database. To solve this, the `fvs2mdb` module is redesigned so that whenever it detects a year in the output treelist that is not a user-defined treatment year, the treelist of this year will be ignored and no snapshot is created for this year. This procedure is implemented in the predicate `fvs2mdb_bypassRecords/2`.

Another major change to the `fvs2mdb` module is plot/cluster number handling in the treelist. In an FVS output treelist, the 7th column of a tree record is the "Plot Number"(see Figure 4.5) and this number also appears as part of the NED-2 Tree ID. According to the design of NED-2, each cluster of a stand has one plot so that the cluster number and plot number are equivalent. Specifically, the plot number in FVS starts from 1 and is always larger than its corresponding NED-2 cluster number by 1. During the regeneration process, some species in a plot may not be regenerated for some reason and in such case the plot number may disappear in the output treelist as well as in the regenerated snapshot. If FVS detects some plot number is "lost" and this number happens to be 1, all the other plot numbers will be reduced by 1 so that there will always be a plot number of 1 in the output treelist. If the lost plot is not Plot 1, no rearrangement takes place.

When `fvs2mdb` writes the tree records to the database, it rebuilds the Tree ID using the output Plot Number. In the first case with Plot Number rearrangement, `fvs2mdb` may get the wrong Plot Number and can not find the corresponding tree record in the previous snapshot. To avoid this some logic is added to recognize this situation. Specifically, when the predicate `fvs2mdb_readData/13` reads a regenerated tree record, it keeps track of the cluster numbers in both the [Overstory_obs] and the [Overstory_plots] tables. When it finds there are different numbers of clusters in the two tables and Plot 1 is missing in the [Overstory_obs] table, the

difference of the two numbers will be added to the Plot Number values in the output treelist. When `fvs2mdb_readData/13` reads a non-regenerated tree record, it will extract the plot numbers from the Tree ID values instead of using the Plot Number values in the 7th column.

4.2.3 Resimulation with REGEN

When the regeneration procedure in the REGEN Core Engine is used, the default regeneration of FVS is disabled using the keyword `NOAUTOES`. Then whenever FVS encounters a treatment that results in regeneration, it will apply regeneration and wait for the Regeneration Agent to trigger the REGEN Core Engine.

Based on the stand data prior to the regeneration triggering treatment, it calls the REGEN Core Engine to apply regeneration and writes the regenerated species to the post-regeneration snapshot which is expected to hold the tree records after the regeneration. The Regeneration Agent will also follow the commnunication protocol to return the Scenario Number, Stand Number, ending year of regeneration, and the post-regeneration snapshot to the Simulation Agent. Then the Simulation Agent can use this information to resimulate the rest of the plan after the ending year.

Chapter 5

Other Extensions to NED-2

5.1 The Integration of Other FVS Variants

Besides the Northeast(ne) and the Southern(sn) variants, some other variants of FVS were integrated into NED-2. They are:

- The Blue Mountain variant(bm)
- The Central States variant(cs)
- The East Cascades variant(ec)
- The Lake States variant(ls)
- The Northern Idaho variant(ni)
- The Inland Empire variant(ie)

Each of these variants handles a list of species in a specific geographic area. The species lists of these variants may overlap one another, but to make sure FVS works correctly on a management unit it's better to use the appropriate variant that covers all the species in the management unit.

To integrate these variants, first the corresponding variant files were added to the `fvs` folder which is under the `prolog` folder. As with the existing two variants, two batch files were created for each of these new variants. One of the batch files is a non-copy version, which runs an FVS variant when called and deletes temporary files after it finishes. The other one is a copy version, which does the same thing as the non-copy version, but it also

copies the output treelist file and moves the input keyword file to the `fvsfiles` folder where they can be examined directly outside NED-2. The simulator can run in both modes.

Some predicates were also added to the files `simulator.dcm` and `mdb2fvs.dut` to accommodate these new variants. For a complete list of these predicates, please refer to the appendix. The lists of species of these new variants can handle were also added to the knowledge base. First for each species in a list, its alpha code and English name are consulted from corresponding variant documentation, and then the English name is used to retrieve its NED code from the `NED_Plants_Master` database. For each species, its variant, alpha code and NED code are written in a `ned_alpha_species_code/3` fact.

At last, these variants were added to the model selection dialog so that the user can choose appropriate variants to simulate their plans.

5.2 Extensions to The Forest Health Agent

The Forest Health Agent implements the Forest Health System, which analyzes the health state of a management unit on the stand level and can generate reports on forest health issues. Recent extensions to the Forest Health Agent are limited to the user interface. Currenly when the user chooses a stand from the `Stand` combobox, the list of species on this stand will appear in the `Species` Listbox. Then the user can choose the species and listed symptoms to do further analysis.

