Available online at www.sciencedirect.com
SCIENCE DIRECT®
Environmental Modelling & Software 20 (2005) 1081–1100
Environmental Modelling & Software
www.elsevier.com/locate/envsoft

Eclpss: a Java-based framework for parallel ecosystem simulation and modeling

Elaine Wenderholm

Computer Science Department, State University of New York at Oswego, Oswego, NY 13126 USA

Received 30 May 2003; received in revised form 10 May 2004; accepted 7 June 2004

Abstract

Eclpss (Ecological Component Library for Parallel Spatial Simulation) is a Java™-based framework designed to give ecologists the ability to easily develop grid-based ecosystem simulations at multiple spatial and temporal scales. The framework automatically targets the model to shared memory parallel machines. Because of the judicious use of Java, both the framework and framework-generated models are platform independent. Users may write arbitrarily complex models without the need to be expert programmers. These models are reusable, easily modifiable and extensible. Collaborative model development, sharing, and dissemination with automatically-generated documentation are all web-accessible. The modelling environment consists of a suite of GUI-based tools which are designed to be intuitive to ecologists. Ecologists specify the model; the Eclpss compiler uses these specifications to generate code. Scientific unit measurements are incorporated into specifications and consistency checking is performed; substance consistency is supported. This paper presents the structure and features of the Eclpss framework, the migration of a Matlab model into this framework, and concludes with a discussion of ongoing and planned future work.

Keywords: Spatial simulation framework; Java; Shared-memory parallel; Platform independence; Units; XML

1. Introduction

As ecological modelling becomes increasingly more comprehensive and complex, it becomes more difficult to reuse some or all pieces of a model.

In the ecological domain, there is the desire to both parallelize and reuse the code from several disparate programs whose commonality might only be that they are written in the same programming language. Accomplishing this may be viewed, in part, as generating code (either manually or automatically) which "glues" together different programs and, optionally, mapping the program to a parallel architecture. This typically poses more technical difficulties than the Fortran "dusty deck" problem of the late 1980s[1] which parallelizes just one program. Since each ecological model is a different program, the initial code design can render it inflexible to incorporation into larger model(s). This also is difficult to automate since general code is hard to analyze.

Eclpss takes a different approach to model design and building: disparate models are not combined; new Eclpss models are developed; these Eclpss models may then be combined.

Eclpss models are component-based (He et al., 2002). An Eclpss Component contains pieces of user-written

[1] Parallelizing compilers analyze sequential code and, using various loop transformations, automatically restructure it into parallel implementations. (See, for example, Allen and Kennedy (1987) and Ruhl and Annaratone (1990)).

E-mail address: wender@cs.oswego.edu (E. Wenderholm).
URL: http://www.cs.oswego.edu/~wender.

1364-8152/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.envsoft.2004.06.006

code. The framework imposes no restriction on the complexity of the code itself, but only on the component interface. Components are independent and may not directly reference other components; interaction is indirect via updates to state variables. An Eclpss model may be viewed as a circuit: components are the "chips"; state variables are the "wires" that connect components. This design allows components (and the models which use them) to be freely shared, interoperable, and interchangeable. Eclpss supports top–down and bottom–up design, debugging, and experimentation. Ecologists can easily rearrange and experiment with model structure, grids, grid cell size and scale.

A programming variable in scientific programs not only has a storage type, but often has a unit of scientific measurement. Different ecological models may (correctly) use different measurement units, different measurement systems, or both, for the same programming variable in different parts of the program. Models with multi-ecological media are typical of the use of different measurement (and hence modelling) units: the density in air (of, say, a nitrogen compound) may be measured in kg/m^3; the same compound in soil surface in mg/cm^3. Unit (and data) conversion at the ecological interface is necessary. Poorly-defined programming interfaces, such as the Mars Climate Orbiter, have lead to the incorrect use of different measurement systems. Since compilers for programming languages only perform storage type-checking, measurement unit type and consistency verification often requires that checking be done by hand.

The manner in which Eclpss models are developed has many of the same characteristics as specialized programming tools.

Specialized programming tools in other application areas (such as spreadsheets, relational database management systems and symbolic mathematics programs) have allowed a community of users to write applications that previously required specialist programmers and many person-months (person-years) of development and support time. Many users would be unable to develop such applications without the use of these specialized systems. In fact, these specialized tools are so widespread that they are taken for granted. They share several common characteristics:

- Each addresses a restricted and well-defined problem domain.
- The user interface is designed to be natural to the target user community.
- Features from declarative programming (viz., specifying "what" to compute instead of "how" to compute) are incorporated into the tools, thereby freeing the user from programming details.
- Some commonly support an automatic parallel implementation.
- Many increasingly are becoming web-based and/or web-accessible.

As a result, large communities of users may now develop fairly complex applications; most users would otherwise be unwilling or unable to develop such sophisticated applications.

These same characteristics are incorporated into the Eclpss framework:

- The modelling domain centers on grid-based simulations over time at multiple spatial scales. The calculation of each grid point depends on data within a smallish, localized neighborhood. In addition to the ecosystem modelling domain, other application areas include thermal diffusion, problems which give rise to PDEs, initial-value problems for ODEs, and cellular automata (El Yacoubi et al., 2003) with Cartesian neighborhoods.
- Models are specified using a suite of GUI-based tools which facilitate and support the design practices that are natural to ecologists.
- The ease of developing models is due largely to the declarative nature provided by the framework.

  Parts of a model are expressed at a high level of abstraction as *specifications*. Specifications relieve the user of the need to write (and rewrite) code that manages storage and other mundane but error-prone programming tasks such as the explicit declaration of data structures and loops; the Eclpss compiler uses the specifications to generate this code.
- The framework takes advantage of cutting-edge technology afforded by Java, which supports graphical, web-centric development, collaboration, and dissemination of models.
- Models are automatically targeted to the host machine.
- Most implementation details are invisible to the user.

As a consequence the user does not need a deep understanding of most of this generated code.

This understanding can be especially difficult for the code that is generated for parallel execution. This invisibility permits framework developers to add independent enhancements to both the framework code and the compiler.

In addition to simplicity for the user, the Eclpss compiler must generate efficient (parallel) code. The programming interface is simplified and restricted to framework `get` and `set` methods, which not only eases the conceptual task of the user, but also the analytical task of the Eclpss compiler. The framework rigorously enforces just enough structure, so that a model is relatively easy to analyze, but it does not impose too much structure, so that users have a high degree of modelling freedom.

This research is a collaborative effort with the plant modelling group at the Boyce Thompson Institute (BTI). The requirements for this Java-based framework arose from their experience with an earlier framework implemented in C++ (Beloin and Weinstein, 1994). The reader is directed to Woodbury et al. (2002) for the complete discussion of both the original design goals for a modelling environment, and the comparisons with other tools for building spatially-explicit models of ecological processes. To summarize, the authors compared these tools, including: AME (AME, 2003); ECOSIM (Lorek and Sonnenschein, 1998), EXTEND (Extend, 2003); HOBO (Lhotka, 1994) MEDIA (Meysman, 1999); OSIRIS (Wolf and Boersma, 1996); SME (Maxwell and Costanza, 1997; Voinov et al., 1999); Stella (STELLA, 2003); POWERSIM (Powersim, 2003); and TRIM (TRIM, 2003). The conclusion pertinent to this research is that while these tools (most notably SME and Stella) have many necessary and desirable features, they did not adequately satisfy the authors' requirements to support users in designing, building, and sharing fully three-dimensional spatially explicit ecological models by connecting reusable software components.

Eclpss both satisfies the design goals of their earlier C++ framework, and provides many additional capabilities: a suite of integrated GUI-based tools, platform independence, code generation for sequential and SMP host machines, web-centricity, and automatic document generation.

This paper proceeds as follows. Software and platform requirements are presented in Section 2. Section 3 introduces the modelling environment. Included is a description of the suite of GUI-based editors. (Appendix A motivates this with a simple example.) Tasks allocated to the Eclpss Compiler are outlined in Section 4. Section 5 illustrates how the framework supports model refinement. Some details on framework design are contained in Section 6. Parallel execution is discussed in Section 7, and Section 8 discusses Java performance. Next, Section 9 presents an ecosystem model developed using Matlab, its migration to the Eclpss framework, and their relative performances. Lastly, a discussion of limitations, and ongoing and planned future work is found in Section 10.

2. Software

The Eclpss framework is written in Java and generates Java source code. To use the framework and also execute framework-generated models, the Java 2 Platform Standard Edition (J2SE) version 1.4.1 or later must be installed on the target system. For good performance it is desirable to have at least 256 MB of memory.

The framework may be launched either using Java Web Start™ or by downloading and running the *executable jar*. Java Web Start is a deployment technology that provides users with "one-click" access to the latest version of Eclpss.

Java downloads are available at: http://java.sun.com/j2se/(J2SE) and http://java.sun.com/products/javawebstart/.(Java Web Start)./

Both the framework and framework-generated models been used successfully on platforms for which there is a Java Virtual Machine (JVM): Sun Solaris (Unix); Microsoft Windows 9x, 2000, XP, NT; several flavors of Intel/Linux, and Apple PowerBook (under Mac OS X).

http://www.cs.oswego.edu/~wender/eclpss/ is the Eclpss web site, and contains links to model development examples, tutorials, and a download page.

3. The Eclpss modelling environment

The design of the modelling environment centered around answers to this question: *"If you could have the ideal system at your fingertips, what would you like to be able to do when you first walk into your office in the morning until you leave in the afternoon?"*

3.1. Using the Eclpss framework

As an example of the modelling environment, Appendix A details the development of a simple model that:

- initially populates each cell of a 2-dimensional grid with trees;
- each iteration of the simulation changes the number of trees in each cell;
- the grid is displayed graphically;
- the simulation terminates after a fixed number of iterations.

This model can be constructed by writing a maximum of seven lines of code. (The non-novice programmer can easily write it using four.) The rest of the model definition is declarative.

3.2. Tool set

The suite of integrated GUI-based tools consists of:

- State Variable Editor
- Grid Editor
- Component Editor
- Model Editor
- Unit Editor
- Model Runner.

These tools are used to specify and run models. Each editor saves user input as an XML-encoded *specification object*. The Eclpss compiler, bundled in the Model Runner, uses specifications to generate Java code.

Model documentation from the source code, generated using Javadoc (Sun Microsystems, 2003), is web-browsable and may be hyperlinked into a web page.

3.3. Constituents of an Eclpss model

The Model Editor is used to specify an executable model. A model consists of three entity types and the relationships between them.

