EPA Public Access

Author manuscript

Environ Model Softw. Author manuscript; available in PMC 2018 August 01.

About author manuscripts

Submit a manuscript

Published in final edited form as:

Environ Model Softw. 2017 August 1; 94: 48–62. doi:10.1016/j.envsoft.2017.03.012.

Modular and Spatially Explicit: A Novel Approach to System Dynamics

Patrick Wingo^a, Allen Brookes^a, and John Bolte^b

^aUS Environmental Protection Agency, Western Ecology Division, Corvallis, OR USA

^bOregon State University, Corvallis OR USA

Abstract

The Open Modeling Environment (OME) is an open-source System Dynamics (SD) simulation engine which has been created as a joint project between Oregon State University and the US Environmental Protection Agency. It is designed around a modular implementation, and provides a standardized interface for interacting with spatially explicit data while still supporting the standard SD model components. OME can be run as a standalone simulation or as a plugin to a larger simulation framework, and is capable of importing Models from several SD model formats, including Simile model files, Vensim model files, and the XMILE interchange format. While it has been released, OME is still under development, and a number of potential future improvements are discussed. To help illustrate the utility of OME, an example model design process is provided as an Appendix.

Keywords

System Dynamics; spatially-explicit; simulation engine; open-source; Simile; Vensim; XMILE

1. Introduction: What is OME?

The Open Modeling Environment (OME) is a System Dynamics (SD) model simulation engine that has been developed as a joint project between the department of Biological and Ecological Engineering (BEE) at Oregon State University and the Office of Research and Development of the United States Environmental Protection Agency (USEPA). At its core, OME is an engine for solving SD models using either Euler or 4th-order Runge-Kutta (RK4) integration methods. SD modeling involves characterizing, through the use of ordinary differential equations, the response of a set of system states through time [1]. These states can be thought of as discrete buckets holding a specific quantity of something that is pertinent to the question the model is trying to answer, and that describe the state of the system at a given moment. Several common SD modeling tools are Simile [2], Vensim [3], and STELLA [4]. All of these tools allow for the construction of a Stock-Flow diagram, a method of visual model construction that uses a set of standard symbols to describe the processes that drive the simulation [5]. OME does not yet have its own tools for constructing Stock-Flow diagrams, but instead relies on existing tools to create models to be run [5]. A significant goal of OME's development has been to address two shortcomings that are apparent in SD modeling tools commonly used for authoring and running SD models: 1) the

absence of an elegant means of representing spatially explicit models and associated datasets, and 2) the insular nature of many extant tools, which creates unnecessary challenges when attempting to couple them with other modeling software [5].

Many modelers have included explicit spatial relationships in the Stock-Flow diagrams used to define their SD models [6]. Traditional approaches to incorporating spatially-explicit data into SD models are tedious and taxing, often requiring the model authors to re-implement well-established spatial relationships specific for each model implementation [7]. Typically, such approaches are haphazard solutions that take advantage of a particular modeling environment's implementation of multi-value variables, which are commonly known as lists, subscripts, collections, or submodels [8][9][10]. None of these approaches to multi-value variables are part of the traditional SD model approach, and are not implemented equally across SD modeling tools [5]. Ideally, incorporating spatial data and relationships into SD models would be accomplished using a standardized interface that managed details associated with reading, writing, accessing, and querying spatial information. Such an approach would allow for the use of spatially detailed inputs and outputs without requiring the SD model to be concerned about the specific representation of space used. OME provides such a service through the Spatial Data Provider [5], discussed in a later section of this paper.

There have been previous attempts to create ways to manage spatially explicit data within the confines of a SD model. The Spatial Modeling Environment (SME) [6] is a tool that would take a model created in the SD modeling tool STELLA, and apply the model once to each cell in a spatially explicit grid [6]. Similarly, SimARC [11] represents another attempt to capture spatially explicit data by bridging the simulation engine of the SD modeling tool Simile [2] with the general Geographic Information Systems (GIS) tool ArcMap [12], allowing the SD model to be applied to each polygon in an ArcMap map layer [11]. While both SME and SimARC focus on applying a SD model as a process internal to each discrete spatial unit (be it a grid cell or polygon), OME applies aspects of a specific spatial coverage to the dynamics captured within a single SD model [5]. This approach effectively addresses a different explicit spatial usage case than is covered by SME and SimARC, targeting a different set of models and research questions [5].