5.3 Other Extensions to the Simulator

The simulator module includes `simulator.dcm`, `mdb2fvs.dut`, `fvs2mdb.dut`, the Treatment Definition Agent, and the Treatment Set Selection Agent. Besides the extensions that implement the communication protocol with the Regeneration Agent and resimulation, some other changes have been made to the simulator to either fix previously found bugs or enhance its performance.

In all earlier versions of the simulator, whenever the Simulator needed to call some routines in `NEDcalc.dll`, `NEDcalc.dll` was loaded and after the call it was closed. This overhead is removed by loading `NEDcalc.dll` only once before it is first called and freeing it after all calculations are finished. This significantly improves the speed of the simulation process.

In FVS's output treelist file, the default length of the Species Number field of a tree record is 3 columns (refer to Figure 2.5). The Species Number is the sequence number assigned to that species in the FVS variant used [3]. Most of the time a species has a 2-digit Species Number, and there is a blank column between the Alpha Species Code field and the Species Number field. In rare cases, a species may have a 3-digit Species Number and then the Alpha Species Code field and the Species Number field are recognized as one field, which will cause the simulator to fail. Changes were made to `fvs2_mdb.dut` so that when the simulator reads a treelist record, it addes one more blank column between these two fields to avoid failing because of the above case.

Chapter 6

Conclusions

This thesis documents recent works on the integration of the Loftis REGEN Model and the Stand Visualization System into NED-2 as well as other extensions. NED-2 is a goal-driven, multi-agent decision support system. Two new agents–the Regeneration Agent and the SVS Agent are developed and become new members of the NED-2 agent family. They work with other existing agents within the blackboard architecture and their integration further demonstrates the expansibility of the NED-2 system.

There are still some issues left for the Regeneration Agent to perform its full functionality. The triggering conditions currently used may not be accurate enough, and there is no effective model built in to apply the DBH distribution to regenerated trees. The Loftis REGEN Model is the first regeneration model integrated into NED-2. In the future additional regeneration models may be integrated so that the user can choose the most appropriate one to perform analysis.

Bibliography

[1] Boucugnani, D.A. (2004) *REGEN Core Engine: A Modular and Generalized Forest Regeneration Agent.* Master Thesis.

[2] Crookston, N.L. (1999) *Using the Forest Vegetation Simulator's SVS Keyword and the Stand Visualization System.* RMRS-Moscow, September 27, 1999.

[3] Dixon, G.E. (2002) *Essential FVS: A Users Guide to the Forest Vegetation Simulator.* Forest Management Service Center, USDA Forest Service, Fort Collins, CO.

[4] Dey, D.C., et al. (1996) *A Comprehensive Ozark Regeneration Simulator.* Gen. Tech. Rep. NC-180. USDA Forest Service, North Central Forest Experiment Station, St. Paul, MN. 35 p.

[5] Van Dyck, M.G. (2002) *Keyword Reference Guide for the Forest Vegetation Simulator.* Forest Management Service Center, USDA Forest Service, Fort Collins, CO.

[6] Ernst, R.L. (1987) *Growth and Yield Following Thinning in Mixed-species Allegheny Hardwoods Stands.* In: Nyland, Ralph D., ed. Managing northern hardwoods: Proc. Symp. 1986 June 23-25. Syracuse, NY. Syracuse, NY: State Univ. New York Fac. For. Misc. Publ. 13: p. 211-222.

[7] Ferguson, D.E. and Crookston, N.L. (1991) *Users Guide to Version 2 of the Regeneration Establishment Model: Part of the Prognosis Model.* USDA Forest Service, Intermountain Research Station, Ogden, UT.

[8] Glende, A. (2004) *The NED Forest Management DSS: The Integration of Growth And Yield Models.* Master Thesis.

[9] Loftis, D.L. (1990) *Regeneration of Southern Hardwoods: Some Ecological Concepts.* In Proceedings: the National Silviculturea l Workshop, July 10-13, 1989, Petersburg, AL., Washington, D.C.: USDA Forest Service, pp. 139-143.

[10] Maier, F., et al. (2002) *PROLOG/RDBMS Integration in the NED Intelligent Information System.* In Meersman, Tari et al. (Eds.), Confederated International Conferences CoopIS, DOA, and ODBASE 2002 Proceedings, Irvine, California, p.528, October, 2002.

[11] Maier, F. (2002) *Notes on a Blackboard: Recent work on NED2.* Master Thesis.

[12] Maier, F., et al. (2003) *Efficient Integration of PROLOG and Relational Databases in the NED Intelligent Information System.* In Proceedings: the 2003 International Conference on Information and Knowledge Engineering (IKE'03), pp. 364-369, June 23-26, 2003, Las Vegas, Nevada, USA.