These entities, added to the model as specifications, are:

(1) A set of one or more State Variables
    Each State Variable has one or more attributes. An attribute is defined with storage type (i.e., double, int) and a measurement unit (see Section 3.3 below).
(2) A Computational Grid
    The grid has from one to three spatial dimensions over time, and one or more time frames. For each spatial dimension the user specifies the number of cells, a measurement *unit*, and a border type of *torus*, *reflected* or *buffered*.
    Users refer to grid cells using the Eclpss `Point` class. A `Point` object allows the user to write code that executes correctly regardless of the grid dimension.
(3) A set of one or more Eclpss Components
    A Component contains user-written code that performs the actual computation.

3.4. Physical measurement units, unit consistency and unit editor

J.A.D.E. (Dautelle, 2003) is a Java API that may be used to define and manipulate "physical quantities" as Objects.[2]

Users select a J.A.D.E. physical quantity for each State Variable attribute and each Grid dimension. Defining physical quantities for grid dimensions facilitates the seamless use of GIS data for grid population. The framework also provides a GUI that lets users easily construct their own units from existing J.A.D.E. physical quantities.

The user is free to use the J.A.D.E. API (or any other API) in component code. The compiler does not generate code to manipulate J.A.D.E. Objects because of known decreased performance. Instead J.A.D.E. quantities are defined and used symbolically. Consistency checking is performed on J.A.D.E. units as State Variables are added to a model (see Section 3.5).

[2] J.A.D.E. provides a package and classes that enforce strongly typed quantities; J.A.D.E. Version 1 is bundled with Eclpss.

3.5. Eclpss Components

As described in Section 1, Eclpss Components contain user-written Java code that updates the State Variables on the Grid. The framework imposes no restriction on the complexity of the code that is written, but only on the grid itself. Component interaction is indirect via grid updates to State Variables. State Variables are accessible only from the grid through a well-defined and restricted interface.

The Eclpss compiler compiles each Component specification into a Java Component Class. A Component Class contains variables and methods, both static and instance. An instance method is created for each simulation phase: the *Pre-Sim* method executes before the simulation begins; *Sim* executes during the simulation; and *Post-Sim* executes afterward. The method bodies are comprised of the user-written code; method headers are written by the compiler.

The user may write code for any or all of these simulation phases; any or all of these phases may be selected to execute in a model.

Each of these methods is *"cell-based"*: the user writes the code *without loops* that updates State Variables in a grid cell. During model creation, the Eclpss compiler generates the explicit loop code using specifications so that the *Sim* method is executed in every cell of the grid. The user is relieved from the need to write explicit loop code.

Typically *Pre-Sim* and *Post-Sim* methods are written for specialized grid population and for data recording, respectively. The Model Editor provides facilities which generate code to read and write grid data using simple sequential files.

3.6. Eclpss models

The Model Editor bundles all the constituent XML-encoded specifications into one XML-encoded model specification. Once bundled, the model constituents (i.e., the grid, the State Variables and Components) may be modified strictly within the model specification, or exported as individual specifications.

The Constituent tabbed pane is used for adding, removing, and editing model constituents. This pane displays a constituent tree, and is comprised of several top-level sections:

- "Grid" contains the Grid specification chosen for the model.
- "Components" are either user-written or framework-written (for graphical displays, data input, and data output) and may be created, added, edited, and removed here. Since Components are bundled with their State Variables, as Components are added to the model their State Variables are also added to the model.

- "Methods" expands into "Pre", "Sim" and "Post" methods sections. After a Component is added to a model the user selects which of its methods (*Pre-Sim*, *Sim*, and/or *Post-Sim*) are to be executed in these sections of the model (see Execution Groups 3.6.1 below).
- "State Variables" are listed here and edited here. As State Variables are added, those with identical names are consistency-checked for attribute names, types, and measurement units.
- "Statics" contain static variables that are global to the model.

3.6.1. Model Execution Groups

Execution Groups allow users to fine-tune execution behavior. Recall that the *Sim* method of a component is "cell-based". An Execution Group is used to give the "grid-based" description of its execution.

An Execution Group contains a list of Component method(s) that is executed inside its own spatial loop (see Fig. B.1). The methods are executed in the order in which they are listed in the Execution Group. The compiler, therefore, generates a spatial loop for each Execution Group.

To illustrate Execution Groups, consider two Components `A` and `B`. In the first case, `A` and `B` are added to one Execution Group. The generated code structure is:

```
for (int x ...) {
  for (int y ...) {
      // invoke A's method: A updates grid at
         index [x][y]
      // invoke B's method B updates grid at
         index [x][y]
  }
}
```

In the second case, two Execution Groups are defined. The first executes `A`; the second executes `B`:

```
// first execution group:
for (int x ...) {
  for (int y ...) {
      // invoke A's method: A updates grid at
         index [x][y]
  }
}
 // second execution group:
 for (int x ...)

  for (int y ...) {
      // invoke B's method: B updates grid at
         index [x][y]
   }
}
```

Execution group properties may be changed:

- The nesting order of the spatial loop.
- The spatial execution range and stride of each dimension.
- The temporal execution range and stride.
- The memory model (see Section 7.1).
- Spatial or nonspatial execution.
- Execution over absolute (includes border cells for grid dimensions with boundary type *border*) or internal grid boundaries.

3.7. Model execution

The Model Runner performs several tasks:

- It launches the Eclpss compiler, which generates Java source code.
- It compiles the source code.
- It has `Play`, `Pause` and `Stop` buttons to control model execution. The `View Workspace` button brings up a selectable and graphical display of grid State Variable values, which the user may also write to a file. A `Paused` model is resumed with the `Play` button.
- It both displays and allows users to modify the number of processors to use on the host machine.
- A command-line statement is displayed that may be cut and pasted into (for Windows) a Run Program, thereby giving users a stand-alone model running capability.

4. Framework support for compiler tasks

The framework eases the conceptual and writing task of the user by allocating these tasks to the Eclpss Compiler:

- Concrete declaration of data structures
  In conventional programming languages, arrays are declared explicitly by the programmer. Instead of defining arrays explicitly, Eclpss users specify the state variables and grid over which the model executes using their respective GUI-editors. The compiler uses these specifications to define the concrete data structures for each state variable.
- Explicit construction of loops
  Conventional sequential languages require the programmer to write loops explicitly. The programming of spatial loops, should any be required for the target architecture, is normally a relatively tedious task and easily prone to error. The compiler generates the code for spatial loops based on Execution Group information.
- Modularity and code reuse
  Modularity and model sharing are supported through the use of Eclpss Components to perform all computations. Users write only this code (the

method body). The framework imposes an interface (viz. the method header), invisible to the user, that is generated by the compiler. This gives the framework the ability to target a model to different platforms and/or execution models whose implementations may require a different method header.

- Optimization
  Loops are generated to maximize parallelism.
- Imposing a parallel execution model
  The programming problems encountered for a parallel execution model are daunting. By the nature of the restricted problem domain, the parallel programming task is simplified: most communication is local, and iteration over time involves relatively simple patterns of communication.
- Efficient parallel implementation
  A shared memory platform model has been selected for the first parallel implementation. This framework version supports three parallel memory models: fully parallel, red/black tiling, and synchronized. It does not support a distributed or ubiquitous framework, and in fact, because of the way threads currently are used it would be inefficient. (For examples of this type of computing using Java, the reader is directed to Java Grande and Global Grid Forum (Java Grande, 2003; ISCOPE-02, 2002)) The framework will, however, be adapted to this environment in a future version.

5. Framework support for model refinement

It is not uncommon to define a spatial model using *successive refinement*. In this scenario the user begins with a high-level specification of the model. Once the model is verified at this high-level, the user may then successively add more detail until the desired behavior is attained.

There are two general techniques involving State Variables that may be employed in the refinement process:

- *Spatial Refinement*:
  Represent the State Variable with a coarse granularity (the coarsest being a scalar value); then increasing the detail (i.e., decreasing the spatial scale) until it is one-to-one with the Grid scale.
- *Composite Refinement*:
  In each Grid cell, represent the State Variable as a *simple* singleton value; then add detail by representing the State Variable as a *composite* of individual State Variables in the cell.

The Eclpss framework is notable in that it seamlessly supports these two refinement techniques *without the need to rewrite code*.

5.1. The computational grid and data arrays

The framework generates two types of data structures: one *computational grid* and a set of *data arrays*. The *computational grid* is an array whose dimensions are determined from the model's grid specification. A separate *data array* of *Iterators* is generated for each State Variable. The grid contains *references* (pointers) to each of its State Variables (see Section 5.3).

A State Variable may have different spatial resolutions, ranging from as coarse as a scalar value (all the grid cells point to one value) to as fine as the spatial resolution of the computational grid.

5.2. Eclpss spatial refinement

Consider a `Sunlight` State Variable. If the user models `Sunlight` at the "coarsest" granularity, i.e., as a scalar value, the framework generates a scalar data array for `Sunlight`, and maps all cells in the Grid to the `Sunlight` scalar. As the model is refined with Component code, the user may also refine the scale of `Sunlight`, up to the scale of the Grid (see Fig. 1).

5.3. Eclpss composite refinement

State Variables may be *simple* or *composite*. A simple State Variable has just one instance in a cell. This is reasonable when modelling an ecological entity that naturally has a singular value in each Grid cell, such as `SoilType` or `pH`.

Other State Variables lend themselves to a composite representation. Consider a `Tree` State Variable. Initially `Tree` may be modeled as one aggregate per cell. As the model evolves, a more accurate representation of the trees may involve modelling the trees in each cell as a set of individuals (Fig. 2).

Framework methods make single and composite State Variables invisible to the user through the use of *iterators*[3].

State Variables may be referenced as single or composite. However, if there is a chance a State Variable can be refined from a single to a composite, then the user should reference the State Variable as a composite from the start: this removes the need to rewrite the code. By always using the iterator methods, the code need not change whenever the State Variables change between single and composite. Because of run-time optimization, the use of iterator methods on single State Variables will not degrade performance.

[3] An Iterator is a well-accepted design pattern which provides access to the elements of a collection of data without revealing its underlying representation. Examples of such representations are: array, vector, linked list, hash table, etc.