The second shortcoming being addressed originates from the fact that most extant SD modeling tools are largely insular in design [5]. Cross-interaction and integration with other modeling frameworks is challenging, as there is no standard approach to tool integration, even when various SD modeling tools are largely accomplishing their simulations in similar ways [13]. While the XMILE standard is attempting to establish a common interchange format for model definitions [14], OME takes cross-interaction one step further by providing an architecture and application programming interface (API) for interacting in real time with other processes, allowing for it to be embedded as a component in other software tools [5]. Additionally, OME's source code is freely available, allowing OME to be extended or modified if the original implementation is insufficient for a given purpose or integration scenario [5].

1.1. Example Model Information

In order to demonstrate a number of the features supported by OME, a demonstration model has been created: The Simple Critter Model. A complete description of the model and a walkthrough of how spatially explicit information is incorporated into this model can be found in Appendix B; an abbreviated introduction to the model follows. This model captures the dynamics of a population moving between square sample plates in response to a stimulus. These plates can be assembled in any configuration, but can only act as traversable neighbors across one of the four edges (top, left, bottom, and right). While the configuration of the plates can be hard-coded into a model (see Appendix B), this approach is inflexible and difficult to maintain; the addition, removal, or rearrangement of sample plates would require significant restructuring of the stock flow diagram. Ideally, the dynamics represented by the model could be applied without any knowledge of the explicit spatial configuration, while still being influenced by it.

OME currently provides no visual interface for defining SD models; rather, models are defined in XML-conformant text files which are then read by the OME runtime. To simplify model creation, the Simple Critter Model was created in Simile, and then converted to OME-conformant XML. The base Simile model used in the conversion process is included in Appendix B. The conversion process involves providing a model file from Simile, Vensim [3], or any tool that supports XMILE to the OME translation tool UniversalConverter, an executable that is part of the OME package. For more information on the model conversion process, see the “Support for Multiple File Formats” subsection in the “Modular Design” section.

2. Modular Design

In order to appeal to the broadest audience possible, OME needs to provide support for a number of usage cases. Experience throughout the course of OME’s development has revealed a number of potential usage cases, such as 1) generating and exporting simulation results for later use by an external program, 2) generating results for immediate perusal by a human being, and 3) sharing intermediate results incrementally during a simulation run as a piece of a larger simulation. The architecture of OME is conducive to all of these potential usage cases.

OME is fundamentally a series of interconnected dynamically-linked libraries and executables, with different configurations used to accomplish different tasks [5]. The interconnected nature of the modules is outlined in Figure 1, and a brief description is provided for each distinct module. This approach to the framework’s architecture is useful for ensuring the efficient use of resources by only using parts necessary for the task at hand. This is exemplified by the OMEEngine, OMESimRunner, and <plugin> modules in Figure 1. OMEEngine is a simple command-line tool for running a simulation and exporting the results, satisfying the first identified usage case. OMESimRunner, by contrast, is intended to not only run a simulation, but also provide an interface for a user to organize and browse simulation results, satisfying the second identified usage case. Both OMEEngine and OMESimRunner provide the ability to export all values from all incremental timesteps in a comma separated values (.csv) file, allowing for further processing of model generated

values using external tools. The <plugin> entry represents OME configured as a plugin to a larger simulation (discussed later in this paper), which would satisfy the third identified usage case. OMEEngine and the <plugin> stand-in for external tools do not require any descriptive details about model components, so they do not link against the OMEDraw utility library. OMESimRunner, by contrast, does require access to tasks unrelated to running the simulation, as it graphically displays the results of a simulation through a series of text-based tables, so it links against OMEDraw. This modular construction also provides flexibility for future implementations; if a new usage case were to arise, such as the need to visualize OME models, most of the necessary parts already exist in the OMERuntime and OMEDraw libraries. The user-facing front end would be the only piece that would need to be added to meet the task's demands [5].

The Simple Critter Model utilizes the OMEEngine executable, the OMERuntime library, and the CSVSpatialDataProvider library modules when executed; see Figure 1 for a description of each module. The details of how the model will be executed are largely dependent on the contents of the control file, discussed in the next section.

2.1. Specifying a model in OME

OME relies on an XML-based specification for model declarations (.omem files) and control and simulation details (.omec files) [5]. Parameter values can be provided through comma separated values (.csv) files and/or Simile's .spf files [5]. The .omem file provides an intermediate model declaration that overlaps with the features that exist across a number SD model authoring tools, while the .omec file combines a model declaration with a spatial representation, simulation specifications, and an optional binary library containing the model's compiled logic [5]. Keeping the model declarations and simulation control details separate allows for simple reuse of a model under various runtime conditions. No other existing SD modeling tools support the .omem and .omec formats; for OME to be useful, it must provide its own means for accessing other model formats. OME provides support for additional file formats through the use of converters, as discussed in the next section.