[13] McGaughey, R.J. (2004) *Stand Visualization System, Version 3.3.* USDA Forest Service, Pacific Northwest Research Station, Portland, OR.

[14] Nute, D., et al. (2003) *NED-2: An Agent-Based Decision Support System for Forest Ecosystem Management.* Environmental Modelling and Software, Vol. 14, No. 5, 2003, Special Issue on Binding Environmental Sciences and Artificial Intelligence.

[15] Nute, D., et al. (2003) *An Agent Architecture for an Integrated Forest Ecosystem Management Decision Support System.* IUFRO 2003.

[16] Nute, D., et al. (2004) *Adding New Agents and Models to the NED-2 Forest Management System.* COMPAG 2004.

[17] Ritchie, M.W. and Powers, R.F. (1993) *A User's Guide for SYSTUM-1 (Version 2.0): Simulating Trends in Young Stands Under Management in California and Oregon.* General Technical Report PSW-GTR-147. Albany CA: Pacific Southwest Research Station, USDA Forest Service. 45 p.

[18] Routh, C. (2004) *The NED-2 Forest Ecosystem Management DSS: The Integration of Wildlife Risk and GIS Agents.* Master Thesis.

[19] Twery, M., et al. (2003) *Adding New Agents and Models to the NED-2 Forest Management System.* IUFRO 2003.

[20] Twery, M., et al. (2004) *NED-2: A Decision Support System for Integrated Forest Ecosystem Management.* COMPAG 2004.

[21] Witzig, S. (2004) *The NED-2 Forest Ecosystem Management DSS: The Integration of A Wildlife Model For The Southeast, The Goal Analysis Agent And The Report Generation Agent.* Master Thesis.

Appendix A

An Index of New Predicates

The following is a list of all predicates newly defined to integrate the Loftis REGEN Model and the Stand Visualization System. For each file the predicates are sorted in the order as they appeared in the file.

A.1 regen.dcm

DCM/1

dcm(regen)

```
+AgentName                <atom>
```

Comment: This initiates the Regeneration Agent.

GET_REGEN_STANDS_AND_FIRST_TRIGGER/6

get_regen_stands_and_first_trigger(ScenarioID,SimStandList,StartYear,
StandsToRegenList,SnapshotIDFirstTrigger,YearFirstTrigger)

```
+ScenarioID               <integer>
+SimStandList             <list>
+StartYear                <integer>
-StandsToRegenList        <list>
-SnapshotIDFirstTrigger   <integer>
-YearFirstTrigger         <integer>
```

Comment: This returns the stands that need regeneration, the snapshot of the first triggering, and the year of the first triggering.

GET_PRE_TRIGGER/4

get_pre_trigger(Scenario,StandTrigger,SnapshotTrigger,SnapshotPreTrigger)

```
+Scenario                <integer>
+StandTrigger            <integer>
+SnapshotTrigger         <integer>
-SnapshotPreTrigger      <integer>
```

Comment: This returns the immediate previous snapshot of the triggering snapshot.

GET_POST_TRIGGER/4

get_post_trigger(Scenario,StandTrigger,SnapshotTrigger,SnapshotPostTrigger)

```
+Scenario                <integer>
+StandTrigger            <integer>
+SnapshotTrigger         <integer>
-SnapshotPostTrigger     <integer>
```

Comment: This returns the snapshot in which regeneration results should be written.

MARK_TRIGGERING_SNAPSHOT/1

mark_triggering_snapshot(SnapshotTrigger)

```
+SnapshotTrigger         <integer>
```

Comment: This writes a string to the triggering snapshot's `scenario_designs_notes` field in the [Scenario_designs] table to indicate that this snapshot has already triggered regeneration.

SCRUB_ANY_INTERIM_SNAPSHOT/4

scrub_any_interim_snapshot(Scenario,StandTrigger,SnapshotTrigger, SnapshotPostTrigger)

```
+Scenario                <integer>
+StandTrigger            <integer>
+SnapshotTrigger         <integer>
+SnapshotPostTrigger     <integer>
```

Comment: This resets all the snapshots between the triggering snapshot and the post-regeneration snapshot to be -997.

REGEN/6

regen(Scenario,Stand,SnapshotPreTrigger,SnapshotTrigger,SnapshotPostTrigger ,TrtYear)

```
+Scenario                <integer>
+Stand                   <integer>
+SnapshotPreTrigger      <integer>
+SnapshotTrigger         <integer>
+SnapshotPostTrigger     <integer>
+TrtYear                 <integer>
```

Comment: This is the core predicate that prepares input data to the REGEN Core Engine, calls the REGEN Core Engine, and processes the output data.

REGEN_CLEANUP/0

regen_cleanup

Comment: This retracts temporary predicates in the memory after running regeneration.

