AN ABSTRACT OF THE THESIS OF

Patrick C. Wingo for the degree of Master of Science in Biological and Ecological Engineering presented on June 8, 2015.

Title: OME: A Framework for Running Spatially Explicit System Dynamics Models.

Abstract approved:

John P. Bolte

The Open Modeling Environment (OME) is a tool developed to address some known shortcomings in ecological System Dynamics (SD) modeling research. OME provides a common set of methods for interacting directly with spatial information, reducing the need for modelers to create their own methods for doing so. The environment is capable of running as a standalone program, or as a plugin to an existing tool, allowing OME to be useful in a wide range of SD applications. This thesis demonstrates OME's capability by taking a pair of existing SD models and running them at a speed and with an accuracy comparable to their original simulation environments. Additionally, OME is demonstrated running as a plugin to a much larger model simulation framework, where it provides benefits through adding additional simulation dynamics. While much work went into OME, this is only the first phase, with performance optimizations, increased compatibility with existing model formats, and multi-platform implementations all being explored as future directions of development.

OME: A Framework for Running Spatially Explicit System Dynamics Models

by
Patrick C. Wingo

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 8, 2015
Commencement June 2015

Master of Science thesis of Patrick C. Wingo presented on June 8, 2015.

APPROVED:

Major Professor, representing Biological and Ecological Engineering

---

Head of the Department of Biological and Ecological Engineering

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Patrick C. Wingo, Author

ACKNOWLEDGEMENTS

The author would like to thank the following people, in no particular order, for their help throughout this project. Without out you the development of OME would not have been possible.

Prof. John Bolte, Oregon State University
Allen Brookes, USEPA.
Roel Boumans, AFORDable Futures LLC.
John Rogers, USEPA.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX FIGURES

Figure Page

LIST OF APPENDIX TABLES

LIST OF APPENDIX TABLES (Continued)

Introduction

The project underlying this thesis is primarily concerned with the development, validation, and adoption of the Open Modeling Environment (OME). OME is an implementation of a System Dynamics (SD) model simulation engine which is intended to address a few hurdles that ecosystem modelers may face today. OME attempts to assist the modern ecosystems-focused SD community by providing a common interface for explicit spatial queries and a flexible implementation designed to either work alongside or integrate with existing modeling tools. By addressing these issues, OME will make a positive contribution to the ecosystem modeling community.

Since a large portion of this discussion revolves around models, it is important to establish what a model actually is. Definitions vary in their complexity, but models basically serve an important role in scientific research at large, primarily in their abilities to assist with understanding a real or theoretical system, predicting some future or alternate state, and controlling for experimental or unknown values when manipulating a system (Haefner 1996). Every model operates under a series of assumptions and constraints for which its conclusions remain valid, and exists in a space that is a tradeoff between reality (how close a model mimics real-world processes), precision (the degree of accuracy in produced predictions), and generality (the range across which model results remain valid) (Haefner 1996). The balance between these trade-offs produces a model which can be used to tease out pertinent information and highlight relationships that are significant to the model's defining research question (Haefner 1996). Models can also be used as communication tools by providing a way to access information derived from complex relationships which would otherwise be obscured from a general audience's understanding (Lane 2008). SD models in particular can be used to develop a better understanding of a given series of processes, especially when one or more causal relationships are emulated (Ford 1999).

As mentioned previously, OME is specifically concerned with the construction and evaluation of System Dynamics (SD) models. SD modeling involves the study of how a system's structure affects the change in quantified system aspects over time (Lane 2000); examples of

“quantified system aspects” in an ecosystem model would be nutrients, pollutants, individuals in a population, or any other quantity that flows through and/or circulates within a system. SD models are generally constructed from a set of graphical elements as a stock-flow diagram, where the configuration of elements in relation to one another defines the differential equations which drive the interactions within the model (Lane 2008). This configuration of diagram elements also assists in model conceptualization (Lane 2008), and can be provided as a communication aid to a larger audience who would be otherwise unable to interpret model dynamics when represented by calculus-based equations (Lane 2008). Different implementations vary, but all SD models contain at least these five components: Compartments, Flows, Variables, Influences, and Source/Sinks (Lane 2008). Compartments, also referred to as state variables or stocks, represent points of storage for quantities moving through the modeled system, and are the components upon which the model's differential equations act upon (Lane 2008). Flows represent the paths of values that stream through the system, and represent the rate at which a value accumulates or dissipates within a targeted compartment (Lane 2008). Variable components represent other values external to flow components, which can influence the values processed within the differential equations driving the simulation (Lane 2008). Influences map out the interactions between variables, flows, and compartments, essentially acting as roadmaps to how all other visual pieces of the representative stock-flow diagram tie together (Haefner 1996; Lane 2008). Sources and Sinks represent quantities that enter from and exit through the implicit boundary of the system, respectively; these quantities are not tracked by the SD model, as their values are considered out-of-scope of the system being simulated (Lane 2008). Each of these components are graphically represented in a consistent way: compartments are rectangular boxes, flows are arrows with a triangle shape (or occasionally some valve-like shape) placed partway along the line, variables are round bubbles, influences are thin or dashed arrows, and sources/sinks have an amorphous cloud shape (Haefner 1996); see Figure 1 for an example SD model and a legend for its icon usage. Mathematically, the collection of graphical model components are combined into a series of differential equations which are then solved along regular time intervals to produce the dynamics of the model, simulating the flow of compartment-captured values through the simulated system. The representative equation that

each constructed differential equation can be reduced to is $dX/dt=f(X,P)$, where **X** is a vector of quantities in each compartment, and **P** is a vector of parameter values associated with the compartments represented by **X** (Lane 2008).

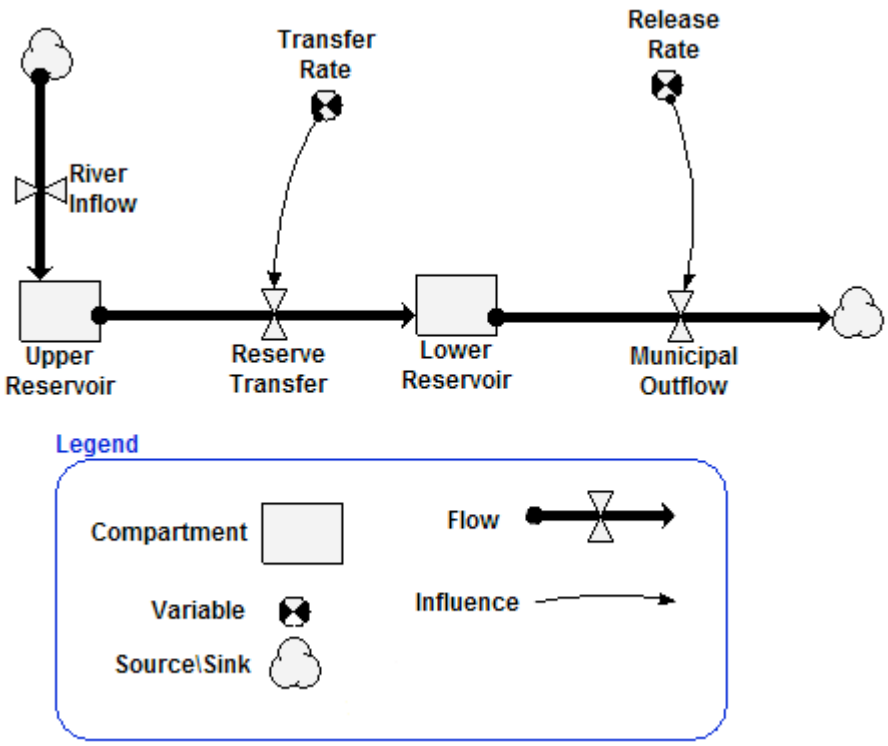

Figure 1. Example of a simple System Dynamics model with an icon legend. The model represents a two-tiered reservoir system; Water flows in to the Upper Reservoir from an outside river, is transferred to the Lower Reservoir at a rate denoted by the "Transfer Rate" variable, and is released from the lower reservoir for municipal use at a rate defined by the "Release Rate" variable. Diagram created using Simile.

While SD models have been used to answer ecological research questions since at least the early 1970s (Sklar and Costanza 1991), there are a number of shortcomings that when addressed could greatly increase their utility. OME primarily focuses on two such deficiencies. The first shortcoming involves the explicit representation of spatial data. A number of SD models utilize explicit spatial relationships as part of their model design (e.g. Ford 1999; Costanza and Voinov 2004). Traditional SD models are inherently spaceless, and it takes

considerable effort to explicitly model spatial relationships within the confines of traditional SD modeling tools (Sklar and Costanza 1991). This leaves it up to the author to construct a spatial representation that is compatible with their model, which can lead to a lot of unnecessary work reinventing the wheel; established and well known spatial relationships have to be redefined and reimplemented for each modeling project (Voinov et al. 2004). By standardizing a common interface for interacting with spatial data, the amount of work required on behalf of the model author would be significantly decreased, and would allow for the reuse of spatial data across models without significant modification being required. To address the possibility of spatial standardization, four spatial usage cases were defined early in the OME design process. The first spatial usage case is that there is no explicit spatial representation whatsoever; this is considered the base case. The second spatial usage case is that the SD model is replicated in each cell in a grid, which in turn represents a spatial coverage. This configuration involves no lateral movement of information between cell models. The third spatial usage case is the same as the second, except there is information sharing across cell boundaries; a flow out of one model's system boundary could act as an inbound flow across an adjacent model's boundary. The fourth spatial usage case is that explicit spatial relationships are defined within a subregion of a SD model. It is this fourth case which was settled upon for OME's spatial interaction with existing SD models.

The second shortcoming that OME is attempting to address is that many of the existing SD modeling tools are largely insular and do not easily couple with other modeling tools. This shortcoming is exemplified by STELLA, one of the more common SD modeling tools (Voinov et al. 2004); other tools surveyed showed varying degrees of modularity and cross-tool communication capabilities, but most of these features did not stretch much beyond the feature set provided by STELLA. By having a SD modeling environment that is readily embeddable in other tools, work on glue code and inter-model communication can be simplified, allowing model authors to spend more time and resources on model details relevant to their research question. Additional flexibility can be derived from adopting a Free and Open Source (FOSS) distribution pattern, which can provide the ability to patch bugs on the fly, expand and customize tools as needed on the finest scale possible, and allow for a researcher to adapt the software to

new and novel runtime environments (von Krogh and von Hippel 2006). The lack of a standardized intermediate representation of SD models can make it difficult to move between tools, or stand as a barrier to understanding between model authors who each work with a different collection of tools; this again is best exemplified through documented experiences working with STELLA when trying to export SD models to other SD model environments (Voinov et al. 2004). Since all SD models have a minimum set of common stock-flow components whose configuration define a functional model (Lane 2008), writing an intermediate file format for storing such models would be beneficial, as it would allow for increased sharing in between tools and authors. There are some broad model specification languages out there, such as Unified Modeling Language (UML), but they are typically too general for the purpose of efficiently representing the common functionality between models originating from different SD tools (Siau and Cao 2001). The Extensible Markup Language (XML) provides a good set of rules that define files that exist as a decent compromise between being program-structure friendly, platform independent, and human readable (Bray et al. 2008). Some tools already support XML representations of their models, but there is no agreed upon representation of models stored as XML between tools (although a candidate does exist; see the Literature Review section) (Costanza and Voinov 2004). By defining a common-ground XML model structure, inter-tool communication will be easier to achieve by avoiding too many tool-specific attributes and entries.

With the aforementioned shortcomings in mind, here are several questions that can be used to measure the success of the OME project:

1. Can an open-sourced, modular SD modeling tool be constructed with performance and features comparable to existing commercial counterparts?
2. Does adding an explicit spatial data interface and adopting a modular design simplify the implementation of spatially-explicit SD models?

The first question stems from a point early in the design process of OME, where the ability to be flexible as possible in OME's implementation was seen as a way to increase the utility of the tool

for model authors and existing modeling projects. As an extension to this thought, it is important for OME to perform in a similar fashion to at least one existing SD modeling tool, and that any inconsistencies can be sufficiently justified by acceptable deviations in the implementations of the different modeling runtimes; minimizing any additional difficulties or obstacles that arise from incorporating OME into a research project will increase the chances of its acceptance amongst the research community. The second question needs to be satisfied in order for the target users (ecological SD modelers) to find any utility in the outcome of this project. Both of these questions will be answered in the context of two existing models whose authors have expressed an interest in the benefits that this project can provide in addressing the previously-outlined shortcomings.

To provide the foundation for further discourse, this thesis will first review previous discussions in the literature surrounding the origins and history of SD modeling, examples of common SD model construction and runtime software, capabilities of common purpose languages to effectively construct and run SD models, previous attempts to bridge the gap between SD modeling and explicit spatial data, and previous attempts at designing a modeling framework in a highly modular fashion. Once a firm foundation for exploration is established, the general design process for OME will be elaborated upon and justified. After the general implementation process is presented, the discussion will focus specifically on the most novel part of the OME project: the implementation of the Spatial Data Provider (SDP) interface. At this point, two reference models originating from Simile will be discussed and the results of performance tests between OME and Simile running these models will be compared. Finally, the overarching discussion will conclude with a demonstration of how OME can increase the utility of an existing SD model when running as a plugin to a large Envision modeling project. Additional implementation details will be provided as a set of appendices for those who are curious about implementation details that exist beyond the scope of the general discussion within this thesis.

Literature Review

Given the wide breadth of material that needs to be covered before a detailed discussion regarding OME can be carried out, this Literature Review will explore a number of seemingly disparate topics that tie together under the umbrella of OME. The discussion in this section will begin by briefly summarizing the origins and history of System Dynamics (SD) models, and their eventual application to spatially-explicit ecosystem modeling. After that, a brief survey of three common contemporary software tools for building and running SD models will be presented. The ability of two programming languages popular in the Scientific modeling and Geospatial Information Systems (GIS) realms, R and Python, to run SD model simulations using previously-published packages for each environment is reviewed. The topic will then shift to examples of spatially explicit ecosystem models implemented as SD models, followed by a discussion of previous attempts to provide a consistent way to explicitly define spatial relationships within SD models. A brief description of Envision follows, which in turn is followed by a discussion of the importance of open source and modular design to the OME project. The discussion will close with a brief description of another modeling tool constructed with a modular architecture, ModCom.

System Dynamics modeling was formally introduced and defined in 1961 in Jay Forrester's *Industrial Dynamics,* with the intent of emulating the flow of materials and resources within an industrial management framework (Forrester 1961). From its inception, System Dynamics modeling was designed to utilize the unique resources of the rapidly expanding computational technology sector to iterate through the dynamics of a model (Forrester 1961). A structured visual syntax has also been defined since the initial introduction of System Dynamics, allowing for a model structure to be communicated completely through visual diagrams (Forrester 1961). As Forrester stated, “Many people visualize interrelationships better when these are shown in a flow diagram than they do from a mere listing of equations” (Forrester 1961); these representative flow diagrams later became known interchangeably as “Forrester Diagrams” (Haefner 1996) and Stock-Flow Diagrams (Lane 2008). The first tool for running SD models, DYNAMO, was introduced at this time as well (Forrester 1961). DYNAMO would take

a series of equations representing the relationships outlined in a stock-flow diagram, check the equations for logical consistency, and, assuming the equations were found to be valid, compile the equations into computer code and carry out the simulation (Forrester 1961).

Forrester's *Urban Dynamics*, published in 1969, expanded System Dynamics from inside the industrial management realm into modeling the dynamics of municipal environments as an aid in urban development policy (Forrester 1969). This successful shift into another problem space supported the argument that System Dynamics was a viable modeling approach for addressing questions across a large spectrum of knowledge spaces. Eventually, SD models would begin to be applied within the Natural Sciences, particularly within ecosystem models (Sklar and Costanza 1991). Given the structure of SD models, any spatial aspects of modeled ecosystems were usually represented implicitly by being bundled into the organization of state variables (Sklar and Costanza 1991). While implicit representation of spatial attributes is sufficient for some ecosystem models, many SD ecosystem models were constructed with explicitly spatial data attributes, a task that could only be accomplished with some difficulty (Sklar and Costanza 1991). To this day the pictorial stock-flow diagram representation of a model has proven appealing to scientists and stakeholders alike for its effective means of communicating relationships in an understandable fashion (Wolstenholme 1982).

Since the introduction of DYNAMO, a number of other SD modeling software tools have been developed. Most of the still-active SD modeling tools provide a method of creating a model through the construction of a modified stock-flow diagram using visual means (Lane 2000). Three common SD visual authoring tools used in ecological modeling are STELLA, Vensim, and Simile.

STELLA was developed by High Performance Systems (now Isee Systems) and was introduced in 1987 as the first SD modeling tool to use a visual approach to constructing stock-flow diagrams as a means to implement models (Haefner 1996). It is still in use today, and is presently being marketed as both an educational learning and research tool (Isee Systems). In addition to its model constructing capabilities, STELLA contains the ability to run model simulations, animate the flow diagram to show the changes in values, and generating results as common graphs and charts (Isee Systems). The model representation is XML-based, and data

can be exported to a Comma-Separated Values (CSV) file (Isee Systems). While STELLA does not have unique tools for constructing explicit spatial relationships within its modeling environment (Costanza and Voinov 2004), Isee Systems does publish a compatible spatial visualizer called Spatial Map, which allows for the graphical representation of exported array data as a two or three-dimensional coverage (Egner 2014). Since Spatial Map is external and strictly interacts only with outputs generated by STELLA (Egner 2014), it does not directly contribute to SD model simulation and is thus limited in its use.

Vensim was developed by Ventana Systems and first released to the public in 1991 (Ventana Systems Inc) and was created out of the need for features that similar tools did not provide (Ford 1999). Like STELLA, Vensim allows for the construction of a SD model using a stock-flow diagram, and provides an environment to run model simulations and present the results and outputs (Ford 1999). Vensim emphasizes its built-in support for the external parameter optimization process, which uses a modified Powell Hill climbing algorithm (Ventana Systems Inc 2014b). Another feature unique to Vensim is the built in ability to produce "Venapps", which are Vensim based models that are bundled within a customized user interface frontend (Ventana Systems Inc 2014c). An interface to accept cross-application control is a standard part of Vensim's API interface and can be accessed with a variety of programming and scripting solutions (including Excel) (Ventana Systems Inc 2014c). Vensim does not appear to have any built in support for explicit spatial representations beyond specific stock-flow configurations constructed by a model's author, but its possible that such a feature may have been overlooked.

Simile was originally developed by a research group at the University of Edinburgh before being spun off as the product of the private company Simulistics Ltd. (Simulistics Ltd.). Simile's visual model construction tools are straightforward and integrated with a simulation runtime which allows for easy browsing of results (Simulistics Ltd.). Models are compiled from C++ code, using either a system-native compiler, or a GNU compiler bundled with Simile's distribution (Simulistics Ltd.). The execution window provides tools for displaying results in a meaningful and accessible way (Simulistics Ltd.), allowing for a simple method of visually verifying specific values at the conclusion of a model simulation run. The model specification

which Simile uses is text-based and fairly easy to understand (Simulistics Ltd.). Simile provides built-in visualization tools which can be used to display spatial data in the form of polygons, square grid cells, or hexagonal grid cells (Simulistics Ltd. 2014c). There are no specific tools provided by Simile designed for representing explicit spatial relationships, but such relationships can be constructed by the model author through the unique configuration of compartment, flow and submodel components (Simulistics Ltd. 2014c). Simulistics recommends that association submodels be used to map out spatial relationships (Simulistics Ltd.). Association submodels are those that can map out conditional relationships between two other submodels through the association via role arrow model components, optionally using specific selection criteria (Simulistics Ltd.). While this approach may be effective, it has been acknowledged that such an approach has a significant overhead on behalf of model authors in understanding the association construct and details of its implementation conceptually and practically (Simulistics Ltd.).