Figure 2 presents the minimum declarations in an OME control file and model definition file to represent a simple SD model. The control file (b) contains the relative path to the model definition file, some execution parameters, and the integration solving method to use. The model definition file (c) essentially encodes the information of the Stock-Flow diagram (a); all diagram components are represented, complete with their update expressions and connection mappings. The Simple Critter Model has a similar configuration, but with more entries; for a full listing of these model files see the "Simple SDP Model demo" files found within the OME Github repository listed in Appendix A.

2.2. Support for multiple model formats

In order for OME to appeal to a broad number of SD modelers, tools are provided for accessing model files originating from other frameworks. The general approach taken involves providing conversion support through the UniversalConverter utility (see Figure 1). UniversalConverter takes a given model format and produces a functionally equivalent model defined in an OME model declaration file (.omem). Additionally, UniversalConverter

can modify the output model's structure to take advantage of OME-specific features through the use of Model Conversion Directives. Model Conversion Directives are instructions that the UniversalConverter will scrape out of free-text fields of individual model components, and then use this information to modify the produced model declaration as directed [5]. "Free-text" fields are tool-specific, but typically are places that allow a user to provide documentation for a specific model component; in the case of Simile, this is a model component's "Comments" field [8]. The conversion process supports Simile's model declaration formats (.sml and .pl), Vensim model declarations (.mdl) and XMILE. XMILE is an xml-based file format developed to provide a standardized SD model representation, analogous to OME's .omem files [15]. Since XMILE is much more comprehensive in the features that it covers, it may be desirable to replace OME's .omem files with XMILE while incorporating OME-specific functionality using XMILE's namespace mechanisms [14]. In the meantime, OME's conversion tools should be sufficient to handle a substantial subset of existing XMILE models.

For the Simple Critter Model, Model Conversion Directives were embedded in the Comments fields of specific model components. The included directives instructed Universal Converter to adjust the model so that the OME Spatial Data Provider would be queried when resolving spatial relationships.

Once a model has been converted to the OME format from another source, its logic can be executed either in a standalone representation or embedded into other tools.

2.3. Engine designed to run as standalone or as a plug-in

OME's module-based design separates the engine and resource management from higher level applications. This configuration allows for the simulation to run embedded in an executable (such as OMEEngine and OMESimRunner), or to be embedded in a container application (such as OMEAdapter, a plugin for the modeling framework Envision [16]). While the standalone configuration allows for OME to be directly utilized for addressing research questions through SD simulation, the plugin configuration can allow for OME to leverage its strengths in a larger simulation where a combination of simulation engines can be used to address different facets of larger research questions, such as having OME simulate simple material flows while having a GIS model handle the derivation of complex spatial statistics. The plugin configuration can also be used to assist with visualizing results from OME, should the hosting environment have relevant visualization capabilities. Both configurations treat OME's SimManager class as a contract for intended behavior; all directives and requests for information with regard to running and maintaining a simulation are managed through the SimManager's Application Programming Interface (API). When intended to be implemented as part of a plugin, OME's OMERuntime library is expected to be linked against one or more classes that act as a bridge between the larger environment's plugin API and OME's SimManager. Behaviorally, the biggest difference between OME implemented as a standalone executable and as a plugin is event management. When OME is configured for a standalone executable, it is in charge of when a simulation starts, steps, and stops. When configured as a plugin, however, some event control is ceded to the parent tool, as it likely has a command and control structure of its own which will want to exert

complete control of when a simulation starts, stops and exchanges information. The SimManager class provides a means to manage both independent and dependent events for an OME simulation by exposing an API that allows for posting control messages that will be interpreted as appropriate.

Since it is intended to be a standalone model, the Simple Critter Model will be run from the OMEngine executable when it is time to generate simulation results. However, if a user wanted to run the Simple Critter Model as part of a larger simulation environment, they would need to implement a wrapper that communicates with the SimManager class through the exposed API. Namely, the wrapper would need to at minimum instantiate a copy of SimManager with the duration and step size of the simulation, call SimManager's PreloadSubmodels() function during the plugin's initialization call, call SimManager's RunSimulation() function when the plugin is ready to run the simulation, and finally request values from the simulation by calling SimManager's GetReadableValue() function. Finer-grain control can be achieved by passing commands through SimManager's PostEvent() function. See OME's source code documentation for more information.

2.4. Example: OMEAdapter: OME as a plugin for Envision

Envision [16] is a spatially explicit simulation platform for modeling coupled human and natural systems. Since every model research project uses different constraints and assumptions, Envision supports the concept of “plug-ins”, software components that are loaded at runtime to provide application-specific functionality to a given Envision model [16]. These components interact with Envision through a straightforward plugin interface. To demonstrate embedding OME-based models into an Envision project, an Envision plugin, OMEAdapter, was developed. This plugin bridges between Envision and OME runtime components, and was used to develop and exercise OME's plugin-associated behaviors [5].