WRITE_REGENERATED_SNAPSHOT/1

write_regenerated_snapshot(SnapshotPostTrigger)

```
+SnapshotPostTrigger      <integer>
```

Comment: This writes regenerated trees to the post-regeneration snapshot.

APPLY_DBH_DISTRIBUTION/0

apply_dbh_distribution

Comment: This assigns a DBH value to each regenerated tree based on a specific DBH distribution.

PROCESS_WINNER_STAT/5

process_winner_stat(SnapshotPostTrigger,ClusterID,FIASpeciesCode, OriginalSize,TPA)

```
+SnapshotPostTrigger      <integer>
+ClusterID                <integer>
+FIASpeciesCode           <string>
+OriginalSize             <atom>
+TPA                      <number>
```

Comment: This chooses trees from the winning ones and writes them to the post-regeneration snapshot.

PROCESS_LSAP_WINNER_STAT/8

process_lsap_winner_stat(SnapshotPostTrigger,ClusterID,FIASpeciesCode, LSapDBHList,LSapNoDBHList,LSapList,SapCount,TPA)

```
+SnapshotPostTrigger      <integer>
+ClusterID                <integer>
+FIASpeciesCode           <string>
+LSapDBHList              <list>
+LSapNoDBHList            <list>
+LSapList                 <list>
+SapCount                 <integer>
+TPA                      <number>
```

Comment: This determines which trees that represented from large saplings before regeneration are chosen to as winners.

UPDATE_TPA/4

update_tpa(SnapshotPostTrigger,ClusterID,ObsID,TPAIncrement)

```
+SnapshotPostTrigger      <integer>
+ClusterID                <integer>
+ObsID                    <integer>
+TPAIncrement             <number>
```

Comment: This updates the TPA value of a record which is also a winner of regeneration.

REGEN_ADD_TREE_RECORDS/5

regen_add_tree_records(SnapshotPostTrigger,ClusterID,FIASpeciesCode, OriginalSize,TPA)

```
+SnapshotPostTrigger      <integer>
+ClusterID                <integer>
+FIASpeciesCode           <string>
+OriginalSize             <atom>
+TPA                      <number>
```

Comment: This adds new records representing winners to database.

PROCESS_LSAP_LOSERS/3

process_lsap_losers(SnapshotPostTrigger,ClusterID,LosersList)

```
+SnapshotPostTrigger      <integer>
+ClusterID                <integer>
+LosersList               <list>
```

Comment: This processes trees that are not selected to represent winners of regeneration.

SUCCEED/1

succeed(P)

```
+P                          <any>
```

Comment: This always succeeds having attempted to prove P.

AGGREGATE_PLOT_STATS/2

aggregate_plot_stats(ClusterList,PlotList)

```
+ClusterList                <list>
+PlotList                   <list>
```

Comment: This sums the counts of winning species of all regenerations runs by Cluster, Species, and OriginalSize.

PROCESS_CLUSTERS/6

process_clusters(ClusterList,PlotList,NEDStumpSpecies,NEDSmallSpecies,

SnapshotPreTrigger,SnapshotTriggerStr)

```
+ClusterList                <list>
+PlotList                   <list>
+NEDStumpSpecies            <list>
+NEDSmallSpecies            <list>
+SnapshotPreTrigger         <integer>
+SnapshotTriggerStr         <string>
```

Comment: This prepares input data to the REGEN Core Engine.

STORE_LARGE_SAPLINGS/6

store_large_saplings(NEDClusterID,DBTableTrigger,ObsID,FIAStemCode,

SizeClass,DBH,StemCount)

```
+NEDClusterID            <integer>
+DBTableTrigger          <string>
+ObsID                   <integer>
+FIAStemCode             <string>
+SizeClass               <atom>
+DBH                     <number>
+StemCount               <number>
```

Comment: This asserts a predicate representing each species that regenerate from large saplings either with or without a DHH value.

DEAL_SMALL_STEMS/2

deal_small_stems(AggregatedListOfSmallStems,PlotIDToReceiveSmallStems)

```
+AggregatedListOfSmallStems  <list>
+PlotIDToReceiveSmallStems   <integer>
```

Comment: This assigns a plot number to each aggregated species that regenerates from small stems.

ALL_SIZE_KNOWN/1

all_size_known(SmallStemsList)

```
+SmallStemsList          <list>
```

Comment: This checks whether all the species that regenerate from small species have their size class known.

NO_BAD_STANDS/4

no_bad_stands(ScenarioStr,StandStr,AdjacentStandsStr,TrtYearStr)

```
+ScenarioStr             <string>
+StandStr                <string>
+AdjacentStandsStr       <string>
+TrtYearStr              <string>
```

Comment: This checks if a stand or its adjacent stands is/are previously marked as a bad stand on which the REGEN Core Engine failed.