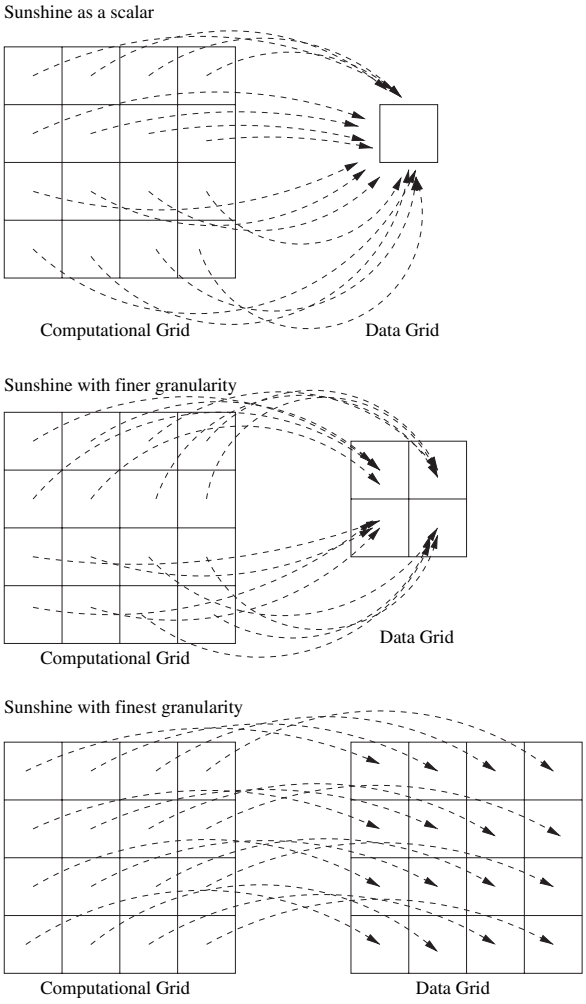

Fig. 1. `Sunlight` state variable with various granularities.

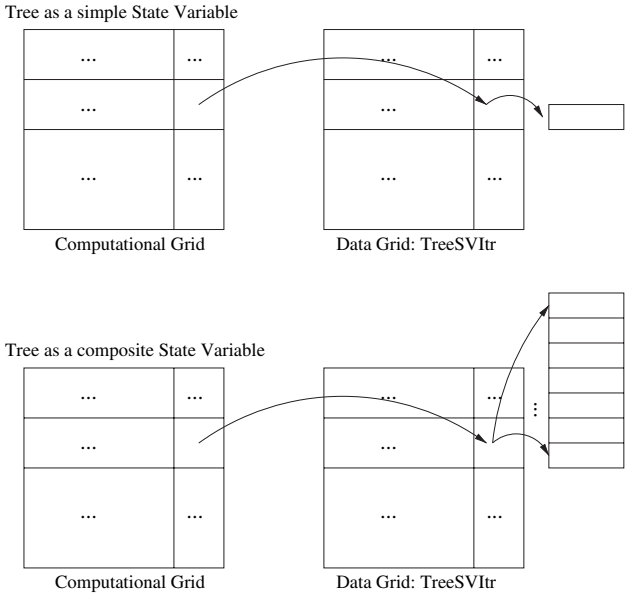

Fig. 2. Example of Iterator for a `Tree` State Variable.

6. Framework design

Eclpss is blackbox framework (van Gurp and Bosch, 2001) that provides the user with three basic functions: editing specifications, generating source code from these specifications, and executing the generated code.

Each Eclpss editor:

- accepts user input from its GUI interface, and interprets this input as specifications;
- creates its own persistent Specification object, which is an XML-encoded document (Quin, 2002);
- has a Builder class (Gamma et al., 1995, Ch 3) that compiles its own specification (and in some cases dependent specifications) into Java source code.

XML is used to describe and structure data, and has become an industry standard for data exchange and web publishing (Quin, 2002). More recently XML has been used as a basis for interfacing databases and simulation models (Kokkonen et al., 2003). An XML document is text-based, and as a result is platform and application independent. Of the many benefits, XML allows documents to be flexible and extensible. The Java XML API makes XML versioning issues invisible: the API sets any newly-added field to its default value (i.e., a field not found in the "older" existing XML object); any newly-removed field causes the field in the "older" XML object to be ignored on open, and not written on subsequent saves. Thus the older framework specification versions may be merged seamlessly into newer versions. XML objects may also be translated into HTML, PDF or Postscript files as an alternative way to display data by using XSLT (XML Stylesheet Language for Transformations). It exists as a supplement to Javadoc documentation, but is not planned at this time.

Generating the source code and executing the generated code are the responsibilities of the Model Runner. The Model Runner's compiler is a Builder class which invokes each editor's Builder class, beginning with the model specification.

The Eclpss compiler uses the Named Object (Rüedi and Sommerlad, 1998) Creational pattern for all Eclpss constituents; the Iterator (Gamma et al., 1995) behavioral pattern is used for State Variable code generation.

6.1. Specification classes

The framework specification classes, shown in Fig. 3, are used to generate the source code classes:

- `ESpec`
- `AbstractESpec`
- `ESVSpecification`
- `ComponentSpecification`
- `GridSpecification`
- `ModelSpecification`

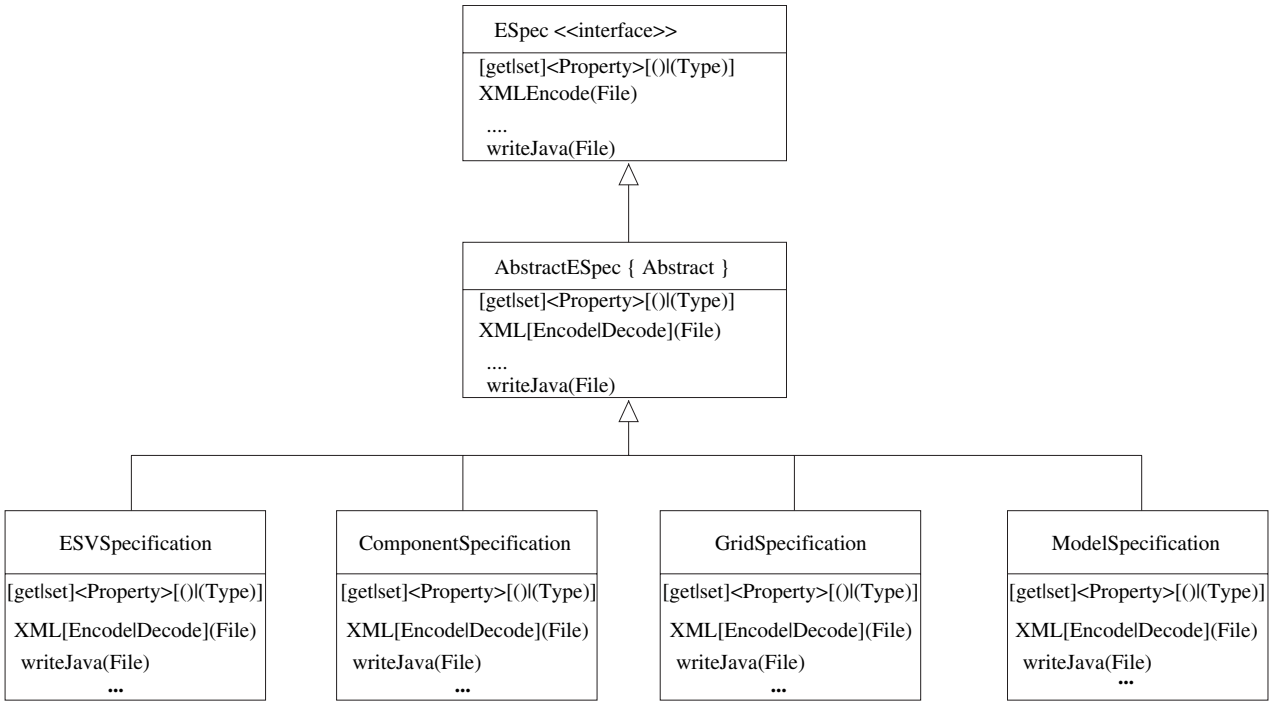

Fig. 3. UML for framework specification classes.

6.2. Code classes

The framework code classes (except where noted) are `interfaces`. Named Object classes are generated by the compiler which either implement or extend these code classes. The code classes are:

- `ESV`
- `ESVIterator`
- `Component`
- (abstract) `Grid [2|3] D_XYZ implements ComputationalGrid`[4]
- (abstract) `Grid [2|3] D_XYZ_NC implements ComputationalGrid_NC`
- `DataGrid`
- `Model`

The abstract grid classes (for example, `Grid2D_BT` and `Grid3D_BBB`) partially implement (interface) ComputationalGrid. The Named grid extends one of these classes.

These abstract grid classes were written for each specific border type combination (Border, Torus, Reflected) to support model debugging. Every grid access (read and write) is checked, and this is done to assist the user in model debugging. It is especially difficult to debug code for reflected boundaries, for example, especially when there has been a boundary violation that does not cause a run-time access violation. The specific borders are tested for and caught as efficiently as possible.

An alternative was to have two grid classes (one for each dimensionality) and test for the border type on each cell access: that adds, at a maximum, two additional tests for border type per dimension, which may degrade performance, depending on the Java compiler, the size of the model, and the number of time iterations. It results in additional framework code, but it is invisible to the user.

The grid classes with suffix "_NB" do no boundary checks; no bounds checking is a simulation execution option. (We found an improvement of about 1 s with the Nitrogen model, described below. This is not terribly significant, but the number of iterations and grid size are also quite small.)

Fig. 4 shows the framework dependencies of a generated model and its own internal dependencies and aggregations. Class names prefixed with "UserNamed" are compiler-generated Named Object source code classes.

6.3. Access to State Variables

There were several (often competing) design goals in designing an easy-to-use API to access the data:

- allow similar data to be grouped in one State Variable;
- eliminate the need to cast;

[4] XYZ are any of the three boundary types B(order) T(orus) R(eflected).

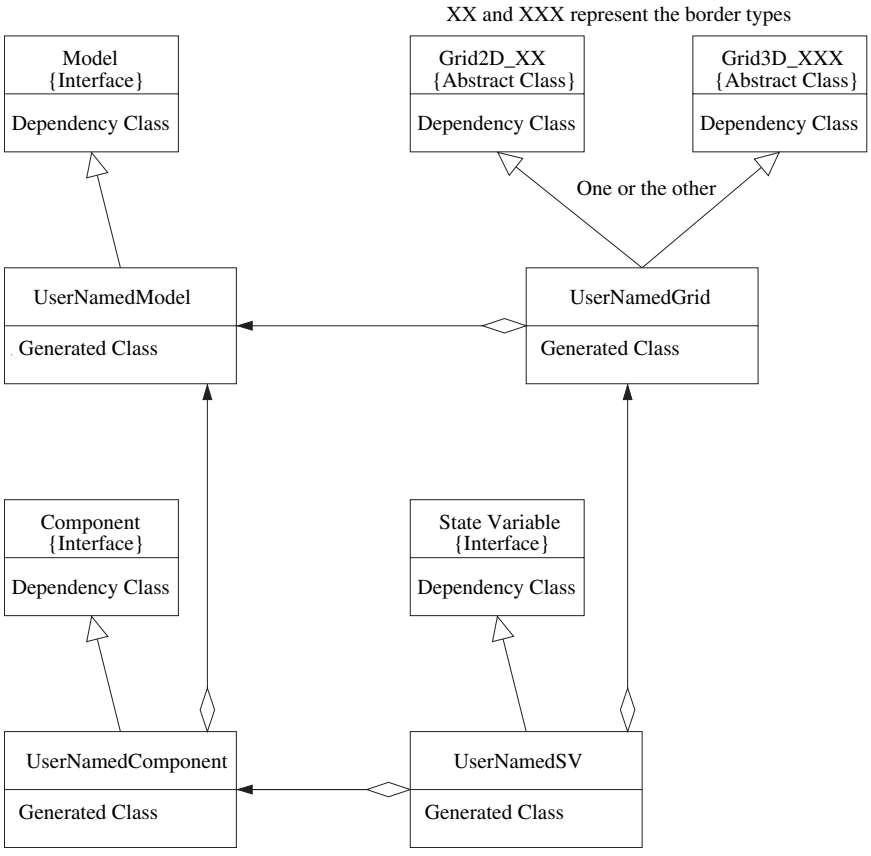

Fig. 4. Class diagram for source code generation.