OMEAdapter conforms to Envision's plugin module interface, allowing it to be linked in as an autonomous process [5]. OMEAdapter, in turn, links against the OMERuntime dynamic library, and acts as a mediator between Envision's event messaging system and OME's event messaging system [5]. The message flow is as follows: Envision tells the OMEAdapter to initialize the OME model for a simulation run. Once initialization of the OME model is complete, a simulation can be run. Envision controls this process, sending an event message to OMEAdapter instructing OME to run the model for a fixed simulation interval. When this occurs, OMEAdapter imports any shared state values from Envision and applies them to the OME model. OMEAdapter tells the simulation engine in OMERuntime to run for the specified interval, and then pause and return control to OMEAdapter. Once the OME engine completes this step, OMEAdapter updates the shared values in Envision. When Envision is notified that the OME data has been updated, it can then process the data however it wants, including incorporating the new values into a spatial visualization [5].

Configuring an OME model to run as an autonomous process in Envision is straightforward, as demonstrated in Figure 3. First, there needs to be an autonomous process entry in an Envision Project file (.envx) that points to the OMEAdapter DLL, and has the path to an OME control file as its sole initialization argument. The control file, in turn, must have a coverage_mapping entry that describes how spatial fields are shared between Envision and

the OME; this entry is identical to any entry used by a Spatial Data Provider implementation [5], which is described below.

3. Support for Explicit Spatial representation

As previously mentioned, one of the primary drivers in OME's development was to create a standard method for representing explicit spatial relationships in a way that is accessible by a SD model, but is not limited by the SD paradigm. OME's accomplishes this through the use of the Spatial Data Provider (SDP). The SDP is a set of functions that can query and retrieve explicit information about a supplied spatial coverage using a well-defined programming interface [5]. The SDP also exposes direct mappings between submodels and variables and attributes of spatial records [5]. The spatial coverage itself is accessed through the SDP backend, a dynamic library that is loaded at runtime by the OME framework [5]. The use of the SDP interface greatly simplifies the logic needed to simulate processes that are dependent upon explicit spatial relationships, leading to models that are easier to understand and improved performance. For the example referenced in the Simple Critter Model, only the dynamics of what occurs in an individual plate are required to be defined by the model. The inflows and outflows to potential neighboring plates are defined in the model itself, with the value expressions referencing the SDP via its API, which in turn evaluates the state of flow based on the presence or absence of neighboring plates.

Presently, OME provides two data-providing plugins that implement the SDP interface. The first, CSVSPatialDataProvider, supports use of spatial data supplied as one or more .csv files [5]. Neighbor relationships can either be determined by attributes passed in as part of the `<spatial_provider>` tag in the .omec file, or by including a simple Lua [17] script that provides the necessary logic for establishing neighbor relationships [5]. The CSVSPatialDataProvider is used by the Simple Critter Model to read in a list of plates and their relative positions, evaluate each plate to find its valid neighbors, and resolve spatial queries that are entered through the SDP API. The second SDP backend, EnvSpatialProvider, is provided as part of the OMEAdapter [5], where the responsibility for resolving neighbor queries is left to the parent modeling environment, Envision. OME provides a virtual C++ interface as an API, so new backend plugins can be created and incorporated into an OME simulation by implementing the necessary interface functions, allowing for the introduction of explicit spatial data from new and/or novel data sources [5].

In the case of the Simple Critter Model, explicit spatial information comes in the form of the specific configuration of plates being used; changes in the organization of plates can have an impact on the migration of the critters through the model, and therefore must be accounted for. The plate records are stored in a simpleCSV file, with the contents organized in a similar fashion to that found in Table 1. The SDP uses the mapped information from the coverage files combined with a user-provided script to determine neighbor adjacency for plates; the actual script in the .omec file used to denote how to find adjacency is provided in Figure 4. Essentially, a plate is represented by an instance of a submodel, with its inflows and outflows connected to sources and sinks, respectively. A combination of model expression logic and queries to the SDP determine which flows transfer values to specific adjacent neighbors, if any. Since the SD model logic is completely decoupled from any of the explicit

spatial relationships, any number of plates in any configuration could be applied to the model without changing any of System Dynamics-based logic. This stands in strong contrast to the approach to representing explicit spatial data using traditional SD model components, where only Stocks, Flows, Influences, auxiliary Variables, and Source/Sinks are used: spatial relationships are encoded directly into the model [7]; any change to the spatial coverage would require modifications to the SD model itself. Some SD modeling environments provide custom model components that can be used to achieve a degree of flexibility in spatial representation, although using such components will limit the model to being run under the environment in which it was modeled. For a more in depth example walking through the process of modifying an existing SD model to utilize the SDP, and to see an example of using Simile's role arrow model component to model explicit spatial relationships, see Appendix B.