CHECK_BAD_STANDS_LIST/1

check_bad_stands_list(BadStandList)

```
+BadStandList              <list>
```

Comment: This asserts the list of bad stands.

CLASSIFY_SIZE/6

classify_size(+Height,+HeightSource,+HeightClass,+HeightClassSource,+DBH, -LoftisSizeClass)

```
+Height                    <number>
+HeightSource              <integer>
+HeightClass               <atom>
+HeightClassSource         <integer>
+DBH                       <number>
-LoftisSizeClass           <atom>
```

Comment: This returns the Loftis Size Class of a species given its height or height source value.

MAKE_HEIGHT_CLASS_NED_LOFTIS/2

make_height_class_ned_loftis(BoundsList,NEDHeightClass)

```
+BoundsList                <list>
-NEDHeightClass            <integer>
```

Comment: This returns a NED Height Class value for each lower-upper bounds pair of the bounds list.

AGGREGATE_SMALL_STEMS/2

aggregate_small_stems(SmallStemsList,AggregatedList)

```
+SmallStemsList          <list>
-AggregatedList          <list>
```

Comment: This returns a list of species that regenerate from small stems with their stems counts aggregated by name and size.

DEAL_STUMPS/2

deal_stumps(CutTreeList,PlotIDList)

```
+CutTreeList             <list>
+PlotIDList              <list>
```

Comment: This processes the list of species that regenerate from stumps as input to the REGEN Core Engine.

ROUND_STUMPS/2

round_stumps(StemCount,RoundedCount)

```
+Stemcount               <number>
+RoundedCount            <integer>
```

Comment: This rounds a given stem count to its nearest integer.

GENERATE_PLOT_IDS/3

generate_plot_ids(MaxPlotNumber,ListOfPlotNumbers)

```
+MaxPlotNumber           <integer>
-ListOfPlotNumbers       <list>
```

Comment: This generates a list of plot numbers in descending order given a maximum plot number.

LIST_TO_STRING_WITH_QUOTES/2

list_to_string_with_quotes(List,StringOfListWithQuotes)

```
+List                    <list>
-StringOfListWithQuotes   <string>
```

Comment: This turns a given list into a string with each of its elements surrounded with single quotes.

LIST_TO_STRING/2

list_to_string(List,StringOfList)

```
+List                    <list>
-StringOfList            <string>
```

Comment: This turns a given list into a string.

SQUARE_BRACKETS_TO_ROUND/2

square_brackets_to_round(BrackStr,ParenStr)

```
+BrackStr                <string>
+ParenStr                <string>
```

Comment: This turns a string surrounded by brakets into one surrounded by parentheses.

A.2 SVS.DCM

DCM/1

dcm(svs)

```
+AgentName                  <atom>
```

Comment: This initiates the SVS Agent.

SVS_SEL/0

svs_sel

Comment: This constructs the user interface of the SVS Agent.

SVS_INIT/0

svs_init

Comment: This obtains the stands list and fills the list into the list box of the GUI.

SVS_SEL_GETSTANDS/1

svs_sel_getStands(StandList)

```
-StandList                  <list>
```

Comment: This returns the list of stands in the current management unit.

SVS_SEL_GETYEARS/1

svs_sel_getYears(YearList)

```
-YearList                  <list>
```

Comment: This returns the list of currently existing treatment years scheduled on a stand.

SVS_SEL_GETPLANS/1

svs_sel_getPlans(PlanList)

```
-PlanList                <list>
```

Comment: This returns the list of treatment plans developed on a stand.

SVS_SEL_TOSTRING/2

svs_sel_toString(List1,List2)

```
+List1                   <list>
-List2                   <list>
```

Comment: This turns each element of List 1 into a string.

SVS_SEL_FILLBOX/4

svs_sel_fillBox(Window,Length,StartingNumber,List)

```
+Window                  <window_handle>
+Length                  <integer>
+StartingNumber          <integer>
+List                    <list>
```

Comment: This fills a list of items in a list box in the GUI.

SVS_SEL_CLEANCOMB/1

svs_sel_cleanComb(Window)

```
+Window                  <window_handle>
```

Comment: This clears existing items in a combo box in the GUI.

SVS_STARTDISPLAY/0

svs_startDisplay

Comment: This pops up the GUI when the SVS Agent is called and does initializations.

SVS_SEL_DISPLAYTREATMENT/1

svs_sel_displayTreatment(StandStr)

```
+StandStr                <string>
```

Comment: This sets up the SVS Agent to display a stand by plan.

SVS_SEL_DISPLAYYEAR/0

svs_sel_displayYear

Comment: This sets up the SVS Agent to display a stand by year.