- maximize performance and safety (see Section 7.1);
- provide seamless access to simple and aggregate values of identical State Variables in a cell;
- make State Variable access code to be self-documenting.

The framework accomplishes these goals. State Variables are treated as first-class Objects. A State Variable may have multiple attributes, static variables, static constants, and Named Object accessor/mutator methods. Grid cells are populated with Named Object State Variable Iterators. The advantages to Named Objects are: it eliminates the need for the user to *cast* data; data access is seamless; code is self-documenting.

The Named Object pattern, for example, gives users a "substance-naming" capability. A State Variable may be given a substance name, say, `soil`, and soil components may be named as its attributes.

6.4. Grid design

Grid properties (such as lower and upper bounds) are accessible only through `get` methods. Grid cells are referenced using the `Point` class. `Point` is an Eclpss class that represents a position (index) in a 2- or 3-dimensional space in time. Component code is independent of both grid instantiation and grid dimensionality.

The user is prevented from inadvertent illegal reads and writes. The following simple rules are enforced at run-time:

- a read may be from any previous or the current time step;
- all writes within a *Sim* Execution Group must be to the current time step;
- writes within a *Pre-Sim* Execution Group may be to any previous or to the current time step.

Illegal writes into boundary cells is particularly irksome for users to detect. Run-time checking of array indices can hinder performance, yet until the user debugs a model these checks are usually necessary. The user can select to execute a model without run-time checking.

There are (legal) writes which affect the memory model that should be used for running the model in parallel. This requires either compiler analysis for code generation, or by (unsafe) user directive.

- Local writes that overlap between threads cause an output dependence.

- Non-local writes (writes to neighboring cells) between threads.

The current version of Eclpss relies on user directives. (see Section 10 for future plans).

The framework provides ample grid accessor methods for each State Variable so that, should the user change the grid properties and border types, the Component code need not be changed.

Grid initialization code may be either user-written or model-generated.

7. Generating parallel code

The Eclpss framework supports sequential and shared memory parallel execution on any operating system where a Java Virtual Machine (JVM) with native thread support is available. Native thread support allows a Java-based thread to map directly to a native operating system thread. Nearly all modern JVMs support native threads. The grid array structure naturally gives rise to a computational tree style algorithm for parallel implementation.

The grid is recursively partitioned until a predefined partition size is reached. (This size is user changeable.) At this point each grid partition becomes the basis for a fork/join task: each task is *forked* to run in parallel and then *joined*. The tasks are *forked* at the beginning of a parallel execution group and *joined* at the end. The tasks enter the machine's processor queue. Fork/join (FJ) techniques are known to scale well and load balancing is not an issue.

Each task, then, sequentially executes all the components in the execution group over its partition. Note that since the tasks are run in parallel, order of execution of the different partitions is indeterminate. Jacobi iteration fits this paradigm.

The framework uses the `dl.util.concurrent` Java package for implementation. Due to the nature of the problem domain (neighborhood communication in different sections of one data array with no other communication or actions required, such as blocking I/O) the implementation does not require the use of Java `Thread` objects. Instead, to achieve better performance, the lightweight `FJTree`, `FJTask`, and `FJTaskRunnerGroup` classes are used. The interested reader is directed to Doug Lea's book (Lea, 1999a, Ch 4.4) and the online supplement (Lea, 1999b) for a complete presentation of these Fork/Join classes.

7.1. Memory models

Each execution group may be executed with a different memory model. Currently this is the user's choosing, making it inherently unsafe (see Section 10).

There are four modes of execution: *fully parallel*, *red/black tiling*, *synchronized*, and *sequential*.

These modes are called Memory Models in order to reflect the manner in which State Variables are read and written (in *memory*).

The decision of which mode to select depends upon the programming model used at both the Component-level and Execution Group-level.

The *Fully parallel* memory model generates code as described above. This model assumes that there are no reads or writes to neighboring cells at the current time, and so tasks may run independently of each other and hence task execution is nondeterministic.

Recall that no order of execution is imposed *between* the different tasks (partitions). When running in parallel, it cannot be assumed that if the current point at time t is (x,y), that $(x-1,y-1)$, say, has been calculated, since that could be a partition boundary!

Whenever such accesses occur the values at time t, if already computed, must be visible to all tasks. This is accomplished by the *synchronized* model. The generated Iterator code defines a mutual exclusion lock (Lea, 1999a; Andrews, 2000) for each State Variable.

The *red/black tiling* memory model (Andrews, 2000, Ch 11.1) is the standard method for parallelizing algorithms such as Gauss–Seidel. Gauss–Seidel iteration uses the most recently computed values to compute each grid cell value. The grid is iterated over from left to right, and from top to bottom. Each grid cell value is computed using a diamond-shaped neighborhood of the current values "above" and "left", and the previous values "below" and "right". In contrast to Jacobi iteration, Gauss–Seidel cannot be parallelized directly because values must be computed in order. Red/black tiling partitions the grid into blocks (or "tiles") using a red/black checkerboard scheme. Since red blocks have all black neighbors (and black blocks have all red neighbors) the parallel algorithm proceeds by updating all red blocks in parallel, and then updating all black blocks in parallel.

Components which perform file I/O, for example, are inherently sequential and require a *sequential* memory model.

8. Java performance

When Java was first introduced its performance was poor because byte code was run interpretively. Because of Sun Microsystem's JIT (Just In Time) compiler, it is now common knowledge that Java's run-time performance is competitive with C++. Executing java programs under these modern Java Runtime Environments (JREs) performs "self-optimization": this means that the longer an Eclpss simulation runs, the faster each

iteration is executed (until, of course, no further optimization can be achieved) (see, for instance, Mangione, 1998; Sun Microsystems (2002)). Garbage collection, where needed, is also no longer an issue; the latest JREs have an option to perform parallel garbage collection. Eclpss models, by contrast, reuse objects within the simulation loop. Consequently the heap is allocated in the *Pre-Sim* phase of simulation and garbage is collected in the *Post-Sim* phase.

9. Migration of an ecosystem model

BTI developed a Nitrogen model, implemented in Matlab, for Hubbard Brook Watershed 6 (Hong et al., submitted for publication). As part of our collaborative work, we converted a preliminary version of the Nitrogen model into the Eclpss framework.

9.1. Matlab model

The Matlab implementation of the Nitrogen model was written by an experienced Matlab user. Loop structures were designed to be consistent with the Eclpss framework. All fluxes are modeled as Eclpss Components. Specifically, components do not invoke other components, and there is also no need for this type of programming model. Components simply update State Variables unlike Eclpss, which implements State Variables as Objects, and includes runtime-accessible units of measurements.

Matlab State Variables are multidimensional arrays of type double and the code, where applicable, takes advantage Matlab loop constructs and operators that execute on [portions of] arrays.

The top-level code consists of a sequence of three loops: the first loop is used for initialization. The second loop is simply a loop over time, wherein each component is invoked once. Each component contains its own spatial loop.

This model consists of water storage pools and fluxes between them. Surfaces capture rainfall and snowmelt. Water infiltration into the soil incorporates both the interception by tree canopy surface and water retention by the soil surface. Retention capacity is modeled using surface slope values. Trees are modeled, in part, with leaf area index and fine root density.

The model executes over a 3-dimensional simulation grid (`X`, `Y`, `Z`) with dimensions $20 \times 27 \times 2$. The `X` and `Y` dimensions have border (depth = 1) boundaries; `Z` has border (depth = 0) boundary.

The model has 30 State Variables and 38 Components. Fourteen Components are used to populate the grid. Two of these Components are invoked at the end of the simulation to output statistical information.

9.1.1. Performance comparisons

The first performance results were obtained on a simpler version of the Nitrogen model that dealt only with hydrology. This model has 23 State Variables and 24 Components. Matlab components were converted essentially line-by-line into Eclpss. Our primary concern was to show how to explicitly map the Matlab code into both the framework and Java code, thereby decreasing the learning curve. Thus, hand-coded optimization of the Matlab code (such as: removing or reducing redundant computations and/or assignments; optimizing internal loops, etc.) was not done. This allowed us to get a fairly good estimate of the performance differences between the two implementations. Both implementations were run on a Dell OptiPlex GX1 with a Pentium 3 and 128 MB of RAM under Windows 2000. The Eclpss implementation, run sequentially, runs at about 3.5 times faster.

The more complex Nitrogen model showed greater performance differences. In this case, both the Matlab and Eclpss models were written to eliminate redundant reads and writes. On the same Dell machine, the Eclpss model runs about 6.5 times faster: 81 s compared to 539 s.

Both models cannot be run fully parallel, red/black or synchronized due to the manner in which reads and writes are performed: current and previous values are read, preventing the use of fully parallel; a square neighborhood is read, preventing the use of red/black tiling. The type of parallelism these models require is block parallelism. Block parallelism is used with distributed memory parallel machines: a grid is partitioned into blocks; the block border cells are then communicated between processors each time step. Block parallelism is anticipated in the next version of the framework.

9.1.2. Experimentation

The Eclpss modelling environment makes it much easier to try alternate behavioral methods.

The initial Matlab implementation executed every component in one cell before advancing to the next cell. (In the framework, this is equivalent to adding all the components to one Execution Group.) The simulation results, though, did not compare well with measured data. Results did compare well when the behavior was changed incrementally until all but one component was executed in all spatial cells before the next component was executed. Thus, instead of one spatial loop nest surrounding all components, the code was rewritten until there were nine spatial loop nests: one with two components, and the rest with one component in each. This model tailoring is not only much easier to do with Eclpss, but is easily understandable because of the visualization afforded by the Model Editor (Fig. B.15).

A "State Variable Report" (Fig. B.16) may be generated, which summarizes the State Variables that are read/written in the model.

10. Discussion of ongoing and future work

Two avenues of framework development are being pursued: compiler analysis for automatic parallel code generation; and speedup of sequentially-executed code.