In addition to specific SD tools, System Dynamics has also been modeled using high level languages. Two such languages which have a strong presence in the Natural Sciences are R and Python (Perez, Granger, and Hunter 2011). R is a language and runtime environment designed for statistical computing and generating high-quality plots and graphs (R Foundation 2014a). The R environment is extensible and provides built in access to a module archive known as the Comprehensive R Archive Network (CRAN) (Adler 2009). Within the CRAN, there are at least two modules that have been used for building and running System Dynamics model simulations: deSolve and simecol (Petzoldt). DeSolve is a package of general equation solvers for Ordinary Differential Equations (ODEs), Partial Differential Equations (PDEs), Differential Algebraic Equations (DAEs), and Delay Differential Equations (DDEs) (Soetaert, Petzoldt, and Setzer 2010). Simecol is a framework for building ecological model simulations in R and relies on the deSolve module for solving simulation equations (Petzoldt and Rinke 2007). While it is possible to write and run SD models in R, it is not necessarily an ideal environment to do so. R is not efficient at querying data structures, handling complex data structures, or handling a data set that is larger than the available space in memory (Adler 2009), all conditions that can arise from particularly complex SD models as observed throughout the development of OME. Additionally, R is an interpreted language (R Foundation 2014b), meaning that there is additional performance

overhead using a model written to run in its runtime environment as opposed to a language which is compiled directly into machine code, such as C++.

Python is a high-level interpreted language and runtime environment with a solid focus on a clear and accessible syntax (The Python Foundation 2014a). Like R, Python has support for extension modules, most of which are accessible on the Python Package Index (PyPi) (The Python Foundation 2014b). There has been a wide-scale adoption of Python across Geographic Information Systems (GIS) centered disciplines due to its adoption as the primary scripting language in ArcGIS (Perez, Granger, and Hunter 2011); Python has seen adoption across general science disciplines as well (Environmental Systems Research Institute 2006). Presently, several Python modules exist which provide the capability to run System Dynamics model simulations; PyDSTool and SimPy are notable examples of such packages. PyDSTool is a Python package that focuses on running SD models, particularly those used in biological applications, as well as supplying some visualization and optimization tools (Clewley 2012). SimPy is a package for running general discrete event-driven simulations, a classification which includes System Dynamics models (Lünsdorf and Scherfke 2014). While its more general nature and its better data handling abilities may make it a more desirable environment for running SD simulations than R, its still fundamentally an interpreted language which leads to additional overhead and decreasing performance when compared to compiled language (The Python Foundation 2014a). It is possible to write performance critical sections in machine native code and link it to Python through a native interface (Oliphant 2007), but this means that the core functionality is not written in Python at all, but instead in some language that compiles into machine native code (such as C++).

Spatially explicit SD models have been in use from at least the early 1970s (Sklar and Costanza 1991). Several examples of spatially explicit research models using the Spatial Modeling Environment (SME) appear in a collection of case studies in the book *Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach* (Costanza and Voinov 2004). SME is a project which attempts to apply explicit spatial relationships to STELLA by taking output from STELLA, converting it to C++ code, compiling it, and running the resulting binary in a custom runtime that executes the SD model as a unit model copied across a spatially explicit

grid (Costanza and Voinov 2004). One model which utilizes SME in this fashion, The Great Bay Model, uses a unit model developed in STELLA to capture dynamics of carbon flow through Eelgrass-centered ecosystems in the Great Bay estuary in New Hampshire for a fixed cell size ($100x100m^2$) (Behm, Boumans, and Short 2004). Each cell has a copy of the unit model initialized with cell-specific values, with flux values from neighboring cells being added by SME to represent the flow of nutrients, detritus, and consumers, satisfying spatial usage case three as defined in the Introduction (Behm, Boumans, and Short 2004). Another implementation that uses SME is a desert tortoise population model by Aycrigg et al., which simulates the impacts of six different land management scenarios on desert tortoise populations in the Mojave Desert (Aycrigg, Harper, and Westervelt 2004). The unit model simulates climate, vegetation, and tortoise population dynamics for fixed-sized cells ($1x1km^2$); copies of the unit model are mapped to cells in a gridded spatial coverage by SME (Aycrigg, Harper, and Westervelt 2004). Similar to the Great Bay Model, unit models in neighboring cells can contribute to one another, simulating tortoise migration, which also satisfies spatial usage case three (Aycrigg, Harper, and Westervelt 2004). A different spatial usage case can be seen in the two models that are described in the Simile Model Compatibility section, which express their explicit spatial components as a subportion of a much larger model (spatial case number four).

The need for a unified external representation of spatially-explicit relationships has been recognized and previous attempts have been made to address it. SME is one such project whose approach parallels that of OME. Since lateral transfer of values between cells and their unit model representations is supported, SME satisfies both spatial cases two and three, but most models appear to focus on the latter case (Costanza and Voinov 2004). Another approach to fulfill this niche is represented through SimARC. SimARC is a bridge between Simile and ArcMap (a common GIS tool) that allows ArcMap to call a Simile's runtime environment to run a compiled Simile model on each polygon within an ArcMap coverage, producing a new map layer (Mazzoleni et al. 2003). Since no lateral transfer is supported between polygons, SimARC can only satisfy spatial case two (Mazzoleni et al.). While both SME and SimARC are addressing a similar need to OME, they differ rather significantly in a few details. The two implementations focus on a different spatial application than OME. Specifically, they are largely

concerned with replicating System Dynamics across discrete units within a spatial coverage, with SME typically being applied across a grid/raster coverage (Costanza and Voinov 2004), and SimARC being applied across a polygon-based coverage (Mazzoleni et al. 2003). OME, in its current implementation, focuses its spatial data management toward applications where a SD model incorporates spatial information into its larger simulation space. As previously mentioned, SME primarily satisfies spatial case three, SimARC only applies to spatial case two, and OME (in its present incarnation) focuses on spatial case four. The target auxiliary tools also differ; SME largely relies on STELLA (Costanza and Voinov 2004). while OME is currently targeting Simile for its input source (both of these tools are designed with the possibility to use other SD modeling tools as input sources (Costanza and Voinov 2004)). SimARC uses Simile as its input, and its approach is to embed the model within another tool (Mazzoleni et al. 2003). OME, by contrast, is designed to be flexible enough to use different SD tools as an input, and to both run standalone and embedded in another tool.

An important objective of the OME project is to demonstrate SD models running as a subcomponent of a much larger simulation framework; Envision was chosen as the target platform for this task. Envision is an agent-based, spatially explicit integrated modeling platform developed at Oregon State University (Oregon State University). Envision focuses on simulations revolving around coupled human and natural systems under the projected effects of Global Climate Change (Oregon State University). High level dynamics are driven by the interactions between agents (decision making units for land parcels), landscapes (the spatially explicit coverage which reflects changes during a simulation), and policies (constraints on simulation behaviors) (Oregon State University). As a collaborative modeling platform, Envision utilizes a plugin architecture that allows for the addition and usage of different models, visualizers, and outputs on a per-model basis (Bolte 2014). Projects involving Envision typically focus on futures projections of climate and resource trends with the goal of assisting in policy decisions (Oregon State University 2014b). Further exploration of Envision as a runtime environment will be explored in the Case Studies section.

Another objective is to provide the components of OME in an Open Source distribution with robust documentation alongside precompiled binaries. Such a configuration can allow for a

higher degree of scrutiny and custom modifications on behalf of interested parties (Gwebu and Wang 2011). In a similar vein, many of the extant SD modeling tools appear to have varying degrees to which they can interchange or embed in other modeling environments (Simulistics Ltd.; Ventana Systems Inc 2014b; Isee Systems). OME's initial implementation is designed to optionally embed in an Envision project as an autonomous process. This configuration, with Envision being a platform designed to run multiple modeling plugins within the same simulation, allows OME to accept inputs from and output to Envision's modeling context, allowing cross-communication between other modeling components (Oregon State University 2014b). This cross tool integration with Envision could potentially be extended to other environments as well.

Modular ecosystem modeling framework design has been previously explored, with ModCom being one of the more recent examples (Hillyer et al. 2003). ModCom is an ecosystem modeling framework which provides a skeleton for constructing models out of modular components (Hillyer et al. 2003). The central library to ModCom's framework, ModComLib, provides interfaces that outline conformation requirements for both core modules and custom external modules (Hillyer et al. 2003). Cross-module communication is accomplished through the Component Object Model (COM) specification, which is an implementation-independent method for communicating between binary applications (Hillyer et al. 2003). Each component exposes inputs and outputs used over the course of a simulation through a series of standardized interfaces (Hillyer et al. 2003). The interfaces include ISimObj (basic ModCom object interface), IUpdateable (interface for periodic update messages), and IODEProvider (interface for objects represented by one or more ordinary differential equations) (Hillyer et al. 2003). This setup allows components to be chained together with the outputs of one or more components feeding into the inputs of additional components; such a design allows for the addition, removal, or swapping of components with minimal effort (Hillyer et al. 2003). Additional interfaces exist for exposing data, inter-object communication, and reading and writing data files (Hillyer et al. 2003). ModCom and its source code were released as an open source project alongside visual tools for constructing additional ModCom components (Hillyer et al. 2003). Both the modular nature of ModCom and its open-source distribution are aspects that would likely prove beneficial to OME as well.

To coincide with a modular and open structure of the program itself, it is important to explore the equivalent construct for storing model details as files. There are a number of standard file format specifications that are used to store SD models for use and distribution. A recent entry, XMILE (XML Modeling Interchange Language) seeks to provide a universal format to be used across the various SD modeling tools that presently exist (OASIS). XMILE is an XML-based file format that is intended to share model definition, drawing, and runtime information between different SD modeling environments (OASIS). The data stored in a XMILE file is divided into three conceptual layers known as compliance levels, with each successive level building upon the previous (Chichakly 2013). The intention behind each compliance level is to provide a roadmap for what specific feature sets a given tool should support, with support of a higher level layer implying support of all levels below it (Chichakly 2013). The first compliance level (Simulation) contains the bare equations needed to run the simulation (Chichakly 2013). The second compliance level (Display) stores all the information necessary to display and modify the individual model components which define the equations in the first level (Chichakly 2013). The third compliance level (Interface) provides information on running the model and any auxiliary information that can be used to describe how to display the results (Chichakly 2013). While the compliance levels are largely conceptual, the levels are practically incorporated into XMILE files as an attribute to the root node of the file (Chichakly, 2013). The architecture of the file itself is conceptually constructed as three sections: Model, Presentation, and Widgets (Chichakly 2013). The model section is synonymous with the Simulation layer; that is, the Model section contains all the information necessary to execute the model (Chichakly 2013). The Presentation section encompasses all the needs of the Display layer and portions of the Interface layer, and is concerned with all aspects of lower level drawing requirements (Chichakly 2013). The Widgets section covers any details of the Interface layer which are not covered by the Presentation section, namely higher-level constructs such as user-interface widgets, graphs, and tables (Chichakly 2013). The format is presently undergoing ratification by the OASIS standards organization in order to be acknowledged as an official standardized interchange file format (OASIS). The degree of detail that XMILE covers is greater than what is presently needed by

OME, and was therefore bypassed in favor of a simpler XML-based model format that has been in use since the early stages of development.

As has been demonstrated, there has been much previously published work that explores the various facets that the development of OME will attempt to cover. Touching upon all of these seemingly disparate branches of research and development has provided sufficient background to continue the discussion by describing the reasons, decisions, and obstacles that were used to shape OME's overall design approach.

Design Approach

As previously stated, OME is designed to provide a universal interface for querying explicit spatial relationships, and to provide a modular configuration for coupling with other tools. To achieve these goals, much planning and deliberation went in to the OME development process. Milestones were defined to guide development, and both Simile and Envision were incorporated into the project both as tools to assist in development and as targets for the application of OME. Decisions regarding how to run model simulations, what external technologies and third-party libraries to rely upon, and what platforms to target were made as needed. The OME development process has been long and complex, and is by no means complete. This is best exemplified by the series of milestones that have been established.

Milestones and Useful External Tools

Goal	Implementation
Load and run pre-existing SD models.	Conversion tools for modifying and copying Simile files into OME-ready files.
Support for explicit spatial representation.	Create a universal spatial data provider (SDP) interface, and provide a syntax for querying spatial relationships within model component expressions.
Flexibility in design.	Follow principles of encapsulation and strive for modularity.
Be both capable of running standalone and as a plugin.	Adhere to principles of modular design.
Cross-platform support.	Use platform-agnostic code wherever possible, and isolate platform specific code using preprocessor macros as necessary.
Comfortable user-interface for any tools that require a GUI.	Use platform-native libraries for generating UI components instead of cross platform libraries. Users on each platform have different expectations about available UI conventions.

Table 1. Overview of design goals and implementation approaches.

When beginning the OME project, it was quickly realized that there would be a lot of unknowns in the process, and that something useful should be produced even if none of the original end goals were achieved. Table 1 provides an overview of some of the broad goals of this project and the attempted methods of implementation. Thus, incremental goals were defined as milestones that defined points in the development where a set of known features would be

implemented and functional enough to be utilized by future researchers or developers; these milestone features would stand on their own even if the initial development of OME had ceased. Milestones were initially set by identifying the desired end goals and working backward along the proposed development timeline. Intermediate points, where subsets of useful features would be considered sufficiently complete for widespread use, were identified and marked as milestones. As development progressed and unforeseen complications were encountered, achieved milestones were noted, while future milestones were added, adjusted, or eliminated as needed. The initial set of milestones consisted of the following: 1) Implement a tool which can convert Simile models into a readily accessible intermediate format; 2) implement a simulation runtime that can handle simple, statically-defined model dynamics using an Euler or RK4-based solver; 3) add a scripting or extension language to the simulation engine that provides support for complex data containers; 4) increase the complexity of supported model dynamics; 5) provide initial spatial data support via an explicit interface; 6) implement an Envision plugin using the explicit spatial data interface; 7) add support to the Simile model converter for adapting model logic for use with the spatial data interface; 8) add support for running some runtime processes in parallel; 9) build a model construction suite that builds SD models directly for OME through the use of standard stock-flow diagram iconography. The present implementation of OME only satisfies the milestones up through milestone 7. Additional milestones were added as the development focus was adjusted over time; the additional milestones were 4.5) Implement model logic that can be compiled into machine code; 6.5) write a straightforward tool for running, dumping, and visually parsing the results of an OME simulation run; and 7.5) port the existing OME components to at least one other platform. Most of the secondary milestones have been satisfied, with a tool for generating source code for compiled models (the ModelClassBuilder project), the creation of OMESimRunner, and the reimplementation of the core OME coding projects under Mac OS X. Milestones 8 and 9 were not reached simply due to time constraints on the initial implementation stage of this project, and will be noted as future points of expansion for any attempts to expand upon the present OME implementation.

As indicated by the aforementioned milestones, both Simile and Envision played a role in the development process, primarily as tools to assist in the validation of the results of OME

simulation runs. Simile had a number of features which proved beneficial throughout the OME development process. Simile's visual model construction tools are straightforward and integrated with a simulation runtime which allows for easy browsing of results (Simulistics Ltd.). Models are compiled from C++ declarations (Simulistics Ltd.), which gives the OME compiled libraries (also compiled from C++ code) a reasonable baseline to measure performance against. Simile's execution window provides a variety of tools for displaying results (Simulistics Ltd.), allowing for values to be visually inspected and compared to those produced by OME when both environments are effectively running the same model. The model file specification which Simile uses is text-based and fairly easy to understand (Simulistics Ltd.), making it a good first target for a conversion tool. As more complex models were incorporated into the development process, Simile's snapshot tool proved essential, as it allowed for a multi-instanced model component to have all of its values for a given time step exported to a CSV file. Data exported from Simile could then be used as a reference point for comparing results between a model run in Simile and a model run under OME. Envision was useful for further vetting OME's model simulation results by providing a visual representation of a model's spatial coverage. By directing important values to be displayed by the map layer constructed in Envision, the correctness of OME's runtime could be checked on a high level by displaying any significant deviations from expected patterns and providing a map to specific submodel instances that were having issues. For more information about mapping values between OME's runtime and a larger coupled modeling environment (such as Envision), see the SDP Implementation section.

OME Model Representations

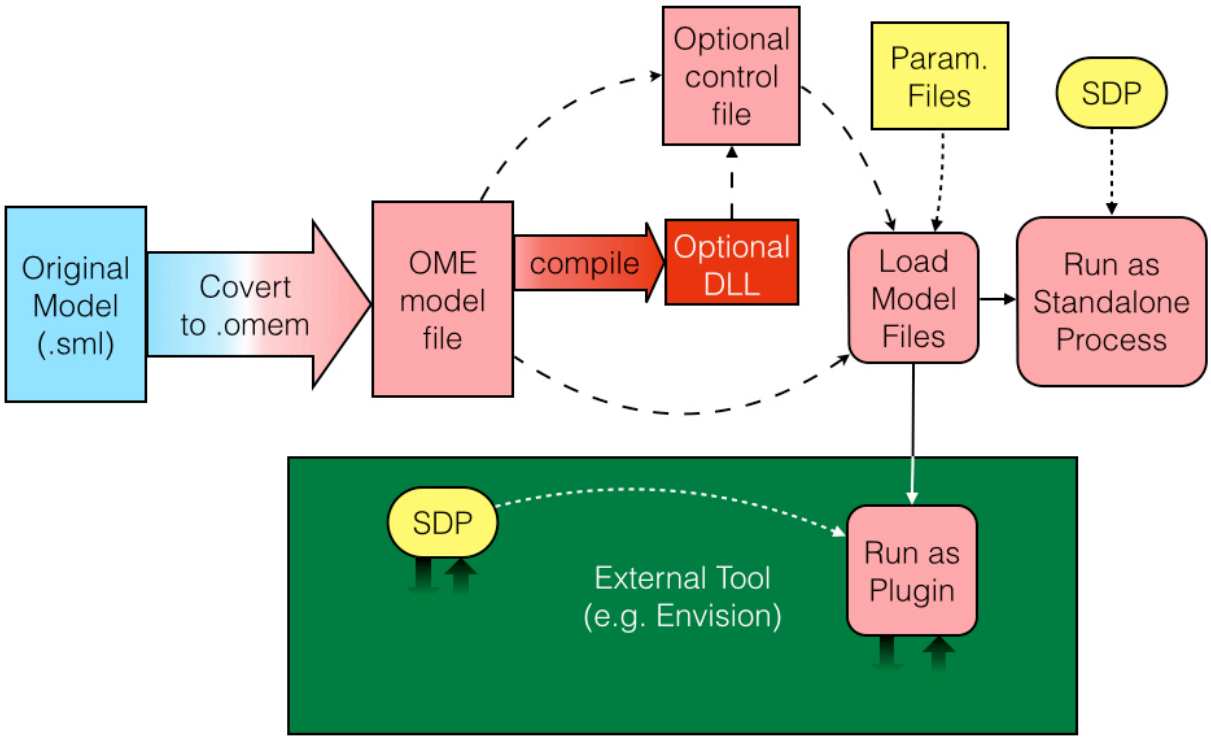

Figure 2. Overview of the model execution process. Filled arrows represent conversion steps from one format to another. Solid arrows are required steps in the execution process, while dotted arrows represent optional linkages between external components and their target processes. The dashed arrows represent the three ways an OME model can be loaded into the runtime, one of which must be taken: load the model file directly into the runtime, load a control file that points directly to the model file, or load a control file that points to a compiled version of the target model.

The modular nature of the Object Oriented Programming (OOP) design was relied upon substantially to explore a number of potential extension languages that could be used to evaluate model component expressions. Extension languages are designed to be embedded in larger programs to handle subsets of logic that tend to be highly volatile or modifiable during runtime (Ierusalimschy, de Figueiredo, and Celes 2007). An additional benefit of using an embedded extension language is the ability to write and test or debug runtime logic while the parent program is running. Several extension languages were experimented with until a satisfactory

candidate was found. MTParser (Jacques 2004) was the first extension language to be evaluated because of its speed and relative simplicity. Unfortunately, MTParser's logic was found to be limited to non-collection values, and it did not have a cross platform implementation. A similar extension language implementation, muparserx (Berg 2005), was the next to be assessed, and while it did address the shortcomings of MTParser, it was found lacking in both flexibility and performance. Finally, Lua (Ierusalimschy, de Figueiredo, and Celes 2007) was evaluated and found to be acceptable for implementing OME's model component expressions. Lua is an open-source language implementation intended to act as an embedded extension language in larger C/C++ programs (Ierusalimschy, de Figueiredo, and Celes 2007). The runtime engine is easy to compile as a self-contained module, while the actual language itself is both well documented and mature (Hirschi 2007). An interface for OME model components was written which allowed Lua to interpret and execute model component update expressions. Additionally, a debugging prompt was implemented to query model components and their states during breaks in runtime, greatly assisting in the search for runtime bugs that were introduced during the development of the Lua interface. Despite the advantages of using Lua as part of the model evaluation process, performance limits were too great to overcome for larger models, so an alternative approach was devised which relied on compiled code instead.