SVS_DISPLAY_STAND/4

svs_display_stand(Variant,Stand,ScenarioS,Dummy)

```
+Variant                 <atom>
+Stand                   <integer>
+ScenarioS               <string>
+Dummy                   <any>
```

svs_display_stand(Variant,Stand,Year,Dummy)

```
+Variant                 <atom>
+Stand                   <integer>
+Year                    <integer>
+Dummy                   <any>
```

Comment: This displays a stand either by plan or by year.

SVS_BULLETPROOF/2

svs_bulletProof(Stand,Scenario)

```
+Stand                  <integer>
+Scenario               <integer>
```

svs_bulletProof(Stand,Year)

```
+Stand                  <integer>
+Year                   <integer>
```

Comment: This makes sure the chosen plan or year has been simulated.

SVS_GETRECORDSBYYEAR/3

svs_getRecordsByYear(Stand,Year,Records)

```
+Stand                  <integer>
+Year                   <integer>
-Records                <list>
```

Comment: This collects records for displaying by year.

SVS_GETSNAPSHOT/3

svs_getSnapshot(Stand,Year,ScenarioList,SnapshotPairsList)

```
+Stand                  <integer>
+Year                   <integer>
+ScenarioList           <list>
-SnapshotPairsList      <list>
```

Comment: This returns a list of (PreSnapshot,Snapshot) pairs for displaying by year.

SVS_GET_RECORDS_FOR_STAND_AND_SCENARIO/3

svs_get_records_for_stand_and_scenario(Stand,Scenario,Records)

```
+Stand                  <integer>
+Scenario               <integer>
-Records                <list>
```

Comment: This gets treelist records for displaying by plan.

SVS_FINDSNAPSHOT/4

svs_findSnapshot(Stand,Scenario,YearsList,SnapshotPairsList)

```
+Stand                  <integer>
+Scenario               <integer>
+YearsList              <list>
-SnapshotPairsList      <list>
```

Comment: This returns a list of (PreSnapshot,Snapshot) pairs for displaying by plan.

SVS_GETTRTRECORDS/2

svs_getTrtRecords(SnapshotPairsList,Records)

```
+SnapshotPairsList      <list>
-Records                <list>
```

Comment: This returns treelist records given a list of (PreSnapshot,Snapshot) pairs for displaying by plan.

SVS_GETTRTRECORDSBYYEAR/2

svs_getTrtRecordsByYear(SnapshotPairsList,Records)

```
+SnapshotPairsList      <list>
-Records                <list>
```

Comment: This returns treelist records given a list of (PreSnapshot,Snapshot) pairs for displaying by year.

SVS_GETOVERRECORDS/4

svs_getOverRecords(Symbol,Snapshot1,Snapshot2,OverRecords)

```
+Symbol                 <atom>
+Snapshot1              <integer>
+Snapshot2              <integer>
-OverRecords            <list>
```

Comment: This returns treelist records from the [Overstory_obs] table.

SVS_GETUNDERRECORDS/4

svs_getUnderRecords(Symbol,Snapshot1,Snapshot2,OverRecords)

```
+Symbol                 <atom>
+Snapshot1              <integer>
+Snapshot2              <integer>
-OverRecords            <list>
```

Comment: This returns treelist records from the [Understory_obs] table.

SVS_GETVARIANT/4

svs_getVariant(VariantA,Variant)

```
+VariantA               <atom>
-Variant                <atom>
```

Comment: This adds single quotes to a given variant name atom.

SVS_WRITETREELIST/3

svs_writeTreeList(Variant,Stand,Records)

```
+Variant                <atom>
+Stand                  <integer>
+Records                <list>
```

Comment: This predicate writes the given tree records to an FVS treelist file for displaying by plan.

SVS_WRITETREELIST/4

svs_writeTreeList(Variant,Year,Stand,Records)

```
+Variant                <atom>
+Year                   <integer>
+Stand                  <integer>
+Records                <list>
```

Comment: This writes given tree records to an FVS treelist file for displaying by year.

SVS_WRITEPREP/3

svs_writePrep(Stand,FVSFile)

```
+Stand                  <integer>
-FVSFile                <atom>
```

Comment: This creates a folder for temporary FVS and SVS files.

SVS_CREATEIOFILENAME/3

svs_createIOFileName(Stand,FVSfile,SVSfile)

```
+Stand                  <integer>
-FVSFile                <atom>
-SVSFile                <atom>
```

Comment: This creates FVS and SVS file names.

SVS_WRITETREELISTFILE/5

svs_writeTreeListFile(Variant,Stand,Cycle,TreeList,Text)

```
+Variant                <atom>
+Stand                  <integer>
+Cycle                  <integer>
+TreeList               <list>
-Text                   <string>
```

Comment: This generates text to be written to FVS treelist files for displaying by plan.