The performance goal is to capture the power of implicit parallel programming systems without sacrificing the performance of explicit parallel programming. Implicit systems are known to generate non-optimal code, but mainly in cases where the compiler must optimize irregular code over an unrestricted domain. Excluding I/O, performance in this problem class is affected mostly by the ordering of statements, the reuse of memory, the distribution of data, and the transformation of loops to maximize parallelism and data locality. These operations are the responsibility of a compiler.

This version of Eclpss, however, leaves decisions that are rightfully the responsibility of the compiler to the user: the user must select the memory model (which is inherently unsafe).

One of the early software engineering decisions made in the development of this framework was to postpone compiler analysis and optimization. The development and acceptance of the GUI-based modelling environment was deemed a more immediate goal for the first phase of this framework. We feel that since sequential Eclpss models with no optimization have at least a six-fold improvement in run time compared to Matlab models our decision to delay the compiler analysis phase is justifiable.

Due to the framework's restricted access to both the Grid and State Variables, it is relatively easy to perform data dependence analysis on Component code. However we believe it is preferable to provide a GUI through which the user specifies read and write *stencils* for the state variables in the component code. This capability relieves the user from the need to write the grid "read" and "write" accessor (and mutator) methods in component code. It then becomes the job of the compiler, not the user, to write this code. Stencils also greatly simplify the analytical task of the compiler: they contain precisely the information obtained from compiler data dependence analysis.

The purpose of implementing State Variables as Iterators for State Variable Objects is to both impose a modelling environment, and to perform grid bounds checking during model development and debugging.

Once the model is debugged, however, the user should be able to make the (sequential) code also run as fast as possible. The approach being used is to basically strip away all the structure and replace it with arrays defined with primitive storage types. This actually is a relatively easy task (for example, Component code that accesses the Grid and State Variables is easily found and changed with lexical analysis) and we anticipate that it will be part of the next version release.

We continue to collaborate with BTI to develop ecosystem models, and have begun expanding the user community by investigating spatial ecological economics models.

The following lists plans for additional framework functionality:

- A *Stencil* GUI to the Component Editor (January 2005).
- Block distributed memory parallelism (January 2005).
- Adding components using Drag-and-Drop, including between separately-running JVMs over the web.
- GIS integration.
- Addition of simple application APIs (such as central differences, forward differences) to the framework to alleviate the need to code them from scratch.
- Finite Element capability.

Acknowledgments

This publication was supported by a subcontract with Boyce Thompson Institute for Plant Research, Inc., under Agreement Number R-82795801-0 from the U.S. Environmental Protection Agency. Anthony Vito contributed significantly to the design and coding of the framework. The Hydrogen and Nitrogen models were written using Matlab by Bongghi Hong, and written using Eclpss by Erick Smith. Thanks go to the anonymous referees, whose comments led to numerous and significant improvements in the paper.

Appendix A. A simple model

The modelling environment is best presented with a very simple (and unrealistic) problem. Appendix B shows screenshots of some GUI editors.

This model grows `Trees` in each cell of the Grid. Each time step, the `Number` of `Trees` in each cell at time t is one plus the number of trees in its corresponding cell at time $t-1$. All the trees in a cell die when the number of trees reaches a maximum number.

We use modular (residue) arithmetic to limit the maximum number of trees in each cell to (`Tree.Number_MAX-1`)[5]. By associating a color with the minimum and maximum number of trees, a graphical display of the number of trees in each cell over time results in a geometrical pattern.

Each cell is autonomous: the number of trees in any cell at time t; a function of its value in the previous time

[5] Java provides the infix operator modulus "`%`". The expression (`d % m`) returns the remainder of dividing `d` by `m`.

$t-1$. This model is even simpler than the Game of Life since the value of each cell is *not* a function of any of its *neighboring* cells.

A.1. Creating the initial model

A.1.1. State Variable Editor

We define a `Tree` State Variable with one attribute named `Number`, with storage type `int`, which we will use to keep a count of the number of trees in the grid cell. Attributes of State Variables have units; we define `Number` to be `dimensionless`. As part of our `Tree` State Variable we decide to define a scalar constant `MAX` for the `Number` attribute: the maximum number of trees in a cell. The framework will generate the constant name `Tree.Number_MAX` (see Fig. B.2).

A.1.2. Component Editor

Eclpss Components are objects. Each component has a set of three methods for you to define (or not): *Pre-Sim*, *Sim*, and *Post-Sim*. The *Pre-Sim* method is executed *before* the simulation begins; *Sim* is executed *during* the simulation, and *Post-Sim after* the simulation.

Our model needs just one Component (`GrowTree`) to manipulate the one State Variable `Tree`. After we `add` the `Tree` State Variable to the component, we may generate the `Tree` javadoc documentation (Fig. B.3) to facilitate code writing.

We define the *Pre-Sim* and *Sim* (Fig. B.5) methods by entering the following code in their respective areas:

- *Pre-Sim* populates the border and interior grid cells with one `Tree` state variable. The number *n* of trees in each grid cell at index `[x][y]` at the current time is initialized to `(i+j) % Tree.Number_MAX`. The user is:

```
int i = current.getX(); // get x index
int j = current.getY(); // get y index
int n = (i+j) % Tree.Number_MAX;
grid.writeTree(current,CURRENT).add(new
Tree(n);
grid.writeTree(current,PREVIOUS).add(new
Tree(n);
```

Code explanation:

- `current` is the absolute address of the current cell.
- `CURRENT` and `PREVIOUS` are constant grid offsets, and are added to `current`. `CURRENT` denotes *here* and *now* (the current time); `PREVIOUS` denotes *here* and *then* (one time step back).
- `grid.writeTree` performs a write bounds check on `grid`.
- `add(new Tree(n))` initializes the grid to have a new Tree object. The number (and order) of arguments for a SV (State Variable) constructor corresponds to the number and order of SV attributes defined.
- `Tree.Number_MAX` is a constant defined for the `Number` attribute.

This code fragment initializes all the cells in the grid, in both the current and previous time frames[6].

- *Sim* contains the code to get the number of trees in the current cell in the previous time frame, increment this value by 1 (*modulo* the maximum number of trees), and set the current cell in the current time frame to this new value. The required user code is:

```
int n = grid.readTree(current,PREVIOUS).get-
Number(); grid.writeTree(current,CURRENT).set-
Number((n+1)% Tree.Number_MAX);
```

Code explanation:

- `grid.readTree(current,PREVIOUS)` performs a read bounds check on `grid`.
- `getNumber()`: the framework generates `get` < Attribute > () and `set` < Attribute > (< value >) methods for every SV Attribute.
- *Post-Sim* writes "Simulation ended" to the console.

Notice that the Component Editor automatically generates (and updates) the "`SV get(s)/ set(s)`" pane (Fig. B.4).

Thus the user writes a total of about seven lines of code the user for this model. The rest of the model is defined via GUI-based specifications with the appropriate editors.

The Component Editor writes the component specification with all of its XML-encoded State Variable specification into one XML-encoded file. This bundling of State Variables with components supports reuse.

A.1.3. Grid Editor

We define (Fig. B.6) a 2-dimensional `SimpleGrid` Grid.

The X and Y dimensions are both defined to have 20 cells. Since grid measurement units are not necessary for this model, we select `dimensionless`. (Measurement units would be relevant for, say, a model with multiple spatial scales, or State Variables whose measurement units require must be mapped to grid measurement units, as in concentration or density). Since the `GrowTree` component does not read any neighbors, we decide on a bordered buffer with a border depth of 0. If we later decide to give the grid border cells or we read neighboring values, only our grid border specifications need to be changed. This is because there are

[6] This model may certainly be implemented with only a current time; the previous time is defined only for illustration.

framework-generated grid accessor methods for referencing interior and border cells. These methods are written in framework-generated code, and also available for the user.

A.1.4. Model Editor

The last step is to define our model. The Constituents pane allows us to both specify the entities that comprise the model, and also to specify the behavior of the model with Execution Groups (Fig. B.7).

We add `SimpleGrid.grd` to *Grid*;

We add `GrowTree.cpt` to *Components*.

Since components are bundled with their State Variables, as soon as we add `GrowTree` we find that `Tree` is listed under State Variables.

Had there been another `Tree` state variable in the model (say, from a different Component that we added previously) the framework would perform a consistency check to verify that all attributes of both `Tree` State Variables are identical in name, in storage type and in measurement unit; and if not identical, would issue a warning and the reasons for the warning.

Once the `GrowTree` Component is added to the model, we need to *add* the *methods* in `GrowTree` that we want executed in the simulation: *Pre-Sim* and *Sim*. When we open up Methods in the Constituents pane we see these three different sections.

Component methods are added to Execution Groups. An Execution Group contains a list of methods that are executed in their own spatial loop and allow you to tailor the simulation. This code gets generated by the framework. The methods listed in an Execution Group are executed sequentially, one after another, in the order in which they appear. Execution Groups are executed sequentially. There are default spatial and temporal execution environments for Execution Groups, which may be overridden. They are:

- *Pre-Sim* and *Post-Sim* Execution Groups are non-spatial.
  Nonspatial is the default execution. Nonspatial execution requires users to write their own loop nest within the *Pre-Sim* method. This allows users to tailor grid population.
  Since we want all the border and inner cells to conform to the modulus arithmetic for number of trees.
  We set the default setting to make the Execution Group spatial *and* make the *Pre-Sim* method execute over the `Absolute grid bounds`.
  `Absolute grid bounds` generate code that executes over the *entire* grid, not just the grid within the simulation space. In our simple model, the absolute grid boundaries and simulation boundaries are the *same* because we selected buffered border with border depth of 0. If we, instead, declared the buffered border with a depth greater than zero, we still want to populate the entire grid. Regardless of the border depth specified in the Grid Editor, neither our *Pre-Sim* component code nor our spatial Execution Group specification need to be changed!
- *Sim* Execution Groups are spatial and execute over the entire grid *within* the grid borders. The nesting of the spatial loops may be changed, as can the grid boundaries. The default execution model is fully parallel, regardless of the number of processors on the host machine. Thus it is possible to run a model fully parallel with one processor.

You may define one or more Execution Groups. Execution Groups may be (re)named and rearranged.

We wish to execute all the methods in `GrowTree` (Fig. B.9).

- We add `GrowTree` to the first (and only) Execution Group in *Pre-Sim* and we name it "Populate Grid with Trees". Since we have written our own loop nest in the method, we keep the *nonspatial* default.
- We add *GrowTree* to the first Execution Group ("Grow the Trees") and only Execution Group in *Sim*. The defaults for execution are: over the entire grid within the inner boundaries; over the entire simulation time; in *Fully Parallel* mode.
- We add *GrowTree* to the first Execution Group ("Simulation ended") and only Execution Group in *Post-Sim*.
- Next, we establish the simulation run time as a *duration*, with a starting (1) and ending time (1000), with a step size of 1.
- Lastly, when we save the model, an XML-encoded file named `Example1.mdl` is created. Just as Components are bundled with their State Variables, models are also bundled with all its constituents, thus facilitating model sharing and reuse.