Since the Simple Critter Model relies on each plate being represented by an instance of a specified submodel, inter-plate critter flows would be directly parseable in their submodel representations by using the results views in OMESimRunner, or by observing the output from OMEEngine (either onscreen or exported CSV files). In this case the values would be captured by the various Flows and the Stock associated with a specific plate-representing submodel instance.

4. Ongoing Improvements and Future Enhancements

4.1. Further refinement to conversion tools

As previously stated, OME must be able to incorporate models originating from different sources in order to be useful. Presently, this is handled by writing conversion utilities which make use of the OMERuntime library to copy a model from some native format and export to the .omem xml format [5]. These conversion tools are then incorporated into the UniversalConverter utility, which selects the appropriate conversion tools based on a model's file extension. The most developed of conversion tools are those which target .sml and .pl files originating from Simile [5]. Development of sml-specific conversion tools is ongoing and supported features are gradually being expanded; see the document titled "Simile Compatibility" in the OME repository for more information. The tools for the other two conversion targets, .mdl files (originating from Vensim) and XMILE files, are less advanced in their development. Since XMILE is intended to be a standardized file specification with thorough documentation, OME would benefit by providing support since it would implicitly provide access to any additional model authoring tools that adopted the format as an export option. The STELLA model development software already has support for exporting XMILE files, and this format is in the process of being ratified as a standard file format for SD model exchange [4]. For this reason it is worth exploring the possibility of replacing OME's current custom model format with XMILE, and wrapping OME-specific behaviors and attributes as extensions to the format (or perhaps in a companion file of some sort). This would take considerable refactoring and adjustments to OME's source code, but the end result would be much wider access to published models and tools with fewer of the user facing workflow steps as outlined in Figure 5. Fewer steps are good for the end-user,

and provides overall for a much more elegant user experience. However, this is not the only piece of the overall OME user experience that could benefit from streamlining.

4.2. Simplification of OME Model Compilation process

Another area that would benefit from simplification and streamlining is the production of compiled binary model logic for use during the simulation. Presently, the OME framework has a tool (ModelClassBuilder) designed to generate C++ code from an OME model [5]. This code is then manually compiled in to a dynamically linked library using an external C++ compiler [5]. Some of this process could be simplified using shell scripts and/or batch files, but the reliance on an external compiler is troublesome. One possibility would be to generate Java or Python code from the model update expressions and fall back to an embedded instance of the appropriate Virtual Machine (VM) or interpreter. While such generated code would work, another set of code serialization classes would need to be written for the newly selected language, while most of the work for generating C++ code would be relegated to a lower priority or altogether discarded. Another approach would be to use an embedded C++ compiler which would provide machine code without the use of a VM; such a niche could be fulfilled by the open source LLVM Project. LLVM is a compiler infrastructure package that provides various modular libraries which can be combined to produce custom compilers and debuggers [18]. LLVM is in use on a number of platforms for various products, with the project's C/C++ compiler (Clang) and debugger (LLDB) underlying the environment used to develop software for modern Apple operating systems, such as Mac OS X and iOS [19]. OME could leverage libraries from LLDB and Clang to compile and link a library in-memory, thus negating the need for an external C++ compiler and a user-initiated external compilation step. Figure 6 illustrates the optimized workflow and the amount of user interaction that would be required. The effort to include model compilation capabilities within OME would not be too difficult, since there is already capability in the OMERuntime library to dynamically generate C++ code; some refactoring to the simulation pipeline would be necessary, and OME would need to be shipped with the required source code support files. By directly supporting XMILE and removing the necessity of external code compilation, the OME model simulation process becomes much simpler, and therefore becomes accessible to a much wider audience.

4.3. Implementation of a useful Stock-Flow diagramming interface

At present, OME does not have a Stock-Flow diagram-based visual interface, unlike many of the tools that OME is attempting to augment. While the development of such an interface was discussed early in development, it was determined that the time and resources to build such a user interface would be better spent developing the core simulation engine, and that the existing models that OME is capable of importing already come from environments that have robust visual interfaces. Still, it would be a desirable to have some sort of native interface, and the internal data structures used by OME are designed to store values pertinent to a representative visual geometry. Designing a visual interface considered robust enough to distribute would take a considerable amount of time to design, implement, and test, as it would have to capture a wide range of modeler expectations while properly mapping with the existing capabilities of OME. Adding a robust interface may add value, but at this stage ensuring the integrity of the simulation engine, implementing a novel Spatial Data Provider,

and properly defining the modular architecture all seem much more important in a modeling ecosystem where nearly all symbiotic tools implement their own approaches to building and representing Stock-Flow diagrams.