Constructing model update logic as a binary library results in faster code at the cost of flexibility. To create a binary model logic file, a model is run through a serialization tool which produces a C++ class encompassing all initialization and update logic. The generated source code must then be compiled as a dynamic library and linked against the OME support libraries (namely OMERuntime and OMEDraw). An optional declaration in the Model's control file is then used to locate the compiled library and instruct the OME engine to load it at runtime. The expression statements from model components are generally less structured than formal C++ statements; to reconcile the two levels of structure an intermediate wrapper layer was developed to simplify the process of model serialization without sacrificing much performance. Objects pertaining to the wrapper layer are rapidly created and discarded; to prevent memory fragmentation, several instances of a custom memory pool class were used. A memory pool is a well known memory construct; it is a block of memory that is allocated by a process and is

divided into equal sized partitions that are continuously reused. The memory pool implementation in OME is advantageous for the large amounts of short-lived fixed sized objects which are constantly being created and destroyed, since retrieving a block of memory for a new instantiation occurs in constant time and memory fragmentation is reduced across the system since a memory pool only requests memory from the operating system when it needs to be resized. Ideally, the serialization process would be sophisticated enough to remove any ambiguity so that a wrapper layer would be unnecessary, but development time constraints limited how much work could be done in this direction. Nevertheless, the conversion/serialization code is largely encapsulated; such a future enhancement could be made without requiring an extensive rewrite of the source code. By converting the model into C++ code, the author of the model can rely on their C++ development environment to supply a sophisticated debugging tool (such as Visual Studio's debugger or GDB), and to thoroughly optimize the model logic during the building of a copy intended for distribution. Unfortunately, this approach requires an external compiler (such as Microsoft's VC++ compiler, Clang, or gcc/g++) and compilation for each target operating system and hardware architecture. Fortunately, nearly all of the non-user interface code is platform agnostic, and generated C++ based model logic should compile with little or no effort under any C++ compiler with support for the C++11 standard.

XML was decided upon as the basis for the model specification, as it is both universally supported and relatively straightforward for others to read, understand, and process regardless of the chosen programming language (Bishop and Horspool 2006). As stated in the Literature Review, the open XMILE standard was skipped in favor of a custom XML-based format mostly due to differences in complexity; XMILE is much more suitable for a fully comprehensive modeling tool, and as such providing support even for just its first conceptual layer would be more time consuming and involved than what was settled upon at this point in the OME project. Nevertheless, it would be beneficial for future iterations of OME to support the XMILE file specification to some degree, and perhaps eventually use it as a complete replacement for OME's custom formats.

Presently, there are two XML-based file types that have been created specifically for

OME (parameter values can be loaded from general CSV files or Simile's .spf files). The first (.omem) is an XML file which stores the complete model specification, while the second type (.omec) is an XML file which handles details for controlling simulation runs, mapping specific coverage values to objects through the Spatial Data Provider interface, managing the output of debugging information, and any other details that are important to simulation conditions but not specific to the model definition. A third file type (.omet) has also been defined, but it is a binary file intended to act as a temporary file during a simulation run and is not intended to be distributed. See Appendix C for an overview of the structure of the .omec and .omem XML files.

All of the aforementioned details factor into the general OME model preparation and execution process, as outlined in Figure 2. Essentially, a model must be converted from its native format to OME's model format. This file (or its compiled representation) is then loaded into the runtime directly, or (preferably) though the inclusion of an OME control file that is loaded into the runtime instead. Optional parameter files are loaded into the engine at this time as well. If OME is running as a standalone product, then a Spatial Data Provider (SDP) is optionally included and the model is executed. If OME is running as a plugin, the parent tool is responsible for linking in an optional SDP interface, configuring the data exchange between itself and OME, and telling OME to execute at the necessary intervals. The divergence in behavior is possible due to the modularity inherent in OME's design.

Modularity and Interoperability

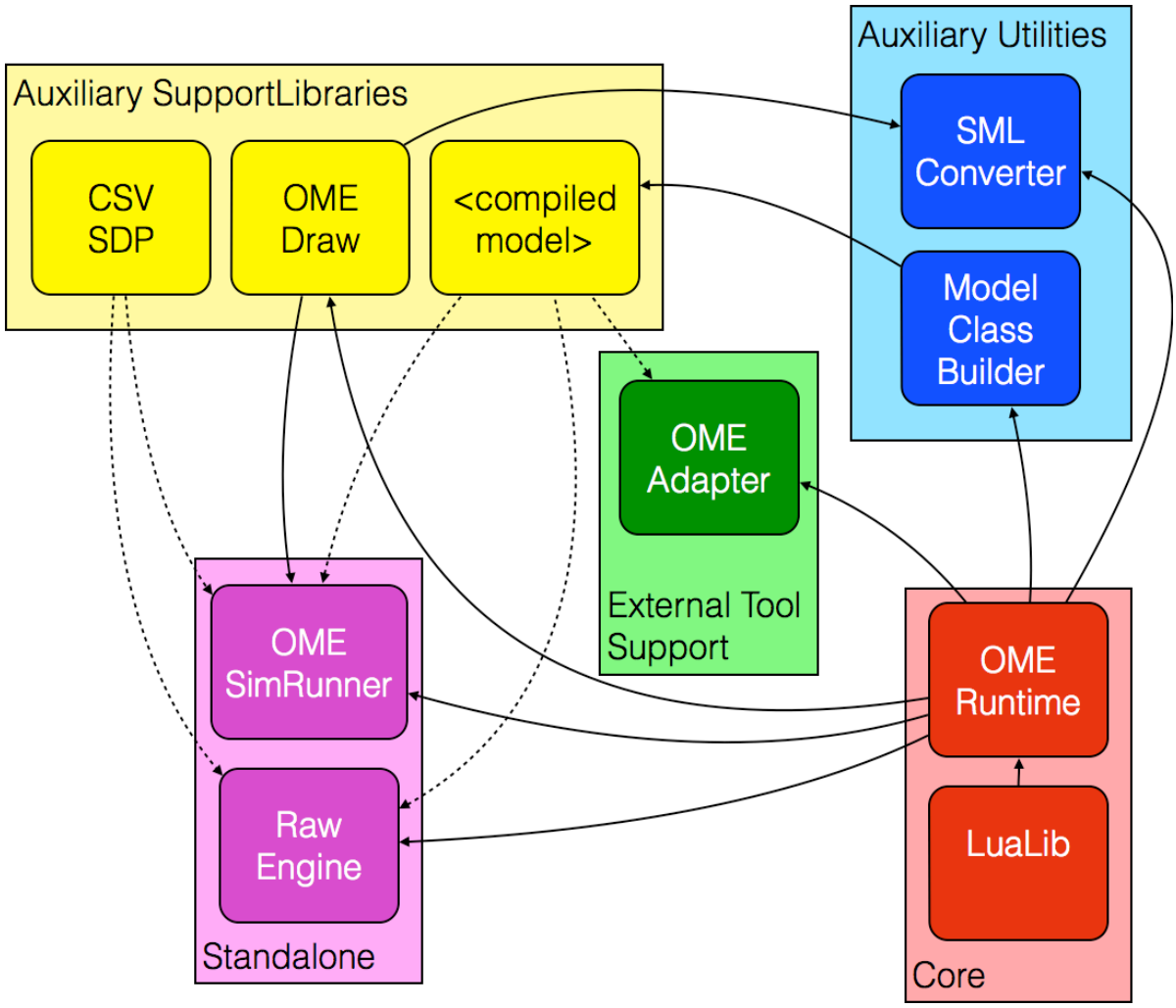

Figure 3. OME architecture and module boundaries for the dynamic libraries and executables that are presently part of the OME distribution. Solid arrows refer to required dependencies, while dotted arrows refer to optional dependencies. The box <compiled model> refers to a dynamic library generated from a given author's model.

OME attempts to address the issue of little or no interoperability between existing SD modeling tools by emphasizing a modular design approach throughout its architecture. The design of OME is constructed following an object oriented programming (OOP) approach. The OOP approach is a paradigm whose defining characteristics appear to fit well with the conceptual design of a SD model. The core conceptual structure in OOP is the object, which is a

discrete entity with distinct attributes and behaviors (Wegner 1990). Each object has a defined interface through which other objects can interact with, the idea being that details of an object's implementation should not need to be known or understood by other interacting objects (Wegner 1990). This separation between the interface and the implementation encourages modularity, and allow for the concepts of encapsulation (protecting internal data by forcing interaction through a known interface with the knowledge that the implementation fulfills a particular relationship) (Wegner 1990) and inheritance (creating a new object by extending the attributes and behaviors of another extant object) (Wegner 1990); both of these concepts were used extensively when implementing OME's model representation, runtime logic, and support code. The conceptual interpretation of SD models lends itself to object-oriented design since the model is made up of discrete components (Compartments, Flows, Variables, Influences, etc.) each of which contain attributes and behaviors unique to their roles within the diagram. Modular design has also influenced the overall development process in a few particular ways. Designing different components with clearly delineated boundaries has allowed for progress to be made in incremental steps; pieces of the engine can be initially implemented with moderately sufficient solutions, and replaced later with more elegant solutions without requiring an extensive rewrite of the codebase at large. This implement-and-replace process was beneficial during the development of the compiled model structure and the higher-level runtime management. The other way modularity has been used has been in the design of the runtime itself; the library containing all core runtime components (OMERuntime) has no user-facing front end. Instead, the library has its manager class exposed, which handles all the details of loading and running an OME-compatible model, resulting in a runtime that is completely decoupled from any specific front end. Such a modular design allows for the engine to run independently, or as an extension to another modeling tool. Model execution is driven by a central event loop, which allows outside objects to be registered to receive specific broadcast events, and to submit events of their own. This configuration allows for outside programs to hook in to the model runtime and receive feedback at regular intervals, or when specific events are dispatched. When OME is configured to run as a plugin, there is an interface exposed that is designed to regulate what information can move into and out of the model at regular intervals as defined by the parent program. The

coarsest level of modularity involves the division of OME into different libraries and executables, as shown in Figure 3. This high level modularity also allows for some of the modules to be optionally included or replaced at runtime, either through the use of configuration files, or dynamic library linking.

The ability to extend Envision's functionality, as well as its spatial coverage management capabilities, makes it an ideal platform to demonstrate OME's plugin configuration when coupled with a parent program. In the case of Envision, the parent program is both controlling the SD model execution, and acts as OME's spatial data provider. A SD model's integration into an Envision project opens up the possibility of a SD simulation interacting with other model simulations that are running on the same spatial coverage at the same time. This allows for different models to represent different sets of processes which can then exchange information with one another. OME's runtime is adapted for use by being linked into another dynamic library known as OMEAdapter. Using Envision's EnvExtension base class, OMEAdapter provides the necessary functions and setup code for OME to be treated as an autonomous process; an autonomous process is a process that carries out its assigned task outside the bounds of the central processes that make up the simulated human-decision making portion of the overall simulation in Envision (Bolte 2014). An autonomous process entry in an Envision project (.envx) file is then used to link in OME along with a .omec file to provide OME-specific instructions.

To provide a spatial representation method that can potentially apply to all four of the usage cases outlined in the Introduction, OME implements an interface for querying about information regarding a coverage's spatial details. Requests through the interface are fulfilled by a backend implementation that varies based on the conditions under which OME is executed. For development purposes, two interface backends have been created; the first implementation is designed to read standalone CSV files, and is implemented as a plugin which is referenced by a model's control file. The second implementation uses Envision as the interface backend, and is used when OME is run as a plugin to Envision. The spatial interface is clearly delineated so that future implementations can be developed and applied with little trouble; templates for dynamic library hooks have also been provided so that plugin authors will be aware of which function signatures will be required for the library linking to work. Further details regarding the

implementation of a spatial data interface is covered in the SDP Implementation section.

An important focus of this project is its open-source approach to development and distribution. Open-Source Software (OSS) is a management approach that encourages communal improvement of a software product by requiring the release of the source code alongside the product, and licensing the source code in a way that allows open modification (Open Source Initiative 1998). An OSS approach has a number of advantages that will encourage uptake and maintenance of the OME environment. OSS distribution meshes well with scientific research since publishing source code allows for transparency and critical scrutiny, both of which are crucial to the scientific process (Bangerth and Heister 2013). Since OME's target audiences are scientific researchers and modelers, it would make sense to follow a communication paradigm that closely parallels one that these audiences have confidence in. The transparency and scrutiny that an open-source project is subject to can increase the chances that potential errors or vulnerabilities will be identified due to the likelihood that the source code is being audited by a large group of individuals. Such openness towards auditing is an advantage over closed-source code that can further increase confidence in results generated by OME (Gwebu and Wang 2011). Similarly, the ability for anyone to contribute to the code can allow for specialists to apply their unique knowledge to specific portions of OME's codebase to increase accuracy, performance, and resource management as they see fit. Not all potential changes will be beneficial to everyone; one advantage of open-source distributions is that they are often suitable for forking, or copying a source code project at a given point of development to continue with a different set of goals and objectives than the those guiding the originating project (Ruparelia 2010). The ability to specialize on such a low level is often unavailable in closed-source tools, but is oftentimes encouraged by open-source communities (such as github) (Gwebu and Wang 2011; Ruparelia 2010). Open-source development is frequently accompanied by cross-platform development (Borshchev and Filippov 2004). While OME has been initially developed under Windows, platform agnostic code and libraries have been used wherever possible and where it doesn't impact the user interface; this has made it straightforward to expand development to Mac OS X and Linux environments. By having the tool exist across multiple platforms, the pool of potential users is expanded, increasing the chances of uptake and acceptance of the model

development community as a whole (Ierusalimschy, de Figueiredo, and Celes 2007). Having the source code published in its entirety opens this project up to a longer active lifespan by allowing maintenance responsibilities to freely change hands, and by allowing it to be incorporated into other projects where it can be of some use.

Simply publishing a project in an open source format is not enough to gain acceptance by a community that is actively engaged and interested in providing their own contributions (Bangerth and Heister 2013). The most obvious initial step is to host the project in a source code repository that is both easily accessible and allows for project forking or branching; most popular version control systems allow for (and in some cases, actively encourage) project branching (Ruparelia 2010). Branching allows for a contributor to focus their resources in a direction that does not impact other users; if any changes or modifications are deemed valuable, they can be merged back into the main branch (Ruparelia 2010). Branching differs from forking in that branching is an alternate line in the same repository as its originating branch (Ruparelia 2010), while forking results in a completely new repository and development project (Robles and González-Barahona 2012). Another practice that will encourage uptake and acceptance is solid documentation throughout the entire project; most people are more inclined to engage in an open-source project if they can easily understand how everything works together (Bangerth and Heister 2013). Documentation on several levels is important: the source code level (individual functions, objects, and single lines of code), the API level (what behaviors other tools can access and how), and higher usage considerations (model and runtime configuration details). All of the aforementioned levels of documentation are being authored alongside the source code development for OME. Tools such as Doxygen are being used to streamline the process of generating source code documentation from inline source code comment blocks, reducing the overall effort required to produce meaningful developer documentation. Good adherence to modular design allows for adjustments to one portion of the source code without interfering with another, and is a widespread technique used for managing complexity in most large programming projects (Bangerth and Heister 2013). Such design also allows for a contributor to swap in their chosen solution to a specific subsection of the entire project, such as alternate implementations of memory management schemes or common search algorithms; this has been

demonstrated through the development of OME's custom management schemes and the various iterations of expression language support schemes. Finally, by actively engaging with the SD modeling community, it is believed that interest can be stirred by both showing off the existing features and demonstrating a responsiveness to their needs and requirements. This process has already begun with the interaction with some model authors.

Platform Targets and 3rd Party Libraries

C++ was chosen for this project over other higher level languages commonly used in scientific programming (notably Python, Java and C#) for several reasons. Personal experience has shown that C++ scales well from small to large projects, and across various degrees of abstraction (from the bit level through high level constructs). This wide range of the language's application space allows for a project to grow in complexity and abstraction incrementally over time, which combines nicely with OME's milestone-based development approach. C++ is well established across a number of hardware and software platforms, with a consistent set of implementation standards shared across multiple compilers (ISO/IEC 2011). The C++ compilers being used in the OME development process are well calibrated for their target environments and are capable of applying a number of performance and/or space optimizations during the compilation process without the intervention of a programmer. No additional runtime layer is needed for C++ programs, reducing the performance overhead and giving more control over to the programmer. While the lack of a runtime layer means that a programmer is required to handle the burden of implementing their own memory management scheme when programming in pure C or C++, it has the advantage of allowing the programmer to know at all times how memory is being handled, and to avoid large memory overheads in memory intensive processes (such as running a complex SD model) (Hertz and Berger 2005). Finally, there is simply a massive number of mature, well documented, and well supported support libraries that exist for C++ due to its age and standing within the software development community. Experience with C++ has shown that it is a language that is very sensitive to the discipline of an author's programming practices, and most of the pitfalls associated with C++ can be avoided by strictly

adhering to good ones. It is through the use of good programming practices that we hope will make OME's source code flexible, approachable, and readable.

While the C++ Standard Template Library (STL) was utilized for various data structures and containers, several third party libraries were employed for various subtasks. The were a number of criteria that were considered when making the decision to include third-party libraries. The first criterion was whether or not the library had a cross-platform implementation; the intention is to avoid any unnecessary hurdles in expanding OME to other platforms. The second criterion is whether or not the library is simple and focused on a singular task. The less a library attempts to take on, the easier it is to understand and troubleshoot. The third criterion is whether or not the library comes as a source code distribution. This is important for debugging since it allows for the tracing of library logic, and if necessary, allows for customizations to be made. The final criterion is whether or not the license for the library is compatible with the distribution plan for OME. This is out of respect for the authors of these libraries and their wishes in how they are used. With these four criteria in mind, three third-party libraries are included in the distribution. The first, Lua (http://www.lua.org), has been previously discussed; it is distributed as source code, has a straightforward application programming interface (API), and an OME-compatible license. The second library is TinyXML-2 (http://www.grinninglizard.com/tinyxml2/), a simple, lightweight library for parsing and writing XML-compliant files (Thomason 2014). TinyXML-2 eschews some of the more advanced XML features, such as document type definitions (DTDs) and extensible stylesheet language (XSL) support, for the sake of simplicity (Thomason 2014). TinyXML-2 is distributed as source code under the zlib license (Thomason 2014), which is compatible with OME's distribution. The final third-party library is Shiny (https://code.google.com/p/shinyprofiler/), a compact C++ and Lua profiler (Abedi 2007). Shiny relies on preprocessor macros to poll specific points within a target program's source code; while this means that the target source code must be modified and recompiled every time a new portion of code needs to be profiled, it also means that all profiling code can be disabled by setting the SHINY_PROFILER macro to FALSE during compilation (Abedi 2007). The assembled polling data is collected in simple human-readable text files(Abedi 2007), which can then be read by a developer, or parsed with an external tool (the OME

distribution comes with a series of Python scripts used to generate performance graphs from Shiny's output). Shiny is distributed as source code and project makefiles under the MIT license, which is compatible with the needs of the OME project.