A.1.5. Model Runner

The "Play" button in the Eclpss menu launches the Model Runner.

The Model Runner generates the Java code from the model file and then compiles all the code.

After the code is compiled, the user has several options. The number of processors are displayed; this number may be changed. The model may be "Play"ed, "Pause"ed, or "Stop"ped. By pausing a model, the Workspace (whose functionality is similar to Matlab) may be viewed and/or saved to a file. Model execution may then be resumed by hitting the "Play" button. Any `Sytem.out.print[ln]` statements in the model code are written to a console window.

Notice that there is text written in the *Command* field. This is the command that you use to run the simulation

outside of the Eclpss framework. Simply *Copy* the command to a file (script) and then execute whenever you wish to run the simulation.

A.2. Enhancing the model

A.2.1. Generating a graphical display for a State Variable

It is more illustrative to have graphical output instead of just displaying "Simulation ended". We take advantage of the feature under *Components* in the Constituents pane of the Model Editor to that lets us *generate* a graphical component for our `Tree` State Variable (see Fig. B.10).

- Select the State Variable and Attribute for the graphical display. In this case, our model has just one State Variable (`Tree`), and Tree has just one attribute (`Number`).
- We have found that standard and high resolution 17-inch monitors render a reasonable-sized (a bit less than 1/4 of the screen) graphics window when Width and Height of 500 are used, and so the framework uses these as default values in the text fields.
- Associate a color with each of the minimum and maximum values. We define 0 for the minimum value, and type in `Tree.Number_MAX` for the maximum value. The HSB color scheme is used to create a color swatch, which is displayed in the GUI and the graphical display (in the running model).
- Time Modulo lets us select how often we want the display to be updated.

We want the grid to be displayed after all the State Variables have been updated in the current time. To do this, we add a new Execution Group to execute after we grow our trees, add our Graphical Component to it, save the model, and launch the Model Runner (see Figs. B.11 and B.12).

For timing studies (an execution option) we usually do not want graphical displays. We also do not want to remove the execution group. The solution: use the disable execution group feature (see Fig. B.13).

Appendix B. Screenshots

B.1. Simple model screenshots

The screenshots of the simple model are presented in Figs. B. 1–B. 13.

B.1.1. Hydrology model screenshots

The screenshots of the hydrology model are presented in (Figs. B.14–B.16).

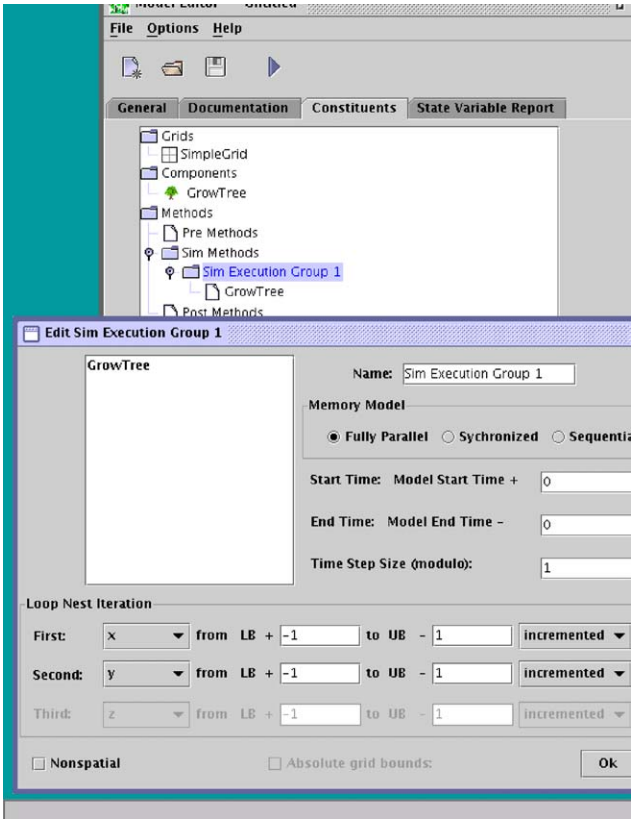

Fig. B.1. Example of a *Sim* Model Execution Group.

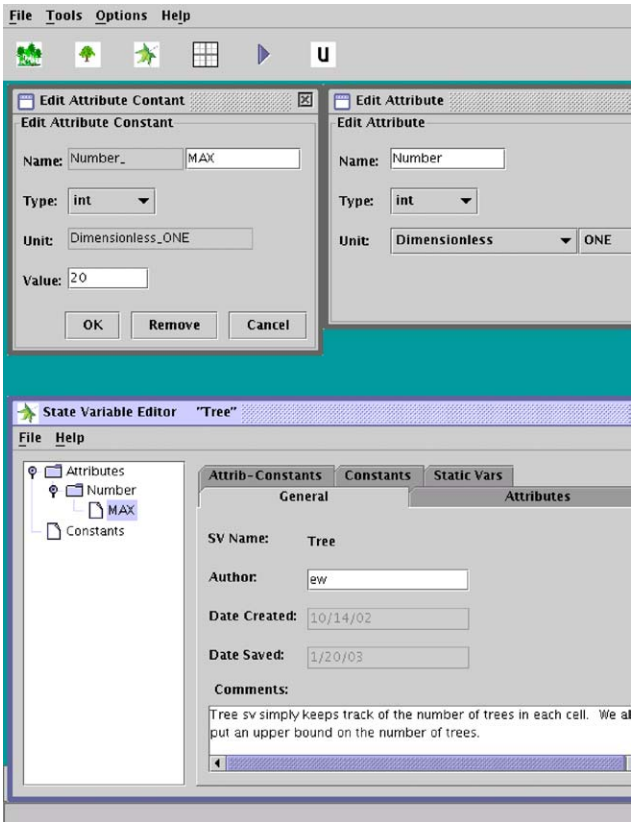

Fig. B.2. State Variable Editor: `Tree`.

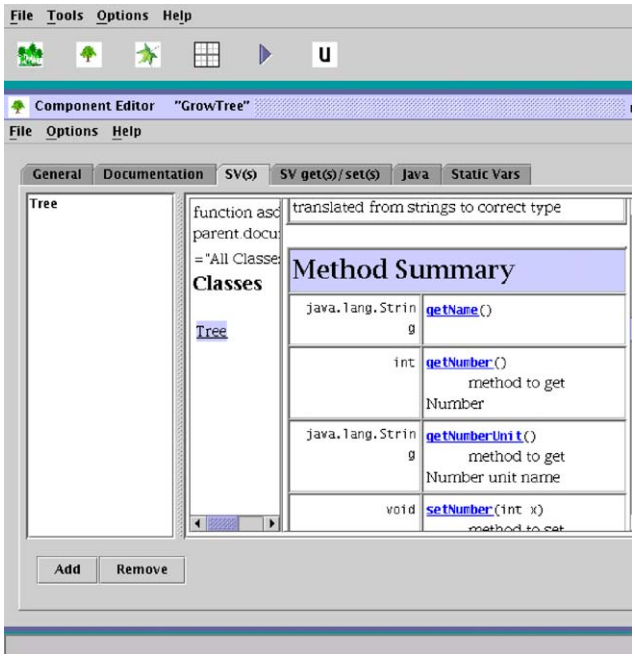

Fig. B.3. Component Editor: generated `Tree` documentation.

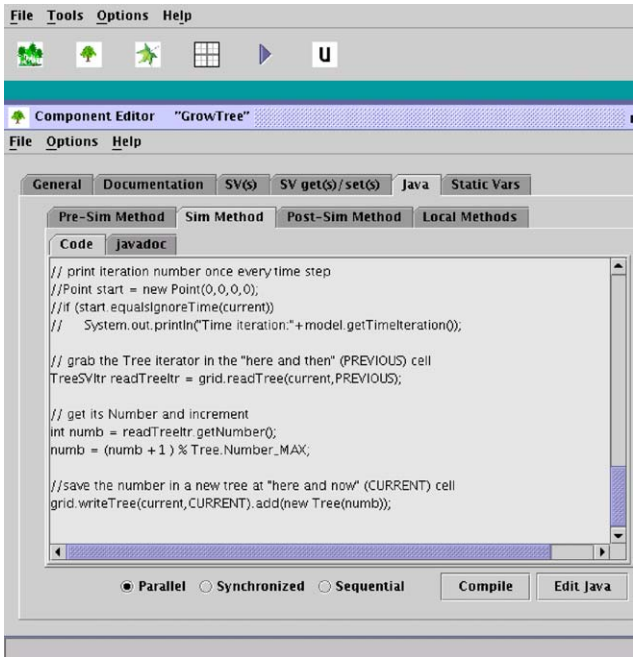

Fig. B.5. Component Editor: *Sim* code.

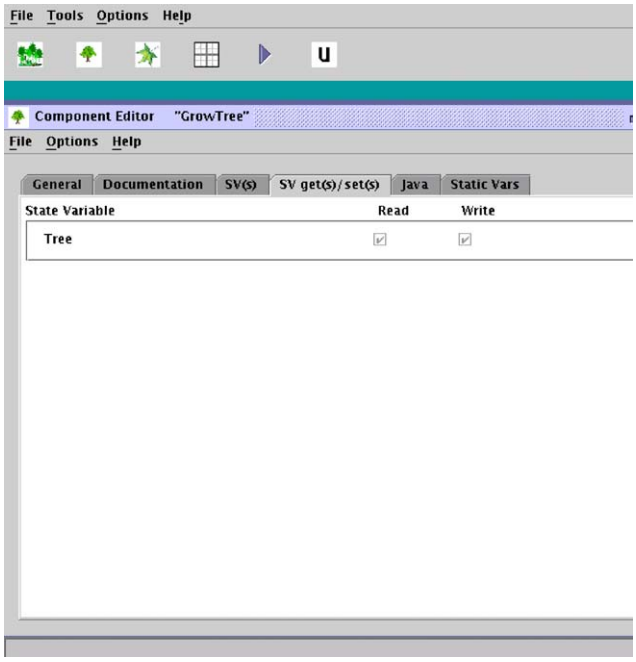

Fig. B.4. Component Editor: generated `Tree` references.