SVS_WRITETREELISTFILE/6

svs_writeTreeListFile(Variant,Year,Stand,PlanNumber,TreeList,Text)

```
+Variant                <atom>
+Year                   <integer>
+Stand                  <integer>
+PlanNumber             <integer>
+TreeList               <list>
-Text                   <string>
```

Comment: This generates text to be written to FVS treelist files for displaying by year.

SVS_PARSETEXT/3

svs_parseText(LisfOfStrings,Stack,String)

```
+ListOfStrings         <list>
+Stack                  <string>
-String                 <string>
```

Comment: This connects string items of a list together.

SVS_WRITETREELISTFILE_A/3

svs_writeTreeListFile_A(Variant,Stand,Cycle,Year,Lenght,HeaderText)

```
+Variant                <atom>
+Stand                  <number>
+Cycle                  <integer>
+Year                   <integer>
+Length                 <integer>
-HeaderText             <string>
```

Comment: This predicate generates header text of an FVS treelist file for displaying by year.

SVS_WRITETREELISTFILE_B/4

svs_writeTreeListFile_B(Variant,TreeList,TempText,Text)

```
+Variant                <atom>
+TreeList               <list>
-TempText               <sring>
-Text                   <string>
```

Comment: This generates text of the given list of tree records.

SVS_WRITETREELISTFILE_C/4

svs_writeTreeListFile_C(Variant,Stand,PlanNumber,Year,Length,HeaderText)

```
+Variant                <atom>
+Stand                  <integer>
+PlanNumber             <integer>
+Year                   <integer>
+Length                 <integer>
-HeaderText             <string>
```

Comment: This generates header text of an FVS treelist file for displaying by year.

SVS_START_SVS/4

svs_start_SVS(FVSFile,SVSFile)

```
+FVSFile                <atom>
+SVSFile                <atom>
```

Comment: This calls FVS2SVS and SVS to generate images of a stand.

A.3 SIMULATOR.DCM

FVS_ERASEOLDDATA/3

fvs_eraseOldData(Scenario,Stand,Snapshot)

```
+Scenario                <integer>
+Stand                   <integer>
+Snapshot                <integer>
```

Comment: This erases previously simulated data after the ending point of a regeneration.

FVS_ERASEREGENNOTES/1

fvs_eraseRegenNotes(ListOfScenario)

```
+ListOfScenario          <list>
```

Comment: This erases the notes inserted by REGEN in previous simulation.

A.4 FVS2MDB.DCM

FVS2MDB_BYPASSRECORDS/2

fvs_bypassRecords(NumberOfRecords,StartingNumber)

```
+NumberOfRecords         <integer>
+StartingNumber          <integer>
```

Comment: This bypasses tree records that do not have a snapshot in the [Scenario_designs] table when reading an FVS treelist file.

Appendix B

An Index of Modified Predicates

The following is a list of existing predicates that are modified since July 2004. For detailed specifications of these predicates please refer to[8].