Writing OME in C++ means that the project needs to be compiled for each platform that it is run on, but this is not a large hurdle. As previously mentioned, nearly all of the non-user interface code is written in a platform-agnostic fashion, relying on the STL and the C++11 standard. There are few places where platform-specific code is required, specifically with runtime loading of dynamically linked libraries, and a handful of places where a function needs to communicate directly with the operating system (such as evaluating file paths). In these cases, the proper code for each environment is selected either through the use of preprocessor macros, or the inclusion of alternate source code files within project files and/or makefiles. Code concerned with user interface organization is the exception; it was decided during the design process that it would be best to use a platform's native user interface (UI) framework to encourage the specific look-and-feel associated with that environment, at the cost of some increase in development time. Practically this means that under Windows user interface portions are written using the WinForms frameworks with a special variant of C++ known as C++/CLI, while under Mac OS X user interface code was written in Objective-C++ and utilizing the Cocoa framework. Due to time constraints and no single outstanding UI framework option, no GUI was constructed for Linux systems. However, if time permitted Linux UI toolkits would be compared and contrasted, and one would be selected based on which one best fit the design decisions made in OME; likely candidates would be GTK+, QT, wxWidgets, and Tk. To maximize code reusability, all UI specific code is separated as much as possible from any data handling structures provided by other OME libraries; all native UI implementations are expected to access the same model data structures and saved data visualization details regardless of their origins. Also, it is worth noting that OME development has almost exclusively target 64-bit processor architectures, as they are both abundant and better suited for larger data problems than 32-bit processor architectures. There is theoretically nothing standing in the way of compiling OME for any architecture that a compiler supports, but time and resource constraints have limited the exploration of building and running OME on alternate architectures.

SDP Implementation

The Utility of a Common Spatial Interface

A significant motivation for this project was the desire for a standardized method for implementing and accessing explicit spatial data from outside of a SD model's composition. The utility of such an implementation has arisen from real world applications; despite being inherently spatially unaware, ecological researchers and modelers have implemented their own solutions for utilizing spatial data in SD models (Ford 1999); a few examples of spatially explicit SD models using one solution, SME, are provided in the Literature Review section. When a model author comes up with their own solution to a spatial data representation problem, the burden of implementation is on them, which increases the amount of time and resources needed to successfully implement the model (Voinov et al. 2004). Such implementations are often redefining well known and commonly used relationships, and could benefit from standardization across various models (Voinov et al. 2004). Unfortunately, custom solutions to spatial relationships are often incompatible between models, making it very difficult to reuse the effort put in to a potentially standardized problem (this has been particularly noted in the reasoning behind building the SME environment for STELLA) (Voinov et al. 2004). Developing a standardized method of spatial representation would address some of the problems previously outlined. If a model is simulating a process common to a number of spatial environments, then a standardized spatial interface would allow the model to be applied to a different region by simply changing the spatial coverage input. This could also work the other way; multiple models working on the same coverage could influence each other by speaking through the same common spatial interface, allowing for more intricate interactions to be captured. By having a standardized spatial interface, modelers would only require knowledge of an opaque interface for requesting information about the coverage; details about the coverage implementation would be irrelevant to the model author. Without the need to focus on spatial implementation considerations, more time and resources can be spent towards factors that directly assist in answering the model's research question.

With the aforementioned issues in mind, a viable common interface should be able to satisfy the following requirements:

1. The model should not be required to have knowledge of the specifics of the spatial coverage, but should have the option to query such specifics if desired.
2. The model should be able to inquire about the values of attributes for any given spatial coverage record.
3. The model should have the option to modify or add attributes to the spatial coverage.

By satisfying these three requirements, a common interface would be both flexible and simple enough to implement in SD models while making it a straightforward to couple with arbitrary spatial coverages.

The Spatial Data Provider: OME's Common Spatial Interface

The solution to a common interface in OME is the Spatial Data Provider (SDP). The SDP is an interface for SD model logic to query for information on common spatial relationships. The implementation of the SDP can vary, but is expected to be responsible for loading, interpreting, and handling queries about a spatial coverage representation. Additionally, the implementation side of the SDP is intended to be defined either as a self-contained, standalone module or an interface to another tool in which OME is embedded. While the model can query for specific spatial coverage attributes (such as the method of coverage representation), it is not strictly necessary; instead, each “unit”, be it a point, grid cell, polygon, or some other coverage-designated discrete unit, can be accessed by its index and can return data about itself, such as its area or the indices of its neighboring units. Spatial records and their attributes are represented as they are in the standard dBASE (.dbf) files: the entire spatial record collection is presented as a relational table, where each discrete unit is a “record” or “row”, and each attribute is a “column” (Environmental Systems Research Institute 1998). By standardizing record access and coverage queries through the SDP interface, requirement 1 is satisfied, with various “getter” and “setter” functions satisfying requirements 2 and 3, respectively. By creating a standardized interface for

querying spatial data, model authors can focus on what makes their model a novel approach, rather than the (sometimes complicated) logistics of defining explicit spatial relationships as a part of the SD model diagram.

OME provides a C++ application programming interface (API) which defines the functionality that a SDP implementation must define, as well as a series of model expression functions that can be used to access the SDP interface during model runs. A full description of how to report implemented SDP functionality, the SDP API functions, and the SDP model expression functions can be found in Appendix B; what follows is a much more general overview of how to utilize a SDP from a model running within OME.

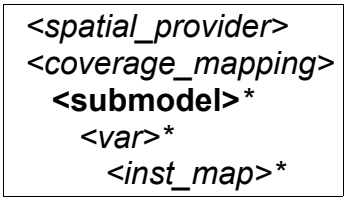

Figure 4. Control file tags pertaining to SDP configurations. Bold tags are required, while italicized tags are optional. Tags followed by an asterisk (*) may occur more than once within their positions.

The coupling between a Spatial Data Provider and a SD model begins within an OME control file. A control file contains two xml nodes relevant to mapping explicit spatial data (Figure 4). The first node of interest, **<spatial_provider>**, provides the path to the compiled SDP implementation and any attributes and/or sub-nodes which are necessary to properly initialize it for use with the provided model. This node may be omitted if OME is running as a plugin and the parent program handles all of the SDP details. The second relevant node, **<coverage_mapping>**, describes the mapping between model components and SDP records and attributes, and how values are shared between the model and the spatial coverage. Single-value model components can be directly mapped to coverage attributes in a one-to-one correspondence, but a model component containing multiple values for a single spatial unit must have any significant sub values directly mapped to a coverage attribute using the **<inst_map>** node (see Appendix C for tag details). There are four designated relationships between coverage

attributes and model components that can be mixed and matched: initialize a coverage attribute from a model component, initialize a model component from a coverage attribute, update a coverage attribute from a model component when its value changes, and update a model component from a coverage attribute when its value changes. Submodels can be specified to take their initial number of instances from the total number of records in the spatial coverage; this means that one or more submodels can directly map their unique instances to each polygon/cell/point in the spatial coverage, using the instance number to reference a corresponding record. While this is a common practice, it is not strictly necessary; since spatial records are referenced by index, any record could be referenced anywhere in the model. The **<coverage_mapping>** node may be omitted if there is no direct mapping between model components in the SD model and the spatial coverage; spatial queries can still be made, there will just be no implicit value sharing between model components and spatial coverage attributes. For a more detailed description of the **<spatial_provider>** and **<coverage_mapping>** nodes in the OME control file, see Appendix C.

For more than the most basic data exchanges between mapped model components and explicit spatial attributes, explicit methods for querying spatial relationships must be defined. OME provides several functions which are defined for use within model component expressions. The set of functions can be categorized into the following behavior-based groups: 1) querying details about the whole coverage, 2) querying about details for a specific attribute across all spatial records, 3) getting and/or setting values specific attributes in a specific spatial record, and 4) querying about spatial relations between spatial records. The first group contains functions that request details like the number of records, the type of coverage, and the spatial extents of the coverage. The second group contains functions that query the attribute index for a column title, and the minimum and maximum values for a given attribute across all records. The third group is essentially getters and setters for individual attributes in a given record. The fourth group is presently primarily concerned with neighbor relationships, focusing on querying or summarizing information about the records that are directly adjacent to a specific record in the spatial coverage. For a complete breakdown of SDP-related functions and the usage groups to which they belong, see Appendix B. These functions can be included in any model update expression

like any other expression function. For example, a submodel whose instances are intended to map one-to-one to a coverage could collect the total area taken by the neighbors of the associated spatial record to one of the instance's variable components by using the expression:

SDPGetNextToArea(index(1))

For more examples of accessing the SDP from model expressions, see details about the Tampa Bay Seagrass Model in the Simile Model Compatibility section.

Existing Example SDP implementations

As previously mentioned, when an OME model with explicit spatial requests is evaluated, a SDP implementation must be provided for the interface; this is accomplished either by specifying an implementation in the model control file (in a stand-alone run), or having a parent tool providing the implementation when OME is loaded as a plugin (in the case of an embedded run). There are two implementations of the SDP which have been used throughout OME's development process: CSV Spatial Data Provider and the Envision OME Adapter. CSV Spatial Data Provider is a stand-alone SDP implementation written as a means to test features as the project advanced in its development. This provider takes a CSV file and interprets it depending on settings in the model control file. Grid and polygon representations are supported, and Lua scripts can be used to describe neighbor relationships. The SDP itself is provided by an external dynamically-linked library that is referenced by the provided OME control (.omec) file. In contrast to the standalone CSV-based SDP, the Envision OME Adapter is a plugin module for Envision that embeds OME as part of the runtime Environment. Envision provides access to the coverage in a project's active context using the SDP interface. OME is also capable of using the SDP interface to update specified values in Envision as needed. The SDP in this case is bundled within the OMEAdapter plugin and is passed directly to the OMERuntime, ignoring any **<spatial_provider>** tags provided by the .omec file. Both of these SDP implementations are provided by dynamically-linked libraries which utilize the previously-mentioned SDP API.

Simile Model Compatibility

The reference models provided for this project were developed using Simile, which is discussed in the Literature Review. Since part of the OME development process has involved modifying and running existing models, it is a worthwhile exercise to compare the performance of the reference models when executed under Simile and OME separately. What follows is a brief overview of the reference models, a short discussion on some of the technical challenges that arose during the development process, and some simple performance comparisons between each of the reference models running under Simile and OME.

Reference Models Overview

The initial focus for this portion of the project was to get two different SD models running in OME, and then have them run under Envision using the OMEAdapter plugin and an Envision-native spatial coverage. Both SD models contained some explicit spatial aspect, and had runtime submodels which were static (submodel instances were not multiplied or removed at any point during simulation runs). Both models were also intended to run with an Euler's method integration solver, but would run with a RK4 solver as well. The two models that were selected for this project were HYGEIA and a draft version of John Rogers' Tampa Bay Seagrass Model.

HYGEIA is a written by Roel Boumans for predicting the rate in the rise in reported adverse health effects in the greater Austin area due to regional changes fueled by Global Climate Change induced heat stress (Boumans et al. 2014). This model is considered complete and has been published (Boumans et al. 2014). The model is the smaller of the two, but still contains a moderate number of model components. The spatial representation is solely used for output, and consists of 696 polygons defined in a CSV file representing the greater Austin area. Each polygon corresponds to an instance of a specific sub model container within the model.

The Tampa Bay Seagrass Model is an unpublished draft model produced by Dr. John Rogers as part of his work for the Gulf Ecology Division of the USEPA, and is used as part of the OME development process with permission. The model is intended to simulate the growth

and nutrient flow of seagrass in the Tampa Bay. The model's draft status is beneficial for OME's development since it allowed for testing of behaviors under developmental conditions prior to extensive model design optimizations. The model is also quite large, with just over 3,000 model components. The spatial representation coverage for this model consisted of nearly 50,000 hexagonal cells in an irregular configuration representing the area occupied by the northern third of Tampa Bay; each cell is defined by its central point as listed in an accompanying CSV. Similar to HYGEIA, each cell in the hex coverage corresponds with an instance of a specific submodel component. Unlike HYGEIA, however, the explicit spatial relationships are part of the simulation, with neighbor cells contributing to the dynamics of one another.

The overall Simile-to-OME workflow (as diagrammed in in Figure 2 in the Design Approach) is designed to allow for the remapping of a custom explicit spatial data representation in a Simile model to be mapped to the SDP interface calls; this remapping occurs during the conversion to OME's model filetype. To prepare an existing model to work with the SDP, a few steps need to be taken. First, the spatial components of the existing SD model must be identified and given instructions on how their structure will change to work with the SDP; these instructions should contain the new model expression utilizing one of more of the aforementioned SDP expression functions to query for spatial data. This is accomplished in Simile models by adding custom conversion instructions to the Comments field of affected model components (see Appendix A). As a Simile model is processed by the OME conversion tool (SMLConverter), custom conversion directives are scraped from the comments field in Simile model components and incorporated into the resulting model output.

Each model and their associated coverage files needed specific modifications before being suitable for running as a part of OME. For HYGEIA, the CSV file inputs were left as is; no interpretation of neighbor relationships between polygons was performed, as such relationships were not used as part of the model. A copy of the coverage was converted to a polygon .shp file for use with Envision by reconstructing the polygons using the values in the source CSV. A few model components were moved, and a submodel used for processing spatial data was removed

using some custom conversion syntax. No variable expressions were directly modified since output to Envision would be handled by a coverage mapping (see the SDP Implementation section).

The Tampa Bay Seagrass model required slightly more effort. To speed up independent neighbor testing, a column was added to the CSV coverage listing the neighbors for a given record; the equation used to find neighbors was functionally equivalent to the model selection criteria used in the originating model to determine neighbors. CSV interpretation was otherwise handled by the control document. A polygon-based .shp file was generated by constructing a hexagon-shaped polygon for each center point found in the CSV (the dimensions of the hexagon grid cell was determined by the distance between the center points of two hexagons and some simple trigonometry). Model augmentation was handled by converter control statements, which removed a relation/association model used to determine neighbors, and rewrote an expression to request neighbor information from the Spatial Data Provider (SDP) instead of the aforementioned removed submodel.

Design Challenges

Each model introduced unique challenges to OME that resulted in changes to the underlying runtime architecture. The first major hurdle introduced was memory fragmentation; as the update equations are solved and temporary values are allocated and freed from the computer's memory manager, available memory is often divided into smaller and smaller blocks and intermingled with allocated regions. Over time, this produces a noticeable spike in memory consumption and a drastic increase in runtime. This problem was resolved by allocating all temporary values from pools of memory put aside for constant reuse; at the end of every expression, the content of the pools is marked as free so that all the pool's memory is available in the next expression. This approach stabilized memory use and performance for arbitrarily long model runs. Another early roadblock was the massive amount of memory each instance of a given submodel would consume; this was due to each instance of a submodel making a complete copy of all internal components and their attributes. This was wasteful, as all the model

dynamics were concerned about were the present values contained within each instance; all other model component attributes were more or less consistent across submodel instances. Initially, each submodel instance was responsible for an instance value for each component contained within the submodel. Values were further decoupled by having all instanced values within the model stored in a single, massive data structure, while relegating submodel instance objects to tracking instance condition flags (flags would indicate if a model instance is active, newly created, or dead). Besides greatly reducing the memory footprint of each submodel instance, storing all model component values in a single monolithic structure had other advantages, namely value coherence (closely related values are stored near each other in memory, which is beneficial for optimizing for CPU caching architectures) and simplifying record keeping and recall.

The Tampa Bay Seagrass Model revealed an issue with the initial method for generating the update statements for each iteration of the model. The naive approach involved using the network of influences to generate an ordering of how each model component would be processed. Every time a submodel boundary was crossed, a for-loop was generated in the update expressions to update each instance (a submodel cannot necessarily be processed all at once if there are components outside of the submodel that both rely on some submodel components and are a dependency for other submodel components). This approach worked fine for small models, but for the Tampa Bay Seagrass Model, there were 19 for-loops generated for a the submodel with nearly 50,000 instances where a single loop would suffice. This situation created a large enough performance bottleneck to be considered unacceptable, so smarter pre-processing was necessary. Initially, a class was written to explicitly deal with building chunks of statements that could be run together and minimized the generation of for loops. While this process worked, it was slow, complicated, and difficult to modify when new edge cases were encountered. The present solution is much simpler; all component expressions are sorted based on the influence hierarchy as before, but, based on the assumption that the expression list is mostly sorted, a simple comparison sort algorithm is used to adjust the final position based on a few simple heuristics. This approach is faster, simpler, easier to maintain, and produces desirable results. With this final approach there are only two loops with nearly 50,000 instances, which results in

acceptable performance. The HYGEIA model introduced a different issue; that of efficient list operations. The original approach for manipulating lists was both wasteful and slow. A series of container types using a single common interface were devised; some were true list containers, while others were mappings into the value storage space, utilizing the previously mentioned value coherence to minimize the value mapping overhead. A number of implementations were iterated through until the present approach was arrived at. While the present approach is still a bit slower than Simile's equivalent operations, it is a bit more flexible and still runs in a period of time that has been deemed acceptable.

Runtime metrics and comparisons of values

To provide insight into the differences of running the two reference models with their native environment (Simile) and the experimental environment (OME), a series of tests were devised. The first set of tests were temporal: comparing simulation run time between OME running under several configurations and contrasting with the equivalent configurations under Simile. For the OME runs, compiled models were used, as Simile runs models compiled as C++ by default (Forrester 1968). Durations for each tool were recorded differently: for OME, the RawEngine executable was automated with a batch script and run against a set of control files (one for each test); RawEngine reports the duration of the simulation in seconds after each run. For Simile, a stopwatch was used; timing started after parameters were loaded and the model was compiled and initialized (indicated by a green status dot in Simile's execution window), and stopped when Simile had run through the duration set for the testing period. The HYGEIA model was run from time 0 through time 365 with a step interval of 1, while the Tampa Bay Seagrass Model was run from time 0 through time 300 with a step interval of 0.3. All tests were carried out in sequence on the same computer running Windows 7. The outcomes of the time trials are shown in Table 2.

Model, Integration method	Simile time (s)	OME time (s)	% Difference
Hygea, Euler	84	1039	1237%
Hygea, RK4	208	4158	1999%
TBSM, Euler	408	167	41%
TBSM, RK4	1265	620	49%

Table 2. Time trials comparing HYGEIA and Tampa Bay Seagrass Model (TBSM) runs using both Euler and Fourth-order Runge-Kutta (RK4) solvers between Simile and OME. HYGEIA runs much faster under Simile, while the Tampa Bay Seagrass Model runs almost twice as fast under OME.

As can be seen, HYGEIA runs faster under Simile, while the Tampa Bay Seagrass Model runs faster under OME. Generally, it is expected that models will run faster under Simile as it is a much more mature and polished tool. List operations, of which there are many in HYGEIA, run very slow in OME. The Tampa Bay Seagrass Model results however, are a bit surprising. Without full knowledge of the implementation details of Simile's simulation engine, the reasons for the better performance under OME are inconclusive. However, the increased performance in OME is likely due to how the explicit spatial relationships are handled when compared to Simile, either due to the preprocessing of neighbor relationships or the reliance on the SDP for handling details regarding explicit spatial relationships.

The second test involved identifying a set of sentinel values from each model and comparing their values at the end of equivalent Simile and OME test runs. The list of sentinel variable values for HYGEIA and the Tampa Bay Seagrass Model generated under Simile and OME are provided in Table 3 and Table 4, respectively. The runs were under the same conditions as the time trial tests, except that a variant of the HYGEIA model with all stochastic elements removed was used in order to ensure that any deviance in value was not due to intentional variability within the model (the Tampa Bay Seagrass Model does not have any stochastic model elements in the provided incarnation). Sentinel values were chosen based on their ability to capture overall activity within the greater model and showed active dynamics throughout the simulation time frame when integrated with Euler's method and/or RK4.

HYGEA Variable	Simile		OME		OME Deviance	
	Euler	RK4	Euler	RK4	Euler	RK4
Pollution Level[1]	0.35	0.175	0.35	0.175	0.00%	0.00%
cumulative morbidity[1]	0	0.2123	0	0.2123	0.00%	0.00%
cumulative morbidity[2]	0.00051	0.4575	0.00051	0.4575	0.00%	0.00%
cumulative morbidity[3]	0	38.2141	0	38.2141	0.00%	0.00%
cumulative morbidity[4]	0	3.5157	0	3.5157	0.00%	0.00%
cumulative morbidity[5]	0	394.56	0	394.56	0.00%	0.00%
Total Mortalities[1]	0	912.4123	0	912.4123	0.00%	0.00%
Total Mortalities[2]	2.1539	2074.5118	2.1539	2074.5118	0.00%	0.00%
Total Mortalities[3]	0	50321.4642	0	50321.4642	0.00%	0.00%
Total Mortalities[4]	0	2132.0221	0	2132.0221	0.00%	0.00%
Total Mortalities[5]	0	244601.96	0	244601.96	0.00%	0.00%

Table 3. Comparison of values generated by Simile and OME for sentinel values in the HYGEIA model. Bracketed ([ ]) values following the variable name refer the specific submodel instance from which the variable value was extracted from.