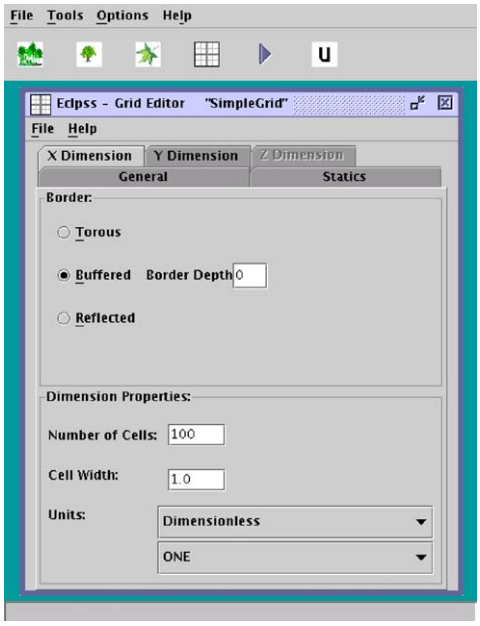

Fig. B.6. Grid Editor: specifying grid dimension `X`.

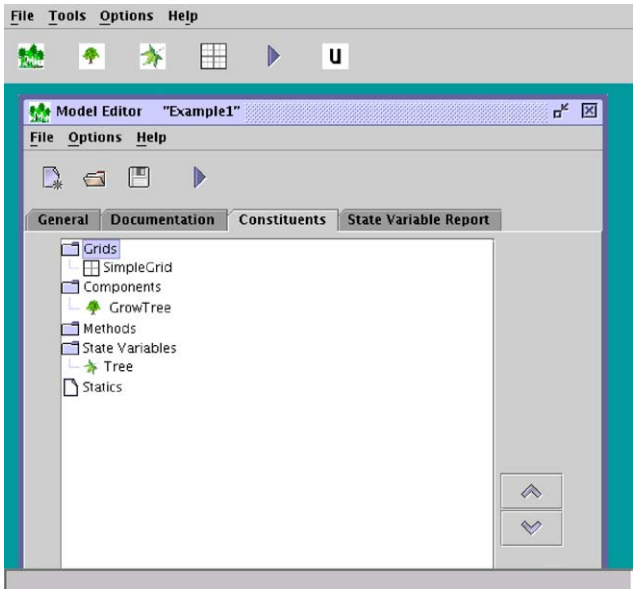

Fig. B.7. Model Editor: adding constituents.

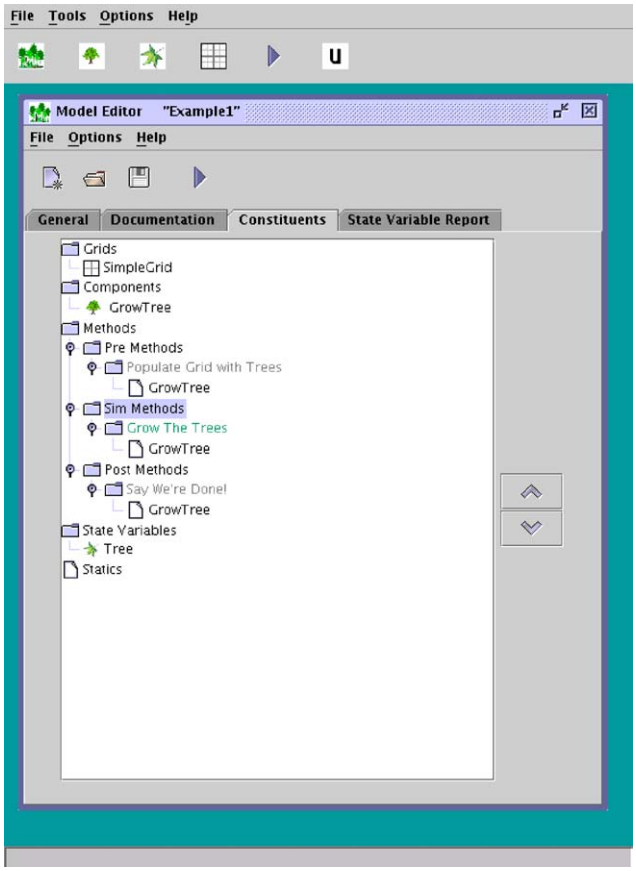

Fig. B.8. Model Editor: adding Execution Groups.

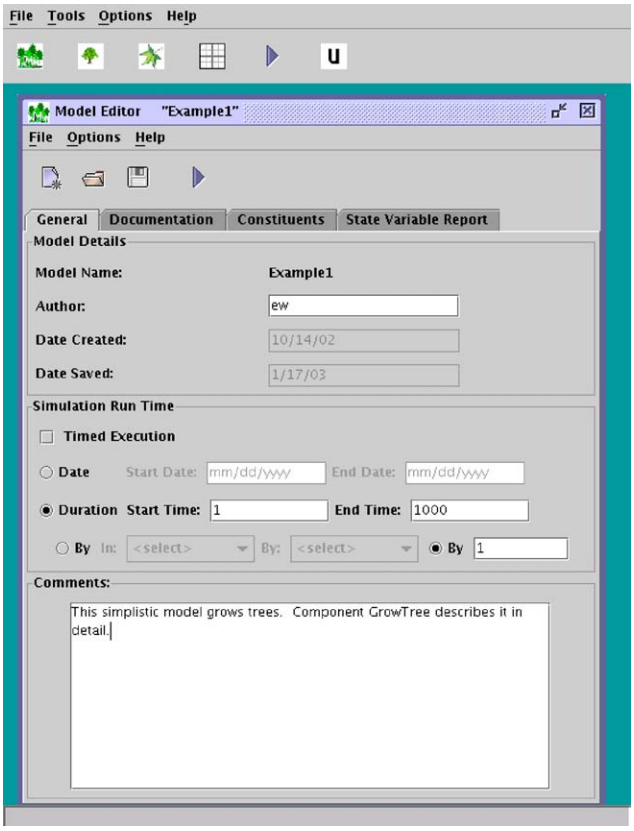

Fig. B.9. Model Editor: setting simulation run time.

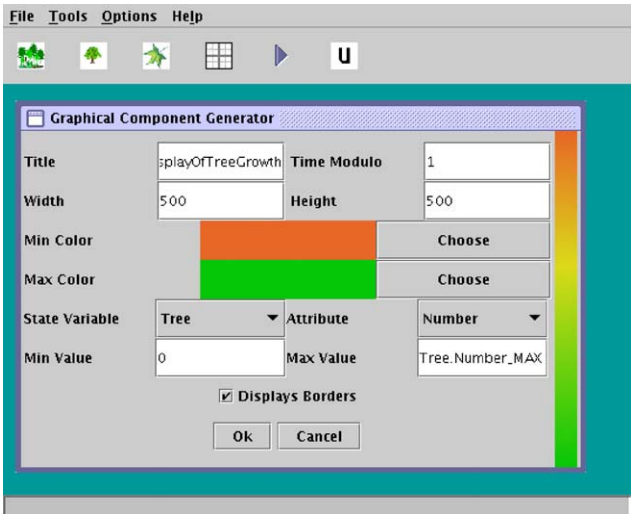

Fig. B.10. Graphical component generator.

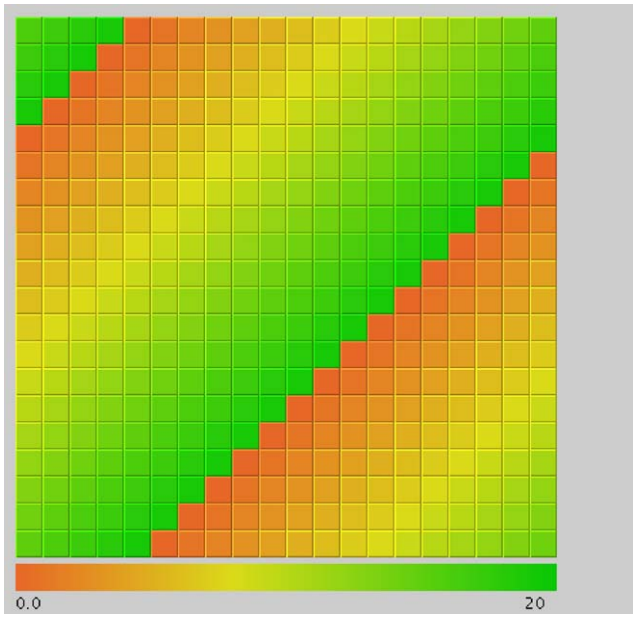

Fig. B.11. Graphical output from Model Run.

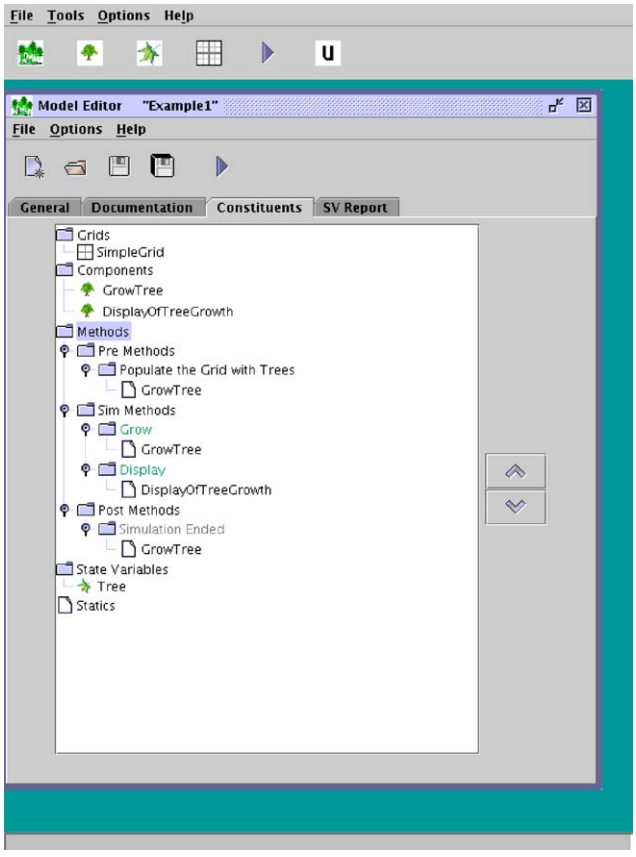

Fig. B.12. Constituent pane.

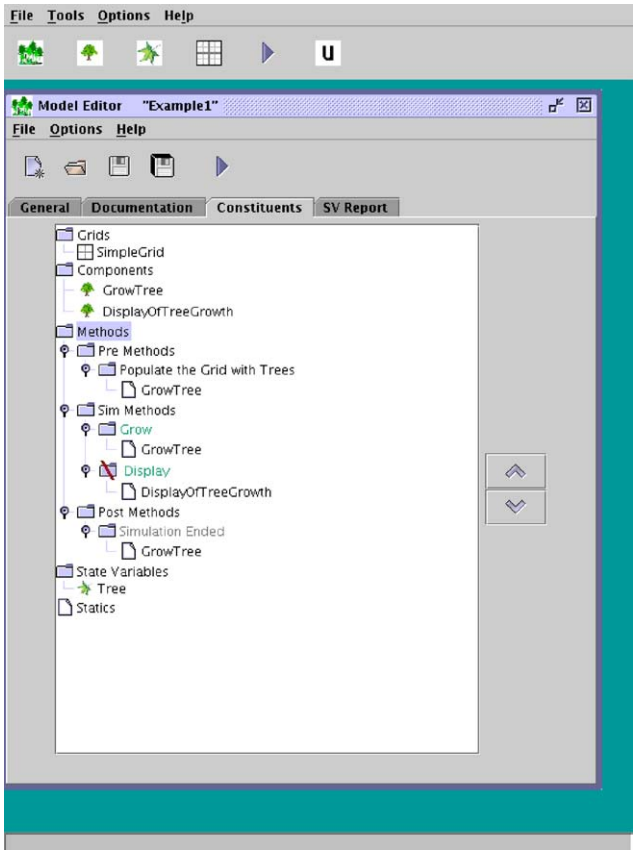

Fig. B.13. Disabling an Execution Group.