5. Conclusion

OME is designed to address two shortcomings within the ecosystem of SD modeling tools: 1) the lack of an elegant means of managing spatially explicit data, and 2) the difficulty involved with coupling largely insular tools with other modeling environments. OME is designed to address both of these shortcomings in a straightforward way.

While spatially explicit relationships can be represented using the conventional components of a Stock-Flow diagram, such configurations can be a challenge to maintain and often become inflexible as complexity increases [7]. OME provides a solution to this inflexibility by removing the necessity of including details about spatially explicit relationships from the Stock-Flow diagram altogether and instead using the Spatial Data Provider (SDP) to handle any queries regarding said relationships. Through this approach OME is capable of both reducing visual and structural complexity of a model's Stock Flow diagram and reducing the burden involved in interacting with explicit spatial data.

The challenges involved in coupling SD models with other software tools is overcome by OME through the use of modular design practices. OME itself is divided up as a series of libraries and executables that interact with one another based on the specific task at hand. The modular configuration is such that the OME simulation engine can run as part of a standalone OME executable, or as a plugin to a much larger modeling environment. If the flexibility that is already provided by the modular design is not enough, the source code for OME itself is freely available, allowing for direct modification for unique usage cases.

Ultimately, OME strives to reduce the amount of work required from a System Dynamics modeler by addressing the aforementioned shortcomings. With the OME-supplied solutions to these issues, authors can focus more on the models themselves instead of committing considerable time to satisfying implementation details. Through the novel functionality it provides, OME is positioned to beneficially contribute to the ecosystem of System Dynamics Modeling tools.

References

1. Lane DC. Diagramming conventions in system dynamics. The Journal of the Operational Research Society. 2000:241–245.
2. Simulistics Ltd. Welcome to simulistics: System dynamics and object-based modelling and simulation software 2015 <http://www.simulistics.com/>
3. Ventana Systems, Inc. Vensim software 2015 <https://vensim.com/vensim-software/>
4. isee Systems. Xmile - an open standard for system dynamics models Jul, 2013 <http://blog.iseesystems.com/news-announcements/xmile-faq>
5. Wingo PC. Master's thesis Oregon State University; Jun, 2015 Ome: A framework for running spatially explicit system dynamics models.
6. Costanza R, Voinov A. Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach New York: Springer-Verlag; 2004

7. VoinovA, , FitzC, , BoumansR, , CostanzaR. Modular ecosystem modeling. In: CostanzaR, , VoinovA, editors Landscape Simulation Modeling: A Spatially Explicit, Dynamic Approach Springer-Verlag; 2004 4376
8. Simulistics Ltd. Simile documentation and help Jul, 2008 <http://www.simulistics.com/help/index.htm>
9. Ventana Systems, Inc. Vensim help: Variable types https://www.vensim.com/documentation/ref_variable_types.htm
10. isee Systems. Stella reference: Objects and properties: Building blocks http://www.iseesystems.com/resources/help/v1-1/#08-Reference/01-ObjectsAndProperties/01-BuildingBlocks/0Overview_Building_Blocks.htm%3FTocPath%3DReference%7CObjects%2520and%2520Properties%7CBuilding%2520Blocks%7C_____0
11. Mazzoleni S, Giannino F, Colandrea M, Nicolazzo M, Di Agraria F, Massheder J. Integration of system dynamics models and geographic information systems. Modelling and Simulation. 2003; 600:304–306.
12. Environmental Systems Research Institute, Inc. Arcmap May, 2016 <http://desktop.arcgis.com/en/arcmap/>
13. Lane DC. The emergence and use of diagramming in system dynamics: A critical account. Systems Research and Behavioral Science. 2008:3–23.
14. OASIS XML Interchange Language (XMI) for System Dynamics TCXML Interchange Language for System Dynamics (XMI) Version 1.0: Candidate OASIS Standard 012015
15. Chichakly K. tech rep Information Systems SIG of the Systems Dynamics Society; 2013 Xmi: An xml interchange language for system dynamics.
16. Bolte JP, Hulse DW, Gregory SV, Smith C. Modeling biocomplexity - actors, landscapes, and alternative futures. Env Modeling and Softwar. 2007; 22(5):570–579.
17. Ierusalimsky R. Programming in Lua PUC-Rio; Brazil: 2013
18. Lattner C. The LLVM Compiler Infrastructure <http://llvm.org/>
19. Apple Inc. LLVM Compiler Overview Dec, 2012 <https://developer.apple.com/library/mac/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/>

Appendix A. Software Availability

The C++ source code and support files for the OME project are presently hosted on the EPA's Github repository, found here: <https://github.com/USEPA/OME>. Presently, the complete source code and build configurations for Windows, Linux, and macOS/OS X are freely available for download, along with instructions on how to build the files for each supported platform. The example model files for The Simple Critter Model can also be found in the repository in the Demos and Examples folder (<https://github.com/USEPA/OME/tree/master/Demos%20and%20Examples>). Additional examples will be added gradually over time.