Tampa Bay Seagrass Model Variable	Simile		OME		OME Deviance	
	Euler	RK4	Euler	RK4	Euler	RK4
sumPC	166460797	N/A	15338848	N/A	90.79%	N/A
SL[1]	8.106	8.0933	8.106	8.0936	0.00%	0.00%
PC[369]	72674.7433	N/A	70944.7133	N/A	2.38%	N/A
PC[370]	72678.217	N/A	70321.0587	N/A	3.24%	N/A
PC[371]	48650.873	N/A	46129.9185	N/A	5.18%	N/A
Light1[1]	67673.3	67132.09	67673.3	67132.09	0.00%	0.00%
Light1[2]	74015.89	74548.173	74015.89	74548.173	0.00%	0.00%
Light1[3]	101517.649	100868.82	101517.649	100691.559	0.00%	0.18%
Light1[4]	113212.808	113171.34	113212.808	113348.674	0.00%	0.16%
Light1[5]	12637.764	126216.841	12637.764	126216.841	0.00%	0.00%

Table 4. Comparison of values generated by Simile and OME for sentinel values in the Tampa Bay Seagrass model. Bracketed ([ ]) values following the variable name refer the specific submodel instance from which the variable value was extracted from. fields marked with "N/A" indicate invalid values introduced into the simulation. The appearance of these values during the application of the RK4 solver reflects the early draft nature of this version of the Tampa Bay Seagrass Model, and should not be construed as a reflection of the validity of the model when it is completed and published.

For the HYGEIA tests, there were no significant deviations between the selected sentinel values regardless of the solver used. The Tampa Bay Seagrass Model, however had some noticeable deviations between the sentinel variable results generated under Simile and OME. The variable with the largest deviation, sumPC, is the result of summing nearly 50,000 values at each time step; even minor deviations in rounding errors between the two environments can

result in a significant difference. It is worth noting that when an experimental solver implementation with a degree of rounding error correction is used, the deviation is reduced, but not eliminated (the experimental solver is available in the OME distribution, as the HiRes solver, but was not used for these tests as it has not been thoroughly vetted). Also, a number of sentinel values report invalid values when integrating with the RK4 method. This behavior is consistent between Simile and OME, and therefore the report deviance is zero percent. However, if the invalid values had not been introduced during the simulation run it is likely that a deviance between OME and Simile generated values would be reported for the afflicted model variables.

While OME needs to emulate some of Simile's behaviors in order to ensure that the models produce similar results, they are not identical; consequently there will be some deviations between the values that are produced for OME and Simile (for a breakdown of the differences between Simile and OME with regards to model components and expression functions, see Appendix D). Nevertheless, the results of the value comparisons demonstrates that often times the values produced by the two environments are identical, and in cases where the values deviate from one another, they are still quite close. With a reasonable demonstration of performance established, we can now demonstrate benefits that are open to a SD model when run within the Envision modeling framework.

Case Study: Extending HYGEIA using Envision

OME has the capacity to run as a part of Envision, interacting with it through the Autonomous Process plugin interface (see Figure 5). This exchange of data provides the opportunity to extend the functionality of a System Dynamics (SD) model by allowing one or more of Envision's various mechanisms to modify the value of SD model components, effectively acting as one or more external inputs. To demonstrate this capability and its potential to extend the utility of an existing SD model, a simple example has been contrived.

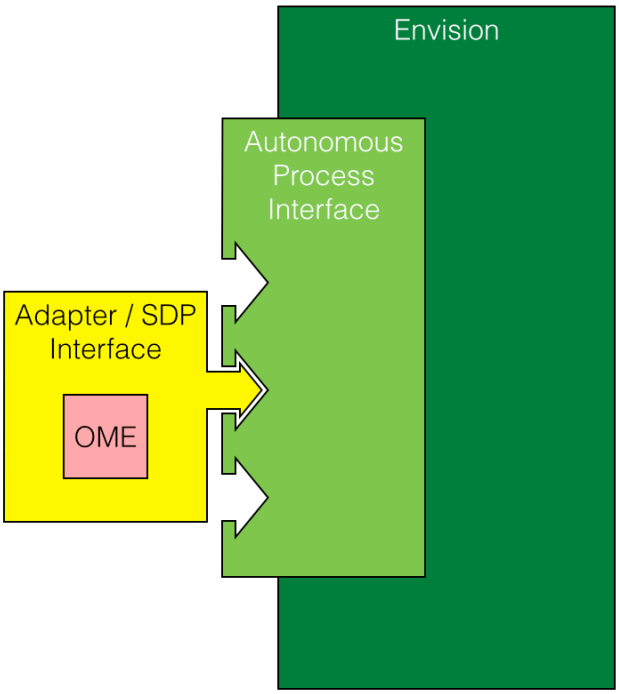

Figure 5. Broad overview of how OME and OMEAdapter relate to Envision. OMEAdapter encapsulates the OME simulation engine, and conforms to Envision's Autonomous Process interface; from Envision's perspective, OME operates as just another Autonomous Process Module.

Premise

Each explicit spatial unit in HYGEIA contains a proportional representation of 15 different landcover types which are defined as part of the parameter input at the beginning of the simulation run. The landcover proportions remain static throughout the run and have an influence on a number of dynamics, such as the reported tree cover ratio, the cumulative air pollution levels for different compounds, and the regional windspeed. While it is perfectly reasonable to treat land covers as static values as part of the collection of assumptions that define the model, land covers realistically do change gradually. In a developed area (such as the greater Austin area that HYGEIA simulates), it is not unusual for the density of developed regions to increase over time. HYGEIA has four classes of developed landcover: open land, low-density developed, medium-density developed, and high-density developed. Thus, if simulations under HYGEIA were to be modified to simulate the succession from low-density to medium-density development and from medium-density to high-density development using a few simple rules, there should be a noticeable shift in dynamic and/or period-specific model outputs.

To simulate succession events during a simulation run, a pair of Envision's policies are used for defining conditions and behaviors, while an actor is defined which will determine the frequency at which the policies are applied. The general idea is that if the proportion of low-density or medium-density developed landcover is above some threshold, it may have its proportion transferred to the next highest tier of developed land cover (medium-density developed or high-density developed, respectively). This would be analogous to a spontaneous spurt of development in a region that is sufficiently developed to sustain the activity.

Implementation Details

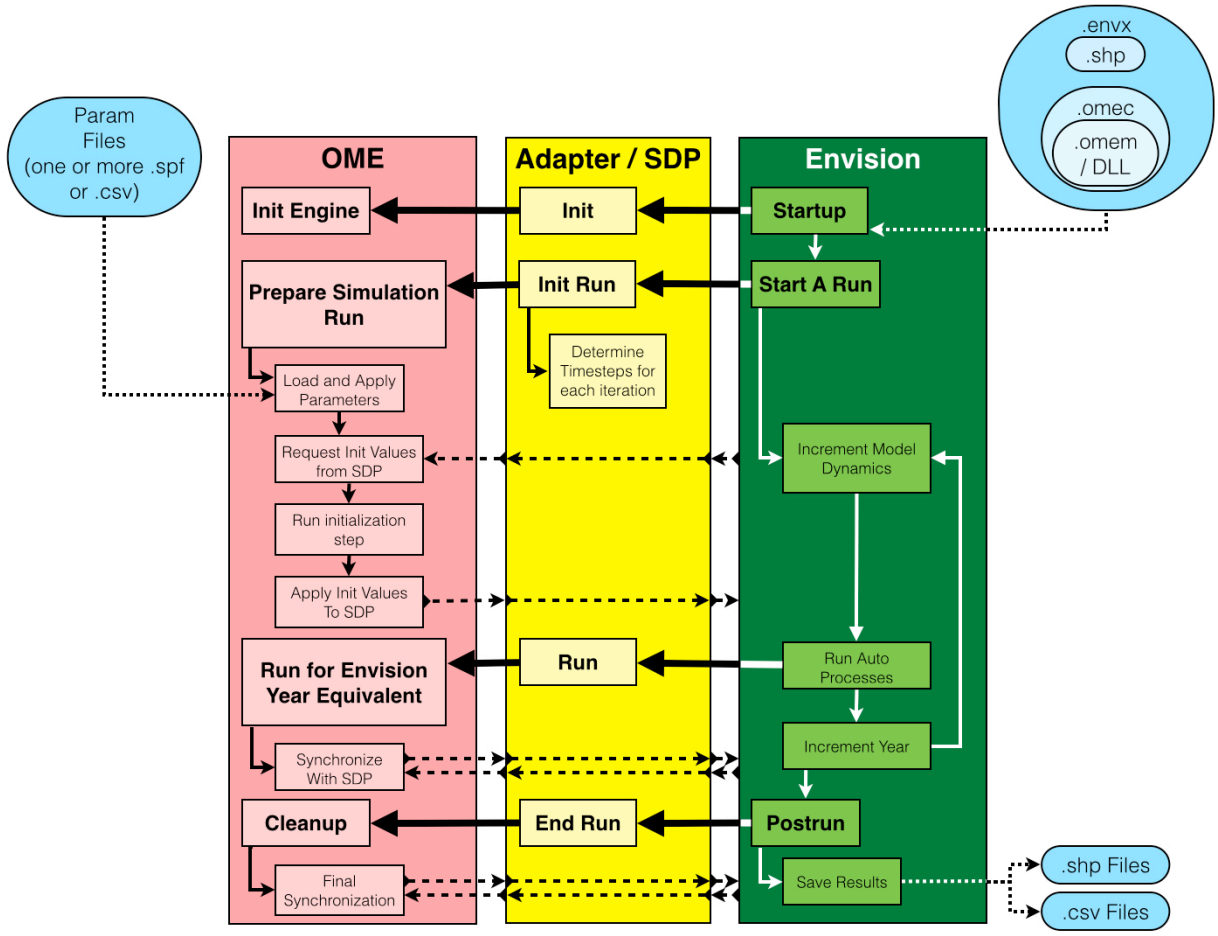

Figure 6. Interaction between OME, OMEAdapter and the SDP interface, and Envision. Pink, yellow, and green boxes refer to processes performed by OME, OMEAdapter and Envision, respectively. Blue capsules represent external files, with nested capsules representing internal file linkages. Thick arrows show process invocations across process domain boundaries, while thin arrows represent the process invocations within domains. Dotted Arrows represent the reading and writing of external files. Dashed arrows represent requests from OME to either read or write spatial data from Envision which are mediated through OMEAdapter's implementation of the SDP interface (see the SDP Implementation section). Note that the SDP spatial requests (the dashed lines) are optional, and not all models will utilize each point of communication.

For the sake of reproducibility, a strictly deterministic variant of the HYGEIA model (HYGEIA-Determ) was used with all the stochastic components set to static values. The

communication between HYGEIA and Envision is best described within the context of Figure 6, which summarizes the interactions between OME, the OMEAdapter and SDP interface, and Envision during a simulation run. The process of running HYGEIA under Envision actually starts within an Envision project (.envx) file, where the HYGEIA spatial coverage is added as a layer, and the OMEAdapter is specified as an autonomous process. As part of the specification of OMEAdapter in the Envision project file, an initialization string is provided which is the path to the OME control file (.omec) for OMEAdapter to load. For this case study, the OME control file specifies a precompiled version of the HYGEIA-Determ model.

The Envision project file is loaded when Envision starts up. During this process, the OMEAdapter initialization function is invoked; the initialization function configures the OME simulation engine for running as a plugin, provides it with the OME control file path that was defined in the Envision project file, and registers a Spatial Data Provider (SDP) to be used. This SDP lives within the OMEAdapter and interprets OME spatial queries in the context of the current Envision MapLayer. For more information on SDP usage, see the section titled SDP Implementation, and/or Appendix B. The OME Engine initialization involves loading the data specified in the OME control file, initializing several containers, and priming the OME Event handler. The HYGEIA-Determ Envision project file defines three policies and an associated actor which are loaded by Envision during the Initialization process as well; these policies and the associated actor are intended to simulate development succession using fields that are defined in the HYGEIA-Determ model and are shared between OME and Envision using the SDP interface. The policies are defined to be trigger the low-to-medium developed density land cover transition if a polygon has low-density developed land being greater than 10% of its landcover. Similarly, the transition from medium-to-high density developed land is triggered by a medium-developed density being greater than 20% of a polygon's landcover. The actor which evaluates these policies is set to apply them to any given valid region on an average of once every 3 years.

When the user begins a simulation run in Envision, the OMEAdapter's run initialization function is called. This function handles two tasks; the first is to determine how many time steps to run the OME simulation for every simulation step in Envision. Envision typically runs in yearly timesteps, while a HYGEIA iteration represents a single day (this is specified by the

“time_units” attribute in the .omec's **<ome_model>** tag; see Appendix C for more details). In this case, the OMEAdapter determines that each time Envision asks it to run for one of its iterations, it will tell the OME engine to run for 365 iterations before returning. The second task is to tell OME to prepare for a simulation run. This triggers a sequence of events in OME. First the external parameter files are applied; in the case of HYGEIA-Determ, the original HYGEIA Simile parameter file (.spf) is used to load specific parameter values, and all .csv files pointed to by the .spf or internally by model components are loaded and applied at this time as well. The next step has OME request values from the SDP to use to initialize model components as specified in the **<coverage_mapping>** tag in the control file (see SDP Implementation and/or Appendix C); HYGEIA does not require any initializations from the spatial coverage, so this step is skipped. Next, OME runs it's internal initialization step, configuring model components for the beginning of a simulation. After this, OME tells the SDP in the OMEAdapter that it wants to initialize any specified fields from values derived during its initialization process; in the case of HYGEIA-Determ this includes all the fields which represent each polygons proportional land coverage, as well as a few fields used for visualizations (e.g. Tree Cover Ratio).

Once all components have initialized for a run, Envision will begin its simulation loop. For HYGEIA-Determ, Envision is set to run for 5 years; this will translate to 1825 days in the OME-side of the run. Envision will apply the actors and policies determined by previously mentioned probabilities, and will then invoke OMEAdapter's Run function. This function tells OME to run for 365 iterations before returning control back to Envision. Once OME runs for the specified period, values are synchronized through the SDP interface based on the specifications provided in the OME control file; for HYGEIA-Determ, any landcover proportions that are changed in Envision through the application of policies are imported through the SDP and used to update the appropriate value in the OME representation of the model. After OME completes its synchronization step, control is returned to Envision, which continues to carry out it simulation loop.

When Envision finishes running its simulation, its post-run and cleanup processes call OMEAdapter's End run function, which in turn calls OME's cleanup function. This releases some of the temporary memory used by OME and forces a final synchronization update. Once this is

done, Envision can generate several outputs which can then be displayed and/or written out to .csv, .shp, or image files. For HYGEIA-Determ, several map coverages are displayed in the post-run results view in Envision, which are then screen captured and used for the discussion in the following results section.

Results

A simulation was run using the HYGEIA-Determ model as described in the previous section. Several results were generated and are displayed here. Figures 7 to 9 shows the differences in low, medium, and high density land covers before and after the five year simulation run. Figure 10 shows regions where the tree cover ratio has changed over the five year period.

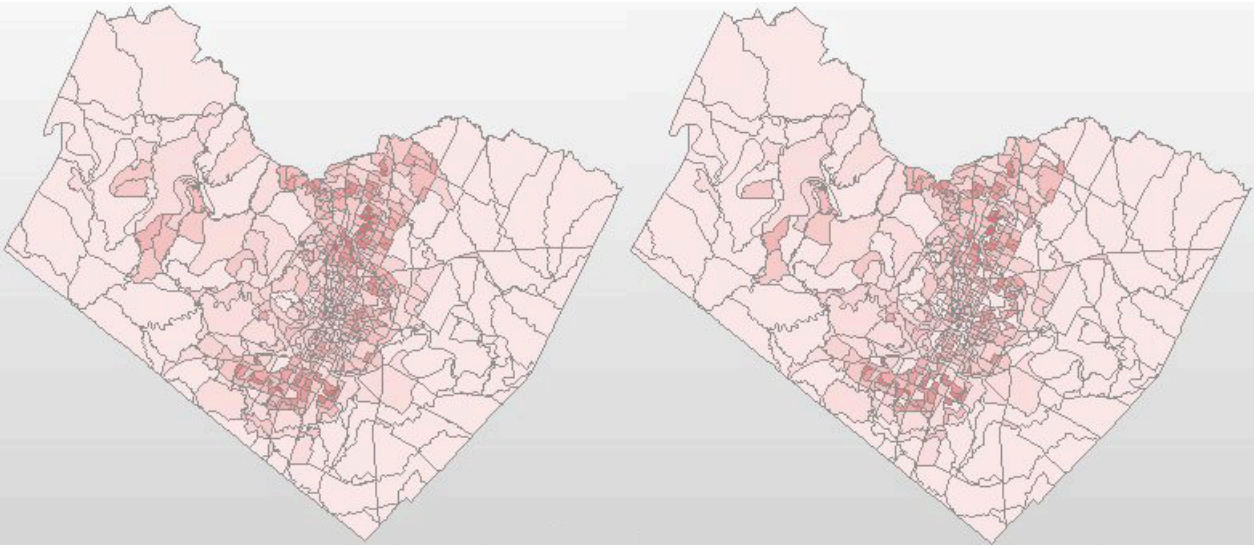

Figure 7. Change in low-density developed regions from beginning to end of run. The map on the left represents the amount of low-density development at the beginning of the run, while the map on the right represents the amount of low-density development after five years of simulation. Darker shades represent a greater proportion of the polygon covered by low-density development. Note the lighter patches in the center of the map on the right which are absent from the map on the left; these polygons had their low-density development reallocated into the medium-density development coverage. Maps generated using Envision.

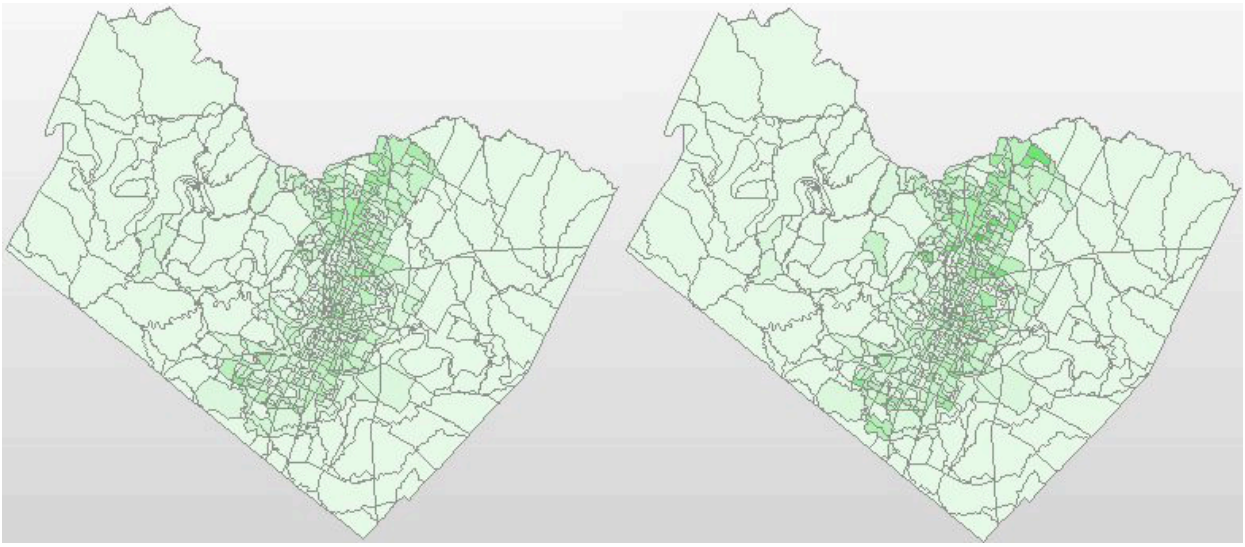

Figure 8. Change in medium-density developed regions from beginning to end of run. The map on the left represents the amount of medium-density development at the beginning of the run, while the map on the right represents the amount of medium-density development after five years of simulation. Darker shades represent a greater proportion of the polygon covered by medium-density development. Polygons which are lighter on the right had their medium-density developed regions reallocated to high density-developed regions, whereas polygons that are darker on the right had their low-density developed regions added to their medium-density developed regions. Maps generated using Envision.