B.1 simulator.dcm

DCM/1

FVS_COMMAND/4

FVS_COPYDATA/5

FVS_RUNNEDCALC/3

FVS_SIMULATE/2

FVS_SIMULATEPLANS/4

B.2 mdb2fvs.dcm

MDB2FVS_GETBAFSIZE/6

MDB2FVS_GETHABITAT/3

B.3 FVS2MDB.DCM

FVS2MDB_READDATA/13

FVS2MDB_READFILE/19

FVS2MDB_WRITEDATA/15

B.4 FOREST_HEALTH.DCM

COMPLETE_SCREEN/1

Appendix C

Sample Input Dataset for the Regen Core Engine

```
% PLOT 10

regen_plot_associated_rkb('10','SAPP_sub_int_int').

plot('10',species('693'),size('s'),count(4)).
plot('10',species('693'),size('m'),count(2)).
regen_potential_sp('10',1,[],'693').

plot('10',species('832'),size('s'),count(3)).
plot('10',species('832'),size('m'),count(3)).
plot('10',species('832'),size('l'),count(1)).

plot('10',species('833'),size('s'),count(3)).
plot('10',species('833'),size('m'),count(2)).

plot('10',species('316'),size('s'),count(12)).
plot('10',species('316'),size('m'),count(2)).
plot('10',species('316'),size('l'),count(1)).



% PLOT 9

regen_plot_associated_rkb('9','SAPP_sub_int_int').

plot('9',species('837'),size('m'),count(3)).
plot('9',species('491'),size('l'),count(1)).
regen_potential_sp('9',1,[],'491').

plot('9',species('833'),size('s'),count(3)).
plot('9',species('833'),size('m'),count(1)).

plot('9',species('316'),size('s'),count(13)).
plot('9',species('316'),size('m'),count(2)).
```

```
regen_potential_sp('9',1,[],'316').

plot('9',species('409'),size('s'),count(4)).
plot('9',species('409'),size('m'),count(3)).
regen_potential_sp('9',1,[],'409').



% PLOT 8

regen_plot_associated_rkb('8','SAPP_sub_int_int').

plot('8',species('316'),size('s'),count(5)).
plot('8',species('316'),size('m'),count(2)).
plot('8',species('316'),size('l'),count(3)).

plot('8',species('711'),size('s'),count(2)).
regen_potential_sp('8',2,[],'711').

plot('8',species('833'),size('s'),count(4)).
plot('8',species('833'),size('m'),count(2)).
regen_potential_sp('8',1,[11.2],'833').

plot('8',species('541'),size('m'),count(2)).

plot('8',species('802'),size('s'),count(9)).
plot('8',species('802'),size('m'),count(2)).



% PLOT 7

regen_plot_associated_rkb('7','SAPP_sub_int_int').

plot('7',species('693'),size('s'),count(4)).

plot('7',species('832'),size('s'),count(3)).
plot('7',species('832'),size('m'),count(4)).

regen_potential_sp('7',2,[],'711').

plot('7',species('316'),size('s'),count(12)).
plot('7',species('316'),size('m'),count(3)).
regen_potential_sp('7',1,[],'316').
```

```
% PLOT 6

regen_plot_associated_rkb('6','SAPP_sub_int_int').

plot('6',species('837'),size('s'),count(3)).
plot('6',species('837'),size('m'),count(2)).

plot('6',species('409'),size('s'),count(3)).

plot('6',species('833'),size('m'),count(2)).

plot('6',species('316'),size('s'),count(23)).
plot('6',species('316'),size('m'),count(2)).
regen_potential_sp('6',1,[],'316').

plot('6',species('711'),size('m'),count(3)).

plot('6',species('372'),size('s'),count(1)).

plot('6',species('832'),size('s'),count(2)).
plot('6',species('832'),size('m'),count(3)).

plot('6',species('491'),size('m'),count(3)).
regen_potential_sp('6',2,[],'491').



% PLOT 5

regen_plot_associated_rkb('5','SAPP_sub_int_int').

plot('5',species('837'),size('s'),count(3)).
plot('5',species('837'),size('m'),count(1)).

plot('5',species('260'),size('s'),count(2)).

plot('5',species('901'),size('s'),count(1)).

plot('5',species('409'),size('s'),count(4)).
plot('5',species('409'),size('m'),count(2)).
```

```
% PLOT 4

regen_plot_associated_rkb('4','SAPP_sub_int_int').

plot('4',species('901'),size('m'),count(1)).

plot('4',species('409'),size('s'),count(4)).
plot('4',species('409'),size('m'),count(5)).

plot('4',species('491'),size('s'),count(6)).
plot('4',species('491'),size('l'),count(1)).
regen_potential_sp('4',1,[],'491').

plot('4',species('833'),size('s'),count(4)).
plot('4',species('833'),size('l'),count(1)).



% PLOT 3

regen_plot_associated_rkb('3','SAPP_sub_int_int').

plot('3',species('833'),size('m'),count(2)).
plot('3',species('833'),size('l'),count(2)).

plot('3',species('711'),size('s'),count(3)).
plot('3',species('711'),size('m'),count(1)).
regen_potential_sp('3',3,[],'711').

plot('3',species('541'),size('s'),count(3)).
plot('3',species('541'),size('m'),count(2)).

plot('3',species('580'),size('s'),count(3)).
plot('3',species('580'),size('m'),count(4)).
regen_potential_sp('3',1,[],'580').

regen_potential_sp('3',2,[],'316').



% PLOT 2

regen_plot_associated_rkb('2','SAPP_sub_int_int').

plot('2',species('651'),size('m'),count(1)).
```

```
plot('2',species('409'),size('s'),count(3)).

plot('2',species('316'),size('s'),count(4)).
regen_potential_sp('2',2,[],'316').

plot('2',species('491'),size('s'),count(2)).



% PLOT 1

regen_plot_associated_rkb('1','SAPP_sub_int_int').

plot('1',species('693'),size('s'),count(3)).
plot('1',species('693'),size('m'),count(2)).
regen_potential_sp('1',2,[2,3],'693').

plot('1',species('832'),size('s'),count(4)).
plot('1',species('832'),size('m'),count(1)).

plot('1',species('409'),size('s'),count(6)).
plot('1',species('409'),size('m'),count(2)).
plot('1',species('409'),size('l'),count(2)).



% PLOT LIST

plot_list(['10','9','8','7','6','5','4','3','2','1']).



% SPECIAL SPECIES

regen_special_species_list([‘yellow poplar‘,‘sweet birch‘,‘black cherry‘]).



% MORTALITY FOR ALL PLOTS IN STAND

regen_plot_mortality(0.3).
```