Appendix B. Example of Explicit Spatial representation: The Simple Critter Model

OME's approach to handling explicit spatial data through the Spatial Data Provider is intended to separate general model dynamics from the explicit data source; this makes it trivial to simulate common processes across varying spatial domains, reducing some of the burden that may be placed on a model author. In order to demonstrate the utility of this approach, let's walk through a contrived example.

AppendixB.1. The setup

A new species of microscopic critter has been discovered, and a research group is attempting to determine its behavioral response to negative stimulus. A hypothesis describing a critter population's anticipated movement has been derived from observations; an experiment has been planned, and the scientists running the experiment want to build a model to predict the critter's movement based on the principles outlined in the hypothesis. These predictions will then be compared with experimental results to determine the accuracy of conclusions derived from the hypothesis.

The experimental design consists of 9 identical plates, each of which contain 1000 individual critters. The plates will be arranged in three different configurations: a line configuration, a box configuration, and an H-shaped configuration Figure B.1. For each configuration, a single plate will be electrified once a second for 20 seconds. The order of the electrification is determined by a list of numbers generated from random dice rolls, and the same order is used across all three configurations. This duration of electrocution is expected to disturb the critters enough to cause roughly 120 individuals to migrate to any neighboring plates. The rate of transfer across plates is assumed to not be saturated, and since the critters are going to head in any available direction, the movement of the 120 critters is expected to be evenly distributed across all neighboring plates. Each plate configuration is defined by a list of x and y coordinates provided within a comma-separated value (.csv) file.

AppendixB.2. Initial Approach: Using traditional System Dynamics Model components

The simplest approach that can be used to simulate the behavior of the three plate configurations is to build three separate Stock-Flow diagrams using Stocks, Flows, Variables, and Influence arrows. Figures Figure B.2, Figure B.3, and Figure B.4 show how these diagrams would look as constructed in Simile. While this approach would work in virtually any System Dynamics modeling tool and produce the necessary predictions, it is rather inflexible. Any new configuration of plates would require a new Stock-Flow diagram, and any additional plate requires any number of new Flows and Influences, further cluttering the diagram and making it less readable. If the configuration of the plates were abstracted out of the System Dynamics diagram, readability could be increased.

AppendixB.3. Abstract Out Space: Use Tool-specific model components

Fortunately, many System Dynamics authoring tools have their own unique model components that can be utilized to simulate explicit spatial relationships without encoding those specific relationships into the diagram itself. In Simile, a pair of submodels and Association Arrows can be used to determine if two plates are adjacent, and to regulate the exchange of critters across plate boundaries based on the input provided from one of the configuration .csv files (Figure B.5). While this approach is much more flexible and robust than the previous attempt, it still has some limitations. The solution from this approach is tool-specific since it requires exotic model components. This approach also produces a lot of

noise in the Stock-Flow diagram that interferes with the portions of the diagram that we are interested in. If we could completely abstract out explicit spatial factors out of the diagram completely, that would be ideal.

Figure B.1. Plate configurations: (a) line shape; (b) box shape; and (c) H-shape. Ideally, this model would be written so that it would work with different plate configurations, starting populations, and electrification orders so that it can be used to make future predictions while attempting to verify related hypotheses.

Figure B.2. Stock-Flow representing the line configuration.

Figure B.3.
Stock-Flow representing the square configuration.

Figure B.4.
Stock-Flow representing the H-shape configuration.

Figure B.5.

Simile-style Stock-Flow diagram where each plate is represented by an instance of the Plates submodel, and neighbors are determined by the neighbor test submodel.

Appendix B.4. Explicit Spatial Data as a service: OME and the SDP

Using OME and its Spatial Data Provider (SDP) interface, the Stock-Flow diagram in Figure B.5 can be modified so that explicit spatial relationships are requested by the model rather than rigidly defined in the diagram. This greatly simplifies the Stock-Flow diagram and provides the greatest flexibility as any arbitrary source of spatial information can be used to satisfy the spatial queries. Figure B.6 provides a mockup of what the Stock-Flow diagram would look like for the OME model if it had a visual component.