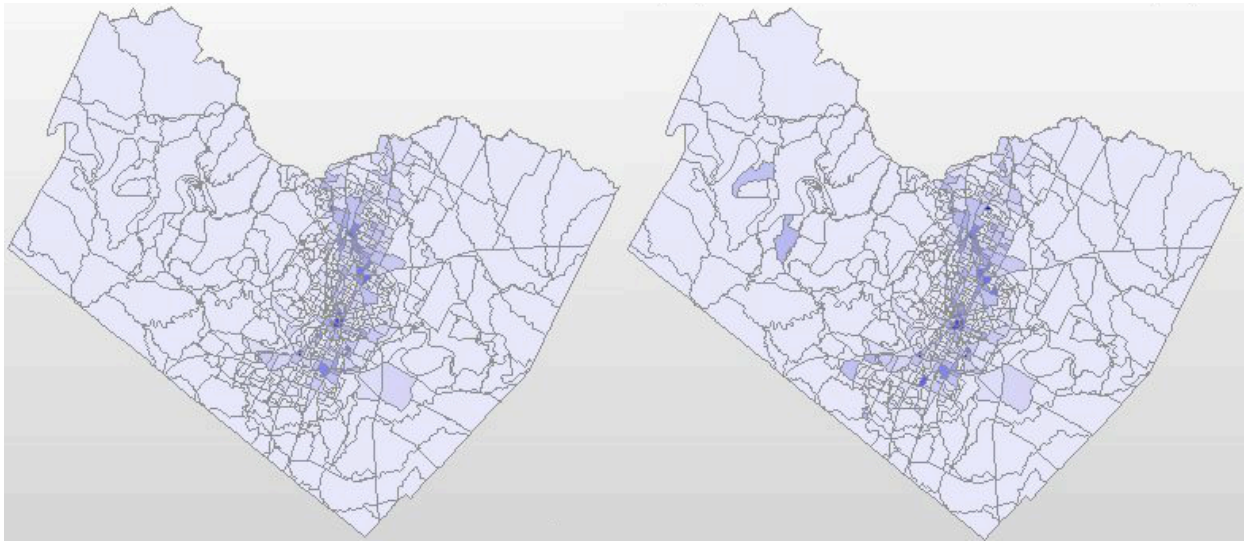

Figure 9. Change in high-density developed regions from beginning to end of run. The map on the left represents the amount of high-density development at the beginning of the run, while the map on the right represents the amount of high-density development after five years of simulation. Darker shades represent a greater proportion of the polygon covered by high-density development. Polygons on the right that appear darker than they appear on the left have had their medium-density developed land cover proportions added to their high-density amounts. Maps generated using Envision.

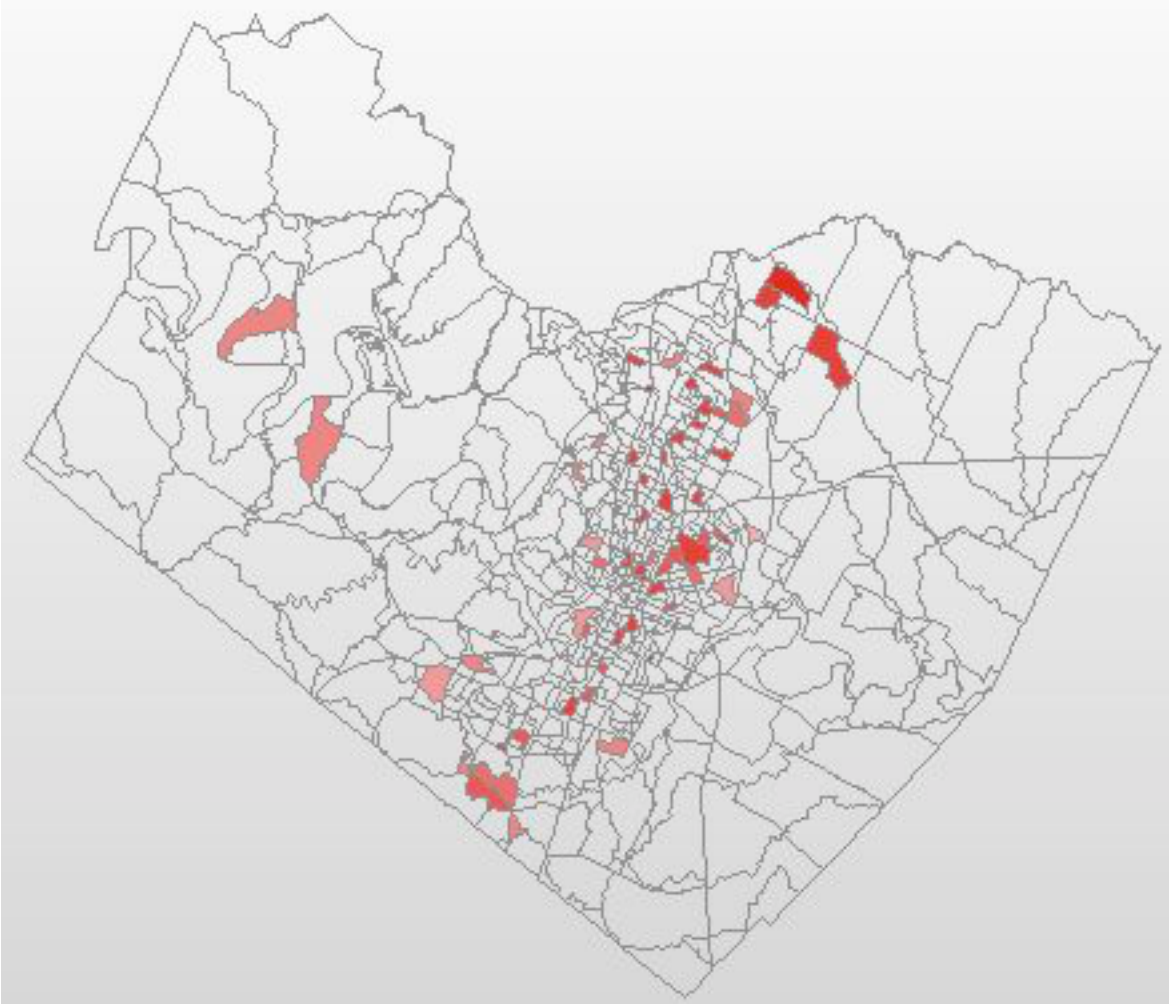

Figure 10. Changes in tree cover ratio. Each colored polygon has had its tree cover ratio value changed at some point during the five year simulation run in Envision. The darker the shade the greater the new ratio is. Each tree cover ratio change is due to a shift in the polygon's land cover proportions. If this map were the result of running without the changes to the landcover, no polygons would be shaded. Maps generated using Envision.

Discussion

For the purposes of this study, the values of the actual change are unimportant; this is a purely contrived example. The fact that Envision is capable of changing a static component of the original model in a reasonable manner is significant, as it shows that not only can Envision

work with data originating from OME, but that it can also modify data within OME as well. The only fields allowed to take values directly from Envision were the 15 landcover type percentages. By just changing these values, other dependent processes are affected, as represented by the changes in the Tree Cover Ratio for the modified regions. The changes may seem small, but that is an artifact of how this model runs in Envision: even though the SD model ran for 1,825 iterations, Envision only ran for five. Thus, the opportunity for the Actor entity to pick regions to be modified only occurred five times. Longer runs would likely produce more variation in coverage, but would not necessarily contribute to the point here, which is that model communication between OME and Envision can be treated as a two-way street.

The mechanics of the bidirectional communication between OME and Envision are sound, as has been demonstrated by this short case study. Future expansion or incorporation into a genuine Envision stakeholder project could provide much more practical utility for relatively little effort on behalf of the modeling group. By introducing OME into the Envision plugin ecosystem, the strengths of system dynamics can be used and/or extended without limiting the inherent flexibility that makes Envision ideal for stakeholder projects.

Conclusion

The Open Modeling Environment (OME) was created as an attempt to address shortcomings that are common with System Dynamics (SD) modeling tools. The two novel aspects, a focus on modularity and a common interface for spatially explicit data, were the components that most directly addressed known shortcomings. To reiterate, the two research question presented at the beginning of this thesis were:

1. Can an open-sourced, modular SD modeling tool be constructed with performance and features comparable to existing commercial counterparts?
2. Does adding an explicit spatial data interface, adopting a modular design, and using a middle-of-the-road model file format simplify the implementation of spatially-explicit SD models?

Regarding the first question, the answer is arguably yes, albeit with some limitations. OME was constructed mostly from scratch at the beginning of this project and is presently capable of converting and running some Simile models (including the two reference models used throughout this study), running as a standalone application or as a plugin to Envision, producing values that are closely in line with the source models running under Simile, and incorporating spatial data from standalone CSVs or from Envision's runtime environment. However, time and resource constraints limited the extent of implementation details: there are some significant Simile features and model components that remain unimplemented in OME (see Appendix D), the only plugin interface exists for Envision (and is somewhat underdeveloped), derived values do not completely match the equivalent values produced in the originating Simile environment, and the spatial data provider (SDP) implementations are limited to a few fringe or simple implementations.

Regarding the second question, the answer is definitely yes based on what has been demonstrated in the Simile Compatibility Details and Case Study sections. An explicit spatial representation allowed for submodels representing spatial geometries to be be completely

excised from both reference models, simplifying them by reducing the total number of model components. In the case of the Tampa Bay Seagrass Model, moving the explicit spatial information into an external SDP simplified the update process of the model, reducing the time it took for the model to run. Additionally, incorporating the reference models into the Envision environment was a simple process involving enabling the plugin in an Envision modeling project, identifying the control file to read, and adding a few lines to the control file itself to describe how the two environments would exchange information. The effort applied to modifying these few files is vastly outweighed by the potential increase in utility provided by interoperability with the Envision modeling framework.

While a significant amount of functionality already exists within OME, its current state is intended to just be the first iteration in the overall development lifecycle; as such, there is significant room for expansion and improvement. The first and perhaps most obvious improvement would be to increase overall compatibility with Simile's functionality. Along the same lines, it would be desirable to get models from other SD modeling tools (such as Vensim and STELLA) successfully converted and running under the OME framework. Presently, a full implementation of OME exists on Windows, a nearly full implementation exists on Mac OS X, and no implementation has been built for Linux; building full implementations of OME on the latter two platforms could increase the available audience who would see utility in OME, as not all research projects are carried out with just Windows-based tools. Overall performance could also use some work toward improvement; further optimizations could be made throughout the source code, particularly with list handling and other update expression operations. One potential set of optimizations would involve implementing a parallel execution scheme; many portions of the code are structured in such a way to make it straightforward to restructure them for parallel processing, and running code in parallel has the potential to increase performance on multi-core machines. Finally, implementing a full graphical interface complete with SD iconography for OME would be beneficial for the overall project, as it would round it out as a full SD modeling environment. SD models have had a standardized graphical representation since their original inception (Forrester 1968), and having a graphical frontend would allow for a canvas that could

be used to increase overall communication and potentially allow for authoring models without reliance on outside tools.

SD models are beneficial to the study of ecological models, but the tools and concepts that make up the SD modeling environment are still relatively young and are undergoing gradual refinement. OME's introduction to this environment will provide another step in the evolution and maturation of SD models and their respective tools.

Bibliography

Abedi, Aidin. 2007. “Shiny FAQ.” Source Code Documentation. https://code.google.com/p/shinyprofiler/.

Adler, Joseph. 2009. *R in a Nutshell: A Desktop Quick Reference*. Edited by Mike Loukides. 1st ed. Sebastopol,CA: O’Reilly Media, Inc.

Aycrigg, Jocelyn, Steven J. Harper, and James D. Westervelt. 2004. “Simulating Land Use Alternatives and Their Impacts on a Desert Tortoise Population in the Mojave Desert, California.” In *Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach*, edited by Robert Costanza and Alexey Voinov, 1st ed., 249–74. New York, New York, USA: Springer.

Bangerth, Wolfgang, and Timo Heister. 2013. “What Makes Computational Open Source Software Libraries Successful?” *Computational Science & Discovery* 6 (1): 015010. doi:10.1088/1749-4699/6/1/015010.

Behm, Pamela, Roelof M.J. Boumans, and Frederick T. Short. 2004. “Spatial Modeling of Eelgrass Distribution on Great Bay, New Hampshire.” In *Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach*, edited by Robert Costanza and Alexy Voinov, 1st ed., 173–96. New York, New York, USA: Springer.

Berg, Ingo. 2005. “Muparser - A Fast Math Parser Library.” http://articles.beltoforion.de/article.php?a=muparserx.

Bishop, Judith, and N. Horspool. 2006. “Cross-Platform Development: Software That Lasts.” *Computer* 39 (10): 26–35. doi:10.1109/MC.2006.337.

Bolte, John. 2014. *Envision Developer’s Manual*. Oregon State University. http://envision.bioe.orst.edu.

Borshchev, Andrei, and Alexei Filippov. 2004. “From System Dynamics to Agent Based Modeling.” *Simulation* 66: 25–29. http://www.econ.iastate.edu/tesfatsi/systemdyndiscreteeventabmcompared.borshchevfilippov04.pdf.

Boumans, Roelof J.M., Donald L. Phillips, Winona Victery, and Thomas D. Fontaine. 2014. "Developing a Model for Effects of Climate Change on Human Health and Health–environment Interactions: Heat Stress in Austin, Texas." *Urban Climate* 8: 78–99. doi:10.1016/j.uclim.2014.03.001.

Bray, Tim, Jean Paoli, Eve Maler, and Sun Microsystems. 2008. "Extensible Markup Language (XML) 1.0 (Fifth Edition)." *W3C Recommendation* 0: 1–37. http://www.w3.org/TR/2008/REC-xml-20081126.

Chichakly, Karim. 2013. *XMILE : An XML Interchange Language for System Dynamics*.

Clewley, Robert. 2012. "Hybrid Models and Biological Model Reduction with PyDSTool." *PLoS Computational Biology* 8 (8): e1002628. doi:10.1371/journal.pcbi.1002628.

Costanza, Robert, and Alexey Voinov. 2004. "Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach." *Modeling Dynamic Systems*.

Egner, Joanne. 2014. "Version 9.1.3 Update Key Features." *Making Connections: Iss Systems Blog*. Accessed August 12. http://blog.iseesystems.com/stella-ithink/version-9-1-3-now-available/.

Environmental Systems Research Institute. 1998. "ESRI Shapefile Technical Description." doi:10.1016/0167-9473(93)90138-J.

———. 2006. "About Getting Started with Writing Geoprocessing Scripts." *ArcGIS Desktop Help 9.2*. http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=About_getting_started_with_writing_geoprocessing_scripts.

Ford, Andrew. 1999. *Modeling The Environment: An Introduction to System Dynamics Modeling of Environmental Systems*. Washington, D.C.: Island Press.

Forrester, Jay W. 1961. *Industrial Dynamics*. *Harvard Business Review*. http://hdl.handle.net/2027/mdp.39015000457070.

———. 1968. "Industrial Dynamics—A Response to Ansoff and Slevin." *Management Science* 14 (9): 601–18. doi:10.1287/mnsc.14.9.601.

———. 1969. *Urban Dynamics*. 1st ed. Cambridge: The M.I.T. Press.

Gwebu, Kholekile L., and Jing Wang. 2011. "Adoption of Open Source Software: The Role of Social Identification." *Decision Support Systems* 51 (1). Elsevier B.V.: 220–29. doi:10.1016/j.dss.2010.12.010.

Haefner, James W. 1996. *Modeling Biological Systems: Principles and Applications*. New York: Chapman & Hall.

Hertz, Matthew, and Emery D. Berger. 2005. "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management." *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA '05*. New York, New York, USA: ACM Press, 313. doi:10.1145/1094811.1094836.

Hillyer, Charles, John Bolte, Frits van Evert, and Arjan Lamaker. 2003. "The ModCom Modular Simulation System." *European Journal of Agronomy* 18 (3-4): 333–43. doi:10.1016/S1161-0301(02)00111-9.

Hirschi, A. 2007. "Traveling Light, the Lua Way," no. October.

Ierusalimschy, Roberto, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. "The Evolution of Lua." *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages - HOPL III*. New York, New York, USA: ACM Press, 2–1 – 2–26. doi:10.1145/1238844.1238846.

Isee Systems. "STELLA: Systems Thinking for Education and Research." http://www.iseesystems.com/softwares/Education/StellaSoftware.aspx.

ISO/IEC. 2011. "ISO/IEC 14882:2011: Information Technology – Programming Languages –C++." Geneva: International Organization for Standardization.

Jacques, Mathieu. 2004. "An Extensible Math Expression Parser with Plug-Ins - CodeProject." *Code Project*. http://www.codeproject.com/Articles/7335/An-extensible-math-expression-parser-with-plug-ins.

Lane, David C. 2000. "Diagramming Conventions in System Dynamics" 51 (2): 241–45.

Lane, David C. 2008. "The Emergence and Use of Diagramming in System Dynamics: A Critical Account." *Systems Research and Behavioral Science* 25 (1): 3–23. doi:10.1002/sres.826.

Lünsdorf, Ontje, and Stefan Scherfke. 2014. "Welcome to SimPy." http://simpy.readthedocs.org/en/latest/index.html.

Mazzoleni, S, F Giannino, M Colandrea, M Nicolazzo, F Di Agraria, and J Massheder. 2003. "Integration of System Dynamics Models and Geographic Information Systems." In *Modelling and Simulation 2003*, 600:304–6.

Mazzoleni, S, F Giannino, M Mulligan, D Heathfield, M Colandrea, and M Nicolazzo. “A New Raster-Based Spatial Modelling System : 5D Environment.”

OASIS. “OASIS XML Interchange Language (XMILE) for System Dynamics TC.” https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xmile.

Oliphant, Travis E. 2007. “Python for Scientific Computing.” *Computing in Science & Engineering* 9 (3): 10–20. doi:10.1109/MCSE.2007.58.

Open Source Initiative. 1998. “The Open Source Definition | Open Source Initiative.” http://opensource.org/definition.

Oregon State University. “Envision: About Envision.” http://envision.bioe.orst.edu/About.aspx.

———. 2014b. “Envision: Home Page.” Accessed October 11. http://envision.bioe.orst.edu.

Perez, Fernando, Brian E. Granger, and John D. Hunter. 2011. “Python: An Ecosystem for Scientific Computing.” *Computing in Science & Engineering* 13 (2): 13–21. doi:10.1109/MCSE.2010.119.

Petzoldt, Thomas. *Dynamic Simulation Models – Is R Powerful Enough ?*

Petzoldt, Thomas, and Karsten Rinke. 2007. “Simecol : An Object-Oriented Framework for Ecological Modeling in R.” *Journal of Statistical Software* 22 (9): 1–31. http://www.jstatsoft.org/v22/i09.

R Foundation. 2014a. “What Is R?” Accessed January 12. http://www.r-project.org/about.html.

———. 2014b. “R FAQ: Frequently Asked Questions on R.” Accessed February 12. http://cran.r-project.org/doc/FAQ/R-FAQ.html.

Robles, Gregorio, and JesúsM. González-Barahona. 2012. “A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes.” In *Open Source Systems: Long-Term Sustainability SE - 1*, edited by Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi, 378:1–14. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg. doi:10.1007/978-3-642-33442-9_1.

Ruparelia, Nayan B. 2010. “The History of Version Control.” *ACM SIGSOFT Software Engineering Notes* 35 (1): 5. doi:10.1145/1668862.1668876.

Siau, Keng, and Qing Cao. 2001. “Unified Modeling Language: A Complexity Analysis.” *Journal of Database Management* 12 (1): 26–34.

Simulistics Ltd. “Simulistics: History.” doi:sim.

———. “Simulistics: Simile at a Glance.” http://www.simulistics.com/overview.htm.