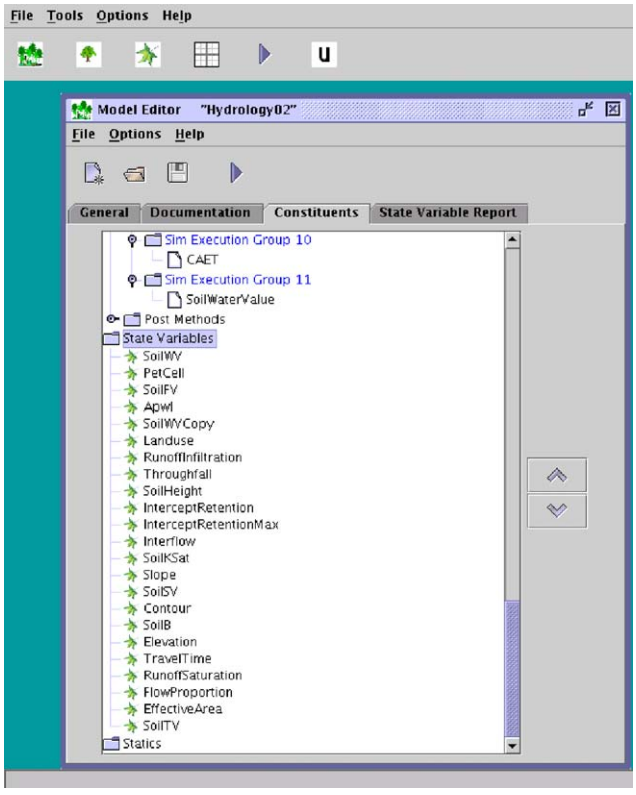

Fig. B.14. Hydrology Model State Variables.

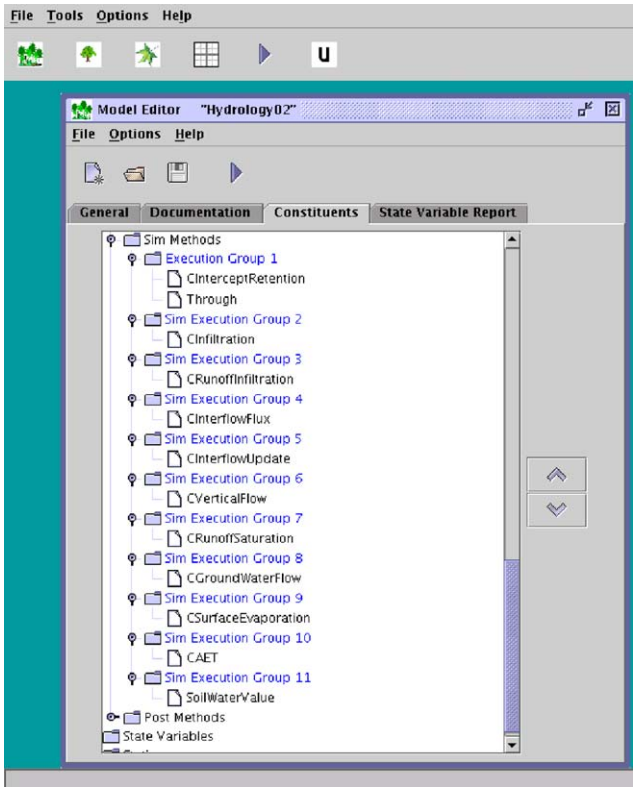

Fig. B.15. Hydrology Model Execution Groups and Components.

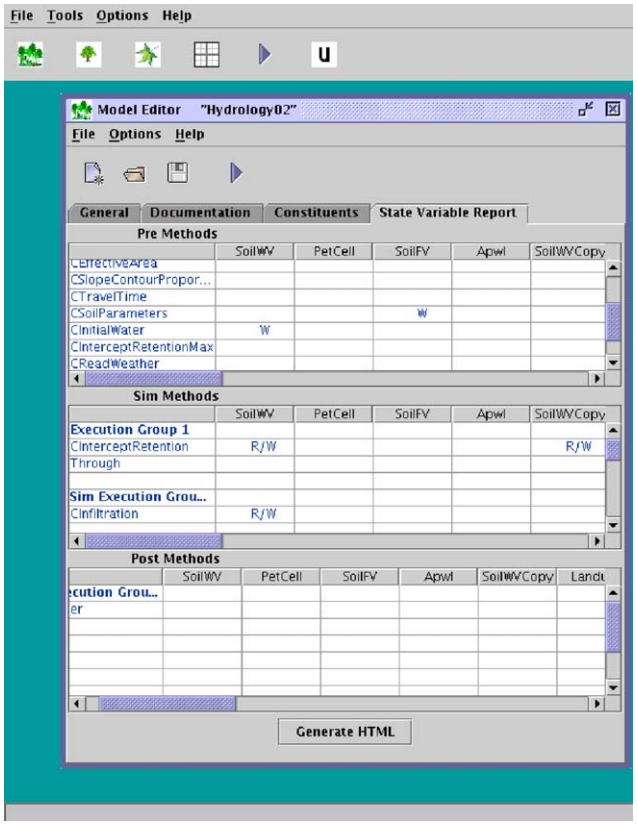

Fig. B.16. Hydrology Model State Variable report.

References

Allen, J.R., Kennedy, K., Oct. 1987. Automatic translation of FORTRAN programs to vector form. ACM Transactions on Programming Languages and Systems 9 (4), 491–542.

AME, 2003. Available from: http://simulistics.com/.

Andrews, G.R., 2000. Foundations of Multithreaded, Parallel, and Distributed Computing. Addison Wesley Longman.

Beloin, R.M., Weinstein, D.A., 1994. An object-oriented framework for spatial modeling using libraries of reusable components. In: Power, M.J., Strome, M., Daniel, T.C. (Eds.), Decision Support 2001, Proceedings. American Society for Photogrammetry and Remote Sensing, Bethesda, MD p. 510.

Dautelle, J.-M., 2003. J.A.D.E. Java™ Addition to Default Environment. Available from: http://jade.dautelle.com/.

El Yacoubi, S., El Jai, A., Pausas, J.G., 2003. Lucas: an original tool for landscape modelling. Environmental Modelling & Software 18, 429–437.

Extend, 2003. Available from: http://imaginethatinc.com.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley.

van Gurp, J., Bosch, J., 2001. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. Software: Practice and Experience 31, 277–300.

He, H.S., Larsen, D.R., Mladenoff, D.J., 2002. Exploring component-based approaches in forest landscape modeling. Environmental Modelling & Software 17, 519–529.

Hong, B., Swaney, D.P., Woodbury, P.B., Weinstein, D.A. Long-term nitrate export from Hubbard Brook Watershed 6 driven by climatic variation. Water, Air, and Soil Pollution, submitted for publication.

ISCOPE-02, 2002. Proceedings of the Joint ACM Java Grande-ISCOPE Conference, November 3–5, 2002. ACM SIGPLAN, Seattle, Washington.

Java Grande, 2003. Java grande forum. Available from: http://www.javagrande.org.

Kokkonen, T., Jolma, A., Koivusalo, H., 2003. Interfacing environmental simulation models and databases using XML. Environmental Modelling & Software 18, 463–471.

Lea, D., 1999a. Concurrent Programming in Java™: Design Principles, second ed. Addison-Wesley.

Lea, D., 1999b. Online Supplement for Concurrent Programming in Java™: Design Principles, second ed. Available from: http://gee.cs.oswego.edu/dl/cpj.

Lhotka, L., 1994. Implementation of individual-oriented models in aquatic ecology. Ecological Modelling 74, 47–62.

Lorek, H., Sonnenschein, M., 1998. Object-oriented support for modelling and simulation of individual-oriented ecological models. Ecological Modelling 108, 77–96.

Mangione, C., 1998. Performance tests show Java as fast as C++. Available from: http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html.

Maxwell, T., Costanza, R., 1997. An open geographic modeling environment. Simulation 68.

Meysman, F., 1999. MEDIA: a general purpose diagenetic modeling environment for marine and estuarine sediments. In: "ELOISE" – European Land–Ocean Interaction Studies 3rd Open Science Meeting.

Powersim, 2003. Available from: http://www.powersim.com.

Quin, L., 2002. Extensible Markup Language (XML). Available from: http://www.w3.org/XML/.

Rüedi, M., Sommerlad, P., 1998. Named objects. In: Third European Conference on Pattern Languages of Programming and Computing.

Ruhl, R., Annaratone, M., 1990. Parallelization of fortran code on distributed-memory parallel processors. In: Proceedings of the 1990

International Conference on Supercomputing, Amsterdam, The Netherlands, June 11–15, 1990, published as ACM SIGARCH Computer Architecture News 18(3), pp. 187–200.

STELLA, 2003. Available from: http://www.hps-inc.com.

Sun Microsystems, 2002. Java™ 2 Platform Standard Edition: Performance and Scalability Guide. Available from: http://java.sun.com/j2se/1.4/performance.guide.html.

Sun Microsystems, 2003. Javadoc™ tool home page. Available from: http://java.sun.com/j2se/javadoc/index.html.

TRIM, 2003. Available from: http://www.epa.gov/ttn.uatw.urban.trim.trimpg.html.

Voinov, A., Costanza, R., Wainger, L., Boumans, R., Villa, F., Maxwell, T., Voinov, H., 1999. Patuxent landscape model: integrated ecological economic modeling of a watershed. Environmental Modelling & Software 14, 473–491.

Wolf, M.M., Boersma, M., 1996. An object-oriented simulation framework for individual-based simulations (osiris): *Daphnia* population dynamics as an example. Ecological Modelling 93, 139–153.

Woodbury, P.B., Beloin, R.M., Swaney, D.P., Gollands, B.E., Weinstein, D.A., 2002. Using the ECLPSS software environment to build a spatially explicit component-based model of ozone effects on forest ecosystems. Ecological Modelling 150, 211–238.