———. 2014c. “Running Models : Working with Visualisation Tools : Polygon Diagram.” Accessed August 12. http://www.simulistics.com/help/run/tools/polygons.htm.

———. “Simulistics: Land-Use Change.” http://www.simulistics.com/examples/landuse/landuse.htm.

———. “Simulistics: Model Diagram Elements : Role Arrow : Two Role Arrows from Different Submodels.” http://www.simulistics.com/help/concepts/object/role/dual.htm.

———. “Simulistics: Working with Submodels : Introduction to Association.” http://www.simulistics.com/help/submodels/association/introduction.htm.

Sklar, Fred H, and Robert Costanza. 1991. “The Development of Dynamic Spatial Models for Landscape Ecology: A Review and Prognosis.” In *Qualitative Methods in Landscape Ecology: The Analysis and Interpretation of Landscape Heterogeneity*, edited by MG Turner and RH Gardner, 239–88. New York: Springer-Verlag.

Soetaert, Karline, Thomas Petzoldt, and Woodrow Setzer. 2010. “Solving Differential Equations in R: Package deSolve.” *Journal of Statistical Software* 33 (9): 1–25. http://www.jstatsoft.org/v33/i09/.

The Python Foundation. 2014a. “General Python FAQ.” Accessed February 12. https://docs.python.org/2/faq/general.html.

———. 2014b. “PyPI - the Python Package Index.” Accessed February 12. https://pypi.python.org/pypi.

Thomason, Lee. 2014. “TinyXML-2 Documentation.” http://www.grinninglizard.com/tinyxml2docs/index.html.

Ventana Systems Inc. “Vensim: Vensim History.” http://vensim.com/vensim-history/.

———. 2014b. “Vensim Brochure.” Accessed December 11. http://vensim.com/vensim-brochure/.

———. 2014c. “Vensim® Applications.” Accessed August 12. http://vensim.com/vensim-applications/.

Voinov, Alexey, Carl Fitz, Roelof Boumans, and Robert Costanza. 2004. “Modular Ecosystem Modeling.” In *Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach*, edited by Robert Costanza and Alexey Voinov, 1st ed., 19:43–76. New York: Springer. doi:10.1016/S1364-8152(03)00154-3.

Von Krogh, Georg, and Eric von Hippel. 2006. “The Promise of Research on Open Source Software.” *Management Science* 52 (7): 975–83. doi:10.1287/mnsc.1060.0560.

Wegner, Peter. 1990. “Concepts and Paradigms of Object-Oriented Programming.” *ACM SIGPLAN OOPS Messenger* 1 (1): 7–87. doi:10.1145/382192.383004.

Wolstenholme, E . F . 1982. “System Dynamics in Perspective.” *The Journal of the Operational Research Society* 33 (6): 547–56.

APPENDICES

Appendix A: Model Conversion Directives

In order to augment System Dynamics models during the conversion process, a simple command syntax has been implemented. The modification directives are intended to be implemented on a per-model component basis, with the actual commands being placed in a field that has no effect on the model execution within its native environment. The upshot to this approach is that an originating model can be modified to make the appropriate changes during the OME conversion process while still being able to function normally within its native environment.

Presently, conversion tools only exist for models originating from Simile. Each Simile model component has a "comments" field which is reserved for free text to be used for any further annotation required by the author. It is within this field that conversion directives can be defined and are extracted from during a model's conversion from Simile's native format to OME's intermediate XML format.

To denote the beginning of OME conversion directives within a free text field, the string (excluding the quotes) "--!!OME:" is used; the ending of the block (again excluding the quotes) is denoted by the string "!!--". Each directive within the directive block is prefixed with an at ('@') symbol. Some directives require an argument; in this case, a colon (':') separates the command from the argument.

Here are a pair of examples of OME conversion directive blocks:

Omit an object:

--!!OME:@omit!!--

Mark an object as init only and change the update expression to read time (the object will now record the start time):

--!!OME:@initOnly@expression:time()!!--

The following are the directives that will be processed by the conversion tool (text enclosed by < and > denote arguments):

omit - Removes the object from the model and any associated influences. If **omit** is applied to a submodel, all objects contained within a submodel are removed as well. Removing a submodel can be useful particularly when a spatial coverage is replacing association-linked models used for determining neighbors.

move:<mdlName> - Move the model component to the model referred to by the name supplied by the **<mdlName>** argument. This will break any influences to/from the object. Influences can be restored using the **influenceTo:** and **influenceFrom:** directives.

initOnly - Explicitly marks an object to only be evaluated once at the beginning of a simulation. This can be useful for optimization reasons.

influenceTo:<objName> - Create a new influence from the current model component, to the component identified by **<objName>**. Can be used to patch holes in the model created by the **omit** command.

influenceFrom:<objName> - Create a new influence from the model component specified by **<objName>** to the current component. Can be used to patch holes in the model created by the **omit** command.

expression:<newExpr> - Override an object's expression with the one supplied in the **<newExpr>** argument. This is useful for patching holes from the **omit** expression and adding Spatial Data Provider commands to the converted model.

expectsSpatial – Indicates that a submodel is expected to have its number of initial instances set by a SDP at runtime.

Appendix B: SDP functions and flag mappings

The Spatial Data Provider (SDP) interface uses a series of flags to inform the parent environment of a given SDP implementation's capabilities. Descriptions of the flags and their associated functions are described below.

Flag	**Description**	**Flag-Specific Required C++ Functions**
NONE	The default NULL value.	
POINT_SUPPORT	Supports a point grid-based coverage.	
HEX_SUPPORT	Supports a hexagon grid-based coverage.	
GRID_SUPPORT	Supports a square grid-based coverage.	GetCellSize()
POLY_SUPPORT	Supports a hexagon grid-based coverage.	
QUERY_SUPPORT	Advanced Queries are supported. Syntax of query is specific to SDP implementation.	NextTo() NextToArea() Within() WithinArea() GetNextTo() - only if NEIGHBOR_SUPPORT and ITR_SUPPORT are set. GetWithin() - only if ITR_SUPPORT is set.
NEIGHBOR_SUPPORT	Basic neighbor queries are supported.	GetNeighbors() GetNeighborCount() GetNeighborRecord() GetNextTo() - only if QUERY_SUPPORT and ITR_SUPPORT are set.
ITR_SUPPORT	Forward iterators can be requested.	GetIterator() GetNextTo() - only if QUERY_SUPPORT and NEIGHBOR_SUPPORT are set. GetWithin() - only if QUERY_SUPPORT is set.
BI_ITR_SUPPORT	Forward and reverse iterators can be requested.	GetIterator()
READ_ACCESS	Values can be read from the SDP.	GetData() - all overloads. GetDataMinMax()
WRITE_ACCESS	Values can be written to the SDP.	SetData() - all overloads.
RW_ACCESS	Equivalent to READ_ACCESS | WRITE_ACCESS	GetData() - all overloads. GetDataMinMax() SetData() - all overloads.

Table 5. SDP behavior flags and their relevant functions.

C++ API functions Descriptions

The following are brief summaries of the functions associated with the Spatial Data Provider interface. For a more detailed description of these functions, see the OME source code, or the auto-generated Doxygen documentation.

AddFieldCol(<label>, <...>) - Add a new column/record field with with **<label>** and initialized according to the additional arguments in **<...>**, which vary depending on which overloaded function is being used.

GetCapabilities() - Queries the SpatialDataProvider for its supported capabilities, which returns any combination of the flags in the above table.

GetCellSize(<width>,<height>) - On return, sets **<width>** and **<height>** to a grid cell's standard width and height, or returns false if SDP is not representing a gridded coverage.

GetColumnCount() - Returns the total number of fields per spatial record. Required for all implementations.

GetData(<record index>,<attribute index>,<value>) - If both **<record index>** and **<attribute index>** are valid, populates **<value>** on return. Otherwise, returns false.

GetDataMinMax(<attribute index>,<minValue>,<maxValue>) - If **<attribute index>** is a valid index **<minValue>** and **<maxValue>** are populated by the minimum and maximum values found in that attribute across all records, respectively. If **<attribute index>** is -1, the minimum and maximum values across all attributes and records are retrieved. If **<attribute index>** is invalid and not -1, false is returned.

GetExtents(<xMin>, <xMax>, <yMin>, <yMax>) - Populates **<xMin>**, **<xMax>**, **<yMin>**, and **<yMax>** with the minimum and maximum extents of of the x and y axes. Required for all implementations.

GetFieldCol(<label>) - Retrieve the index of the column who is named **<label>** or -1 if no column with that name exists. Required for all implementations.

GetIterator(<kind>) - returns an iterator that moves either forward or backward through records based on the type of iterator specified by **<kind>**.

GetNeighborCount(<record index>) - Returns the total number of neighbors for the record at **<record index>**.

GetNeighborRecord(<record index>, <neighbor>) - Returns the record index of the nth **<neighbor>** of the record at **<record index>**. This can be used in conjunction with **GetNeighborCount()** to iterate through all neighboring records.

GetNeighbors(<record index>,<count>) - Returns a list of indices for the total number of neighbors for the record at **<record index>**, with **<count>** containing the total number of indices returned.

GetNextTo(<record index>,<count>) - retrieves an iterator to a record of all neighbors of the record at **<record index>**, optionally populating **<count>** with the total number of records found.

GetRowCount() - Returns the total number of spatial records. Required for all implementations.

GetWithin(<record index>,<query>,<distance>) - Retrieve an iterator to records within **<distance>** from the record at **<record index>** which satisfy **<query>**, whose syntax is SDP-dependent.

Load(<filename>,<optional expression parser>) – Attempts to load parser data from a file pointed to by **<filename>**, with an optional expression parser being passed in as well. Returns a flag indicating success or failure.

NextTo(<index>, <query>) - Returns true if the spatial component at **<index>** has any neighbors that satisfy the **<query>** whose syntax is SDP-dependent. Otherwise, returns false.

NextToArea(<index>,<query>) - Return the total area of all neighbors of the record at **<index>** that satisfy **<query>**, whose syntax is SDP-dependent.

Save(<filename>,<flags>) - Attempts to save data to a file at **<filename>**, using any optional **<flags>** that are interpreted by the specific SDP. Returns a flag indicating success or failure in saving the file.

SetData(<record index>,<attribute index>,<value>) - Set the value of the attribute at **<attribute index>** for the record at **<record index>** to **<value>**.

Within(<index>,<query>,<distance>) - Returns true if any records within **<distance>** from the record at **<index>** satisfy **<query>**, whose syntax is SDP-dependent. Otherwise, returns false.

WithinArea(<index>,<query>,<distance>) - Returns the total are of all records within **<distance>** from the record at **<index>** that satisfy **<query>**, whose syntax is SDP-dependent.

Model Component Expression SDP Functions

The following functions are for use within the expressions used to for updating model component values. All expression functions that interact directly with the loaded SDP are prefixed with "SDP". A "spatial representation unit" refers to the base unit of representing a section of space; typically this is a grid cell or a polygon.

Function	Usage Group(s)
SDPGetBooleanData	3
SDPGetCapabilityFlags	1
SDPGetCellSize	1
SDPGetColumnCount	1
SDPGetExtents	1
SDPGetFieldCol	2
SDPGetDataMinMax	1,2
SDPGetIntData	3
SDPGetNumberData	3
SDPGetRowCount	1
SDPGetStringData	3
SDPListCapabilities	1
SDPNextTo	4
SDPNextToArea	4
SDPNextToIDs	4
SDPNextToValues	4
SDPSetData	3
SDPWithin	4
SDPWithinArea	4

Table 6. SDP model expression functions and their usage groups. The usage groups are: 1) querying details about the whole coverage, 2) querying about details for a specific attribute across all spatial records, 3) getting and/or setting values specific attributes in a specific spatial record, and 4) querying about spatial relations between spatial records.

SDPGetBooleanData(<row>,<column>) - Return boolean value for **<column>** in **<row>**, or nil/NULL if it doesn't exist.

SDPGetCapabilityFlags() - Return the flag markers for capabilities of SDP. See Table 5 for a list of possible flags.

SDPGetCellSize() - Return the extents (width, height) for a given cell in a gridded coverage.

SDPGetColumnCount() - Return the number of columns of attributes within the SDP.

SDPGetExtents() - Return the four values representing the extents of the spatial coverage: the minimum x-value, the maximum x-value, the minimum y-value, and the maximum y-value.

SDPGetFieldCol(<label>) - Return index of column with the header matching **<label>**, or -1 if no column matches.

SDPGetDataMinMax(<index>) - Return the minimum and maximum values within the column at **<index>**, or the minimum and maximum values for all numeric columns in the coverage if **<index>** is -1.

SDPGetIntData(<row>,<column>) - Return the integer value for the attribute in **<column>** and **<row>**, or nil/NULL if it doesn't exist.

SDPGetNumberData(x,y) - Return the floating-point value for the attribute in **<column>** and **<row>**, or nil/NULL if it doesn't exist.

SDPGetRowCount() - Return the number of rows in spatial data provider.

SDPGetStringData(<row>,<column>,<maxChars>) - Return a string value representation for the attribute in **<column>** and **<row>**, limiting the length of the returned string to **<maxChars>**. If the record or value does not exist, nil/NULL is returned instead.

SDPListCapabilities() - Return A string listing all flagged capabilities.

SDPNextTo(<row>,<query>) - Return true if any neighbors next to the spatial representation unit at **<row>** validate **<query>**.

SDPNextToArea(<row>) - Return the total area of all neighbors next to the spatial representation unit at **<row>**.

SDPNextToIDs(<row>) - Return array of row ids for all neighbors of the spatial representation unit at **<row>**.

SDPNextToValues(<row>,<label>) - Return array of values for the attribute column whose header matches **<label>** for all neighbors next to the spatial representation unit at **<row>**.

SDPSetData(<row>,<column>,<value>) - Set value of the attribute in **<column>** for **<row>** to **<value>**.

SDPWithin(<row>,<query>,<dist>) - Return true if any spatial representation units are within a distance of **<dist>** from the spatial representation unit at **<row>** validate **<query>**.

SDPWithinArea(<row>,<query>,<dist>) - Return area of all spatial representation units are within a distance of **<dist>** from the spatial representation unit at **<row>** validate **<query>**.

Appendix C. OME File Format Specifications

OME makes use of two custom XML files: OME model file (.omem), which contains a model's definition and parameter linkages, and the OME control file (.omec), which contains information pertinent to running a model, linking to external data sources, and displaying results. What follows is some general documentation of the two xml specifications. Please note that OME is still under development, and that the file specification may have changed as of this writing. For a more in-depth description of these XML specifications, see the omem.xsd and omec.xsd XML schema files in the OME source code distribution.

OME Model files (.omem)

The OME model file contains the declaration of the overall model structure and any interlinkages between components. The root element of the model file is **<ome>**, and the general tag structure is outlined in Figure 11 below.

```
<ome>
    <param_file>
    <model>
        <description>
        <tables>
            <table_data>*
            <interp_table_data>*
        <enumerations>
            <enum>*
                <val>*
        <variables>
```

Figure 11. Tag hierarchy for a typical .omem file. Bold tags are required, while italicized tags are optional. Tags followed by an asterisk (*) may occur more than once within their positions. The ellipse (…) at the end of the hierarchy represents the recursive nature of the **<model>** tag, as it may contain any number of **<model>** tags within itself.

```
        <variable>*
            <description>
        <vararray>*
            <description>
        <varts>*
            <description>
    <states>
        <state>*
            <description>
    <flows>
        <flow>*
            <description>
    <influences>
        <influence>*
            <description>
    <labels>
        <label>*
            <description>
    <iterators>
        <iterator>*
            <description>
    <modelports>
        <port>
            <description>
            <subsources>
                <src>*
            <subtargets>
                <trg>*
            <outsources>
                <src>*
            <outtargets>
                <trg>*
    <assocports>
        <assoc_port>
            <description>
            <subsources>
                <src>*
            <subtargets>
                <trgs>*
            <outsources>
                <src>*
```

Figure 11 (Continued).

```
            <outtargets>
                <trg>*
        <assocs>
            <assoc>*
                <description>
        <spawners>
            <spawner>*
                <description>
        <submodels>
            <model>*
                …
```

Figure 11 (Continued).

<assoc> Link which defines the intersecting relationship between two submodels; analogous to Simile's Role arrow. Attributes are listed in table 7.

Attribute	Required	Description
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
in_object	Yes	Source object of the submodel association.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
origin	No	Origin in OME drawing system.
out_object	Yes	Target object of the submodel association.
source	No	ID of link's source object.
target	No	ID of link's target object.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 7. Assoc tag attributes.

<assoc_port> Point at which a submodel association links to a submodel. Used for when a submodel association links two submodels that are separated by several nested submodels. Attributes are listed in table 8.

Attribute	Required	Description
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
in_object	Yes	Source object of the submodel association.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
out_object	Yes	Target object of the submodel association.
source	No	ID of link's source object.
target	No	ID of link's target object.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 8. Assoc_port tag attributes.

<assocports> Collection of **<assoc_port>** tags in a submodel. No attributes.

<assocs> Collection of **<assoc>** tags in a submodel. No attributes.

<description> Human readable annotation of model object. No attributes.

<enum> An enumerated collection type within the containing submodel. Attributes are listed in table 9.

Attribute	Required	Description
name	Yes	The name of the enumerated class.

Table 9. Enum tag attributes.

<enumerations> Collection of **<enum>** tags in a submodel. No attributes.

<flow> Element representing a System Dynamics model Flow arrow. Attributes are listed in table 10.

Attribute	Required	Description
ctrl_pt1	No	Control point influencing shape of curve.
ctrl_pt2	No	Control point influencing shape of curve.
id	Yes	Universally unique identifier of model element.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
source	No	ID of link's source object.
target	No	ID of link's target object.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 10. Flow tag attributes.

<flows> Collection of **<flow>** tags in a submodel. No attributes.

<influence> Element representing a System Dynamics model influence arrow. Attributes are listed in table 11.

Attribute	Required	Description
ctrl_pt1	No	Control point influencing shape of curve.
ctrl_pt2	No	Control point influencing shape of curve.
id	Yes	Universally unique identifier of model element.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
role	No	Presently serves no purpose (deprecate?).
source	No	ID of link's source object.
target	No	ID of link's target object.
units	No	Units of measure associated with element.
use_curr_val	No	If true, retrieve the value from the source object before it updates, rather than after.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 11. Influence tag attributes.

<influences> Collection of **<influence>** tags in a submodel. No attributes.

<interp_table_data> Instance of table data which defines ranges for interpolation; analogous to Simile's graph construct. Attributes are listed in table 12.

Attribute	Required	Description
bound_mode	Yes	Flag denoting how to deal with values that lie outside of the table's bounds (Document flags?).
column	No	The column from source CSV file to pull values from.
data	Yes	Cached values to use to populate table in case source file cannot be located.
dimensions	Yes	The dimensions of the table; either 1D or 2D.
filename	No	The name or path to the file used to populate the table.
id	Yes	Unique identifier of table within parent submodel.
interp_mode	Yes	Flag denoting method of interpolation between anchored values (Document flags?).
lower_bound	Yes	Lower bound of the range of values encapsulated by table.
upper_bound	Yes	Upper bound of the range of values encapsulated by table.

Table 12. Interp_table_data tag attributes.

<iterator> Stand in for Simile's Alarm component; presently only partially implemented. Attributes are listed in table 13.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
expr	Yes	Expression used to update element value.
extents	No	The width and height in graphical units of the element.
external_init	No	If true, element expect an initial value from an external source (such as a parameter file).
id	Yes	Universally unique identifier of model element.
init_only	No	If true, element is only evaluated during the initialization processes.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
max_value	No	Maximum expected value of element.
min_value	No	Minimum expected value of element.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 13. Iterator tag attributes.

<iterators> Collection of **<iterator>** tags in a submodel. No attributes.

<label> Simple label object to annotate visual representations of model. Attributes are listed in table 14.

Attribute	Required	Description
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
origin	No	Origin in OME drawing system.
text	No	Text to apply to label (Deprecate and move from attribute to element?).
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 14. Label tag attributes.

<labels> Collection of **<iterator>** tags in a submodel. No attributes.

<model> Component that represents a discrete unit of encapsulation of model components. Models can optionally represent multiple instances. Attributes are listed in table 15.

Attribute	Required	Description
expects_spatial	No	Hint to OME tools that a submodel expects to have its initial number of instances set by a Spatial Data Provider.
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
init_instances	No	The submodel's initial number of instances.
inner_box	No	The bounding scale within which to draw OME submodel components.
int_method	No	The submodel's preferred integration method.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
origin	No	Origin in OME drawing system.
step_size	Yes	The submodel's preferred temporal step size.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 15. Model tag attributes.

<modelports> Collection of **<port>** tags in a submodel. No attributes.

<ome> Root element of the .omem file. No attributes.

<outsources> Collection of links that point into a **<port>** or **<assoc_port>** from outside of its parent submodel. No attributes.

<outtargets> Collection of links that point away from a **<port>** or **<assoc_port>** from outside of its parent submodel. No attributes.

<param_file> Identical to the **<param_file>** tag in the .omec file, and can be used to override parameter mappings in the linked model file. Attributes are listed in table 16.

Attribute	Required	Description
filepath	Yes	Path to parameter file (either .spf or .csv)
target_model_path	No	Path through model heirarchy to target root model for parameter file. Defaults to the root model.

Table 16. Param_file tag attributes.

<port> Interface between model components that exist on either side of a submodel boundary. Attributes are listed in table 17.

Attribute	Required	Description
evaluated	No	If true, the aggregate value of all model components (and all their instance values) linked to the Submodel port are recorded during an update step.
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	No	Human-readable label of object.
origin	No	Origin in OME drawing system.
source	No	ID of link's source object.
target	No	ID of link's target object.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 17. Port tag attributes.

<spawner> Model element used to spawn or kill submodel instances; analogous to Simile's channel model components. Attributes are listed in table 18.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
expr	Yes	Expression used to update element value.
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
init_only	No	If true, only evaluated on initialization.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
is_conditional	No	If true, Spawner is evaluated to determine whether a given submodel instance should be evaluated at a given timestep.
is_loss	No	If true, Spawner subtracts instances rather than adding them.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
per_instance	No	If true, Spawner is evaluated for each submodel instance; otherwise it is only evaluated once per update.
stochastic	No	If true, a degree of randomness is applied to the Spawner's value accumulation process.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 18. Spawner tag attributes.

<spawners> Collection of **<spawner>** tags in a submodel. No attributes.

<src> Reference to source object pointing to the containing **<port>** or **<assoc_port>**. Attributes are listed in table 19.

Attribute	Required	Description
name	No	ID of referenced object.

Table 19. Src tag attributes.

<state> Represents a State Variable/Compartment/Stock in a System Dynamics model; analogous to Simile's Compartment. Attributes are listed in table 20.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
expr	Yes	Expression used to update element value.
extents	No	The width and height in graphical units of the element.
id	Yes	Universally unique identifier of model element.
init_condition	No	The starting value of the State Variable.This is necessary if no influence is used to retrieve an initialization value.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 20. State tag attributes.

<states> Collection of **<state>** tags in a submodel. No attributes.

<submodels>Collection of **<model>** tags in a submodel. This results in a recursive tree of submodels, each defined by a **<model>** tag. No attributes.

<subsources> Collection of links that point into a **<port>** or **<assoc_port>** from inside of its parent submodel. No attributes.

<subtargets> Collection of links that point away from a **<port>** or **<assoc_port>** from inside of its parent submodel. No attributes.

<table_data> Instance of table data to be linked with an Evaluable model element. Attributes are listed in table 21.

Attribute	Required	Description
column	No	The column from source CSV file to pull values from.
data	Yes	Cached values to use to populate table in case source file cannot be located.
dimensions	Yes	The dimensions of the table; either 1D or 2D.
filename	No	The name or path to the file used to populate the table.
id	Yes	Unique identifier of table within parent submodel.

Table 21. Table_data tag attributes.

<tables> Collection of **<table_data>** and **<interp_table_data>** tags in a submodel. No attributes.

<trg> Reference to target object pointing away from the containing **<port>** or **<assoc_port>**. Attributes are listed in table 22.

Attribute	Required	Description
name	No	ID of referenced object.

Table 22. Trg tag attributes.

<val> A value representation belonging to an enumerated (**<enum>**) type. Attributes are listed in table 23.

Attribute	Required	Description
name	Yes	The name/value for an enumerated value.

Table 23. Val tag attributes.

<vararray> Special variable that defines an n-dimensional array of values across a single instance. Attributes are listed in table 24.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
dimensions	No	List of dimensions defining how to access the VarArray's values.
expr	Yes	Expression used to update element value.
extents	No	The width and height in graphical units of the element.
external_init	No	If true, element expect an initial value from an external source (such as a parameter file).
id	Yes	Universally unique identifier of model element.
init_only	No	If true, element is only evaluated during the initialization processes.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
max_value	No	Maximum expected value of element.
min_value	No	Minimum expected value of element.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
value	No	List of whitespace-delimited values to populate Vararray with. Can be empty.
version	No	Informational; the current revision number for the element.

Table 24. Vararray tag attributes.

<variable> Represents an intermediate value which affects model dynamics. Attributes are listed in table 25.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
expr	Yes	Expression used to update element value.
extents	No	The width and height in graphical units of the element.
external_init	No	If true, element expect an initial value from an external source (such as a parameter file).
id	Yes	Universally unique identifier of model element.
init_only	No	If true, element is only evaluated during the initialization processes.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
max_value	No	Maximum expected value of element.
min_value	No	Minimum expected value of element.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 25. Variable tag attributes.

<variables> Collection of **<variable>**, **<vararray>**, and **<varts>** tags in a submodel. No attributes.

<varts> Special variable that updates its value over time instead of using an equation. Attributes are listed in table 26.

Attribute	Required	Description
as_int	No	If true, value is to be treated as an integer instead of a floating point number.
ask_val	No	If true, element will request a parameter value before constructing its own.
extents	No	The width and height in graphical units of the element.
external_init	No	If true, element expect an initial value from an external source (such as a parameter file).
fixed_val	No	Value to use with the "fixed" mode.
id	Yes	Universally unique identifier of model element.
init_only	No	If true, element is only evaluated during the initialization processes.
internal	No	Denotes element's visibility to external tools; if true, element is hidden from outside tools.
interval	No	The time series update interval.
label_origin	No	Origin of label in OME drawing system.
last_modified	No	Informational; date of element's last modification.
max_value	No	Maximum expected value of element.
min_value	No	Minimum expected value of element.
mode	No	Method by which to resolve values between interval steps. Allowed values are “nearest”, “interpolate”, “last”, and “fixed”.
name	Yes	Human-readable label of object.
origin	No	Origin in OME drawing system.
table	No	ID of associated EvalTable or EvalInterpTable.
units	No	Units of measure associated with element.
validated	No	If true, object has been validated (Deprecate?).
version	No	Informational; the current revision number for the element.

Table 26. Varts tag attributes.

OME Control files (.omec)

The OME control file is responsible for details pertaining to setting up and executing a model declared in a .omem file. The general tag structure is outlined in Figure 12 below.

Figure 12. Tag hierarchy for a typical .omec file. Bold tags are required, while italicized tags are optional. Tags followed by an asterisk (*) may occur more than once within their positions.

Detailed Element descriptions

<bp> An entry for a single break point when running a model using interpreted expressions. Attributes are listed in table 27.

Attribute	Required	Description
enabled	Yes	If true, break on the associated line; otherwise, do nothing.
line	Yes	The line to break on.

Table 27. Bp tag attributes.

<coverage_mapping> Provides details on how to map the targeted model's components to a spatial coverage's fields/columns. This entry can still be present when **<spatial_provider>** when OME is operating as a plugin to another model simulation tool since it is expected that the parent tool will take care of any coverage linkages. No attributes.

<debug> This section contains information for dumping evaluation scripts and breakpoints to activation throughout a run. Only applies to simulation runs in interpretive mode; compiled model runs are not supported. No attributes.

<dump_eval_batch> Dump the update script generated during an interpretive model run. The default path is “evalBatch.lua”. Attributes are listed in table 28.

Attribute	Required	Description
path	No	Path to batch dump save location.

Table 28. Dump_eval_batch tag attributes.

<dump_init_batch> Dump the initialization script generated during an interpretive model run. The default path is “initBatch.lua”. Attributes are listed in table 29.

Attribute	Required	Description
path	No	Path to batch dump save location.

Table 29. Dump_init_batch tag attributes.

<eval_break> Breakpoints to apply to generated update script during interpretive model run. Break points can either be specified in a separate file or directly in the element using the

<bp> tag. Attributes are listed in table 30.

Attribute	Required	Description
path	No	Path to text file containing whitespace-delimited line numbers.

Table 30. Eval_break tag attributes.

<field> Individual entry for a represented results view field. Attributes are listed in table 31.

Attribute	Required	Description
color	Yes	RGB channel triplet defining the field's color; each channel value falls in [0,1].
id	Yes	The ID of the model component to include.

Table 31. Field tag attributes.

<fields> A collection of **<field>** elements. No attributes.

<init_break> Breakpoints to apply to generated initialization script during interpretive model run. Break points can either be specified in a separate file or directly in the element using the **<bp>** tag. Attributes are listed in table 32.

Attribute	Required	Description
path	No	Path to text file containing whitespace-delimited line numbers.

Table 32. Init_break tag attributes.

<inst_map> The mapping of a specific instance of the parent **<var>** value to a Spatial coverage column. Attributes are listed in table 33.

Attribute	Required	Description
index	Yes	1-based index of specific instance value to assign to sdp_column.
sdp_name	Yes	The name of the field in SDP.

Table 33. Inst_map tag attributes.

<multival_view> View type which shows all values for a single component. No attributes.

<ome_ctrl> The root element. No attributes.

<ome_model> Provides path to .omem file to execute, as well as overrides for start time, stop time, time step, and report interval. This tag also optionally provides a path to the compiled version of the model, if it exists. Attributes are listed in table 34.

Attribute	Required	Description
compiled_name	No	Path to a compiled version of the model, minus the file extension (.dll or .dynlib).
filepath	Yes	Path to the .omem model definition file.
interval	No	Override Model's step interval.
relative_step_size	No	Override Model's relative step size.
report_interval	No	Override Model's report interval.
start_time	No	Override Model's start time.
step_size	No	Override Model's step size.
stop_time	No	Override Model's stop time.
time_units	No	Temporal units to apply to model.

Table 34. Ome_model tag attributes.

<param_file> Identical to the **<param_file>** tag in the .omem file, and can be used to override parameter mappings in the linked model file. Attributes are listed in table 35.

Attribute	Required	Description
filepath	Yes	Path to parameter file (either .spf or .csv)
target_model_path	No	Path through model heirarchy to target root model for parameter file. Defaults to the root model.

Table 35. Dump_eval_batch tag attributes.

<results_views> A collection of sub-elements which outline details for different output views. This section is typically only used by environments that have advanced methods for displaying results (such as OMESimRunner). Attributes are listed in table 36.

Attribute	Required	Description
selected_only	No	If true, only display selected fields.

Table 36. Results_views tag attributes.

<solver> Provides details on preferred solving method and the preferred solver variant. Attributes are listed in table 37.

Attribute	Required	Description
method	Yes	Integration solving method; either “euler” or “rk4”.
type	No	The solver variant to use; either “default”, or “hires”.

Table 37. Solver tag attributes.

<spatial_provider> Provides a path to an SDP's dynamic library and provide additional initialization arguments to be passed on to the SDP module, if necessary. Attributes are listed in table 38.

Attribute	Required	Description
library_path	Yes	Path to dynamic library storing the SDP.

Table 38. Spatial_provider tag attributes.

<submodel> Identifies any relationships that exist between the submodel and any SDP fields within the **<coverage_mapping>** tag. Attributes are listed in table 39.

Attribute	Required	Description
id	Yes, if name is absent	The unique id of the sub-model.
instance_per_unit	No	If true, the model component only updates the SDP field.
name	Yes, if id is absent	The human-readable name of the sub-model.

Table 39. Submodel tag attributes.

<tree_view> Details for an instance of a tree results view, which lays out the included fields in a structure that reflects the model hierarchy. Attributes are listed in table 40.

Attribute	Required	Description
selected_only	No	If true, only display selected fields.
sort_dir	No	Sort direction; 0=ascending, 1=descending.
sort_mode	No	Sorting components and submodels; 0=components first, 1=models first, 2=mixed.

Table 40. Tree_view tag attributes.

<var> Mapping between SDP field and submodel component. Attributes are listed in table 41.

Attribute	Required	Description
bidirectional	No	If true, updates to one value in the var pair updates the other.
mapto_id	Yes, if mapto_name is absent.	Unique id of sub-model component.
mapto_name	Yes, if mapto_id is absent.	Human-readable name of sub-model component
sdp_name	Yes	The name of the field in SDP.
update_only	No	If true, the model component only updates the SDP field.

Table 41. Var tag attributes.

Appendix D. OME Compatibility With Simile

For OME to be of any initial use, the projected needed to target models that already existed. Both reference models were written and executed in the Simile modeling framework (see the Literature Review section for a brief description of Simile). Since all model update expressions and model components beyond the basic universal System Dynamics pieces were Simile-centric, it is pertinent to reimplement behaviors and structural pieces that the models rely on to work. What follows is a description of how the Simile's model structure meshes with that defined by OME, the extent of OME model expression function compatibility with that found in Simile, and a few functions defined for OME's expression syntax which have no equivalent in Simile.

Mapping Simile Model Components to the Equivalent in OME

Table 42 shows how Simile model components map to their equivalents in OME. The compatibility between Simile and OME components has fluctuated over time, with the biggest shift occurring when focus on supporting dynamic allocation and deallocation of submodel instances was reduced in priority in favor of ensuring that the static model behavior in the reference models would be sufficiently supported within a reasonable timeframe. Increasing compatibility with Simile, as well as adding support for System Dynamics models that originate from other environments, are goals for future development.

Simile Component	Supported?	OME Equivalent	Notes
Alarm	No	IterConditional	Implementation was started, but never completed.
Compartment	Yes	StateVar	
Condition	Yes	Spawner	Functionality consolidated into Spawner class.
Creation Process	Partial	Spawner	Functionality consolidated into Spawner class. Functionality is present, but likely broken; has not been tested in a while.
Destruction Process	Partial	Spawner	Functionality consolidated into Spawner class. Functionality is present, but likely broken; has not been tested in a while.
Event	No	N/A	Discrete Event injection is not supported.
Flow	Yes	Flow	
Ghost	No	N/A	Not relevant to model runtime. Ghost connections are mapped to represented model component by SMLConverter. There are plans to implement an OME equivalent to this for result summary purposes, but it has yet to be implemented.
Immigration Process	Partial	Spawner	Functionality consolidated into Spawner class. Functionality is present, but likely broken; has not been tested in a while.
Influence	Yes	Influence	
Reproduction Process	Partial	Spawner	Functionality consolidated into Spawner class. Functionality is present, but likely broken; has not been tested in a while.
Role Arrow	Partial	SubmodelAssoc	Functionality is present, but has not been thoroughly tested.
Squirt	No	N/A	Discrete Event injection is not supported.
State	No	N/A	Discrete Event injection is not supported.

Table 42. Simile model components as supported by OME. The list of model components are taken from Simile 6.x.

Simile Component	Supported?	OME Equivalent	Notes
Submodel	Yes	Model	Most, if not all, functionality should be present.
Text	Yes	SimpleLabel	A container for text-only labels is present, in anticipation of a future System Dynamics diagram canvas, but presently is not used since Text labels have no bearing on model dynamics.
Variable	Yes	Variable VarArray TimeSeriesVar	Variable types are broken up by functionality. Variables representing a collection of values are represented by VarArrays. Variables that change their value solely based on simulation time are represented by TimeSeriesVars. All other Variable components are represented by Variables.

Table 42 (Continued).

Simile Expression Support in OME

Since OME is expected to understand Simile-derived System Dynamics models, most, if not all expression functions in Simile would need to be reimplemented in OME. This was done mostly through trial-and-error, with testing primarily focused on the functions used in reference models. Table 43 outlines the degree of compatibility between Simile and OME expression functions.

Simile Function	OME Compatibility	Notes
abs	Yes	Uses C implementation.
acos	Yes	Uses C implementation.
all	Yes	
any	Yes	
asin	Yes	Uses C implementation.
at_init	Yes	Implemented, but not in the most efficient way.
atan	Yes	Uses C implementation.
atan2	Yes	
binome	Yes	Uses C++11 STL implementation.
ceil	Yes	
channel_is	Yes	Minimally tested.
colin	Yes	Minimally tested.
const_delay	Yes	
cos	Yes	Uses C implementation.
cosh	Yes	Uses C implementation.
count	Yes	
default	Yes	Untested.
dies_of	Yes	Untested.
dt	Partial	Just returns step size, which in OME is static throughout simulation and across sub-model components.
element	Yes	
exp	Yes	Uses C implementation.
factorial	Yes	May break with large numbers.
first	Yes	Untested.
firsttrue	Yes	OME returns 0 if no element is evaluated to true; this may not be the same behavior as Simile.
floor	Yes	
fmod	Yes	Uses C implementation.
following	Yes	Untested.
for_members_of_type	No	Unknown behavior.
gaussian_var	Yes	Uses C++11 STL implementation of a normal distribution.
graph	Yes	Directly maps to InterpTable() function in OME, as the function technically interpolates across a table of values.
greatest	Yes	
howmanytrue	Yes	Minimally tested.
hypergeom	Yes	Custom implementation; minimally tested.
hypot	Yes	Uses right-triangle formula.
in_preceding	No	
in_progenitor	No	
index	Yes	

Table 43. Simile expression function support in OME.

Simile Function	OME Compatibility	Notes
init_time	Partial	Presently just grabs start time from SimManager; this will need to be changed when dynamic model instance allocation is reintroduced.
int	Yes	
interpolate	Yes	Untested.
iterations	No	Linked to Alarm components in Simile. The equivalent is not currently implemented in OME.
last	Yes	Untested.
least	Yes	
log	Yes	Uses C implementation.
log10	Yes	Uses C implementation.
makearray	Yes	May be missing some Simile behavior.
max	Yes	
min	Yes	
ordinals	Yes	Untested.
parent	Yes	Untested.
pi	Yes	Implemented as constant.
place_in	Yes	Minimally tested.
poidev	Yes	Uses C++11 STL implementation of a Poisson distribution.
posgreatest	Yes	Minimally tested.
posleast	Yes	Minimally tested.
pow	Yes	Uses C implementation.
preceding	Yes	Untested.
prev	Yes	
product	Yes	
product3	Yes	Untested.
rand_const	Partial	Presently just calls rand_var, returning a different random value each time.
rand_var	Yes	Uses C++11 STL implementation of a uniform distribution
rankings	Yes	Minimally tested.
round	Yes	
sgn	Yes	
sin	Yes	Uses C implementation.
sinh	Yes	Uses C implementation.
size	Yes	Untested.
sqrt	Yes	Uses C implementation.
stop	Partial	In OME, reports error to event system, which then broadcasts the errors for any event handlers listening. More testing is needed.

Table 43 (Continued).

Simile Function	OME Compatibility	Notes
subtotals	Yes	Minimally tested.
sum	Yes	
table	Yes	
tan	Yes	Uses C implementation.
tanh	Yes	Uses C implementation.
time	Yes	
transform3	Yes	Untested.
var_delay	Yes	OME does not impose the same restrictions on var_delay() as Simile does; this may change in the future.
with_colin	Yes	Untested.
with_greatest	Yes	Untested.
with_least	Yes	Untested.

Table 43 (Continued).

OME-specific functions

The following are a handful of OME-specific expression functions which have no equivalent Simile implementations.

valuesFromInstances(<variable>,<inds>) - Retrieves a list of values for **<variable>** populated with the value from the parent model instances pointed to by the indexes in the list **<inds>**.

getAsTable(…) - Takes a variable number of arguments and returns a list object which has packaged the values.

upgroup(<variable>,<level>) - Retrieve a list of values for **<variable>** by up-scoping **<level>** number of submodels. Primarily used to reconcile operations on model components that exist at different submodel depths

omecleanup() - Deallocation function which should not be called directly.