In order to take advantage of OME, we need to first convert the Simile model to the OME .omem format. This can be accomplished with OME's UniversalConverter utility. During the conversion process we can use directives in model component comments boxes to modify the model to fit our needs. Specifically, we can remove the neighbor test submodel altogether, as it is no longer needed; dirCount can now directly ask the SDP how many neighbors are present for a given plate; and each Flow can be modified to request information about the availability of neighbors in each of their respective directions. See Table B.1 for the list of directives provided.

Table B.1

Modifier Expression as applied to model components. Each expression in the “Modifier Statement” column contains one or more modifier expressions, each prefixed with a ‘@’ character. The “Purpose” column describes what the combined modifier statement is used for with regard to the model’s intent.

Component	Modifier Statement	Purpose
dirCount	--!!OME: @initOnly @expression:SDPGetNeighborCount(index(1)) !!--	Count all neighbors, but only do it once (on initialization).
downIn	--!!OME: @execGroup:2 @expression:if SDPHasNeighborForDirection(index(1),SDP_SOUTH) then valueFromInstance(upOut, SDPGetNeighborForDirection(index(1),SDP_SOUTH)) else 0 !!--	Link inflow from south neighbor, if present. Evaluate after all *out labelled flows are evaluated across all instances.
downOut	--!!OME: @execGroup:1 @influenceTo:upIn @expression:if (electrified!=0) && SDPHasNeighborForDirection(index(1),SDP_SOUTH) then dirTrans else 0 !!--	Creates an influence to upIn. If plate is electrified, transfers a portion to critter to southern neighbor if present. Evaluate across all instances before any *In labelled Flows.
leftIn	--!!OME: @execGroup:2 @expression:if SDPHasNeighborForDirection(index(1),SDP_WEST) then valueFromInstance(rightOut, SDPGetNeighborForDirection(index(1),SDP_WEST)) else 0 !!--	Link inflow from west neighbor, if present. Evaluate after all *out labelled flows are evaluated across all instances.
leftOut	--!!OME: @execGroup:1 @influenceTo:rightIn @expression:if (electrified!=0) && SDPHasNeighborForDirection(index(1),SDP_WEST) then dirTrans else 0 !!--	Creates an influence to rightIn. If plate is electrified, transfers a portion to critter to western neighbor if present. Evaluate across all instances before any *In labelled Flows.

Component	Modifier Statement	Purpose
neighborTest	--!!OME: @omit !!--	Removes the neighbor test submodel and its enclosed components, as well as any external linkages
Plates	--!!OME: @expectsSpatial !!--	Hint for OME that the model expects to be populated by SDP.
rightIn	--!!OME: @execGroup:2 @expression:if SDPHasNeighborForDirection(index(1),SDP_EAST) then valueFromInstance(leftOut, SDPGetNeighborForDirection(index(1),SDP_EAST)) else 0 !!--	Link inflow from east neighbor, if present. Evaluate after all *out labelled flows are evaluated across all instances.
rightOut	--!!OME: @execGroup:1 @influenceTo:leftIn @expression:if (electrified!=0) && SDPHasNeighborForDirection(index(1),SDP_EAST) then dirTrans else 0 !!--	Creates an influence to leftIn.If plate is electrified, transfers a portion to critter to eastern neighbor if present. Evaluate across all instances before any *In labelled Flows.
upIn	--!!OME: @execGroup:2 @expression:if SDPHasNeighborForDirection(index(1),SDP_NORTH) then valueFromInstance(downOut, SDPGetNeighborForDirection(index(1),SDP_NORTH)) else 0 !!--	Link inflow from north neighbor, if present. Evaluate after all *out labelled flows are evaluated across all instances.
upOut	--!!OME: @execGroup:1 @influenceTo:downIn @expression:if (electrified!=0) && SDPHasNeighborForDirection(index(1),SDP_NORTH) then dirTrans else 0 !!--	Creates an influence to downIn.If plate is electrified, transfers a portion to critter to northern neighbor if present. Evaluate across all instances before any *In labelled Flows.

Note that if we were to start designing the Simple Critter Model with OME in mind, we would basically start with the model setup provided by Figure B.6, and would still use the modifier expressions in Table B.1, minus the modifier applied to the “NeighborTest” submodel, and all the “@influenceTo:” modifiers. In this case, the Model components listed in Table B1 would contain expressions with placeholder values necessary to satisfy Simile’s error checking; placeholders would then be replaced during the conversion process.

Next we need to create a control file which has an entry that tells OME what SDP to use and how it is to behave. Since our spatial data comes in the form of a csv, we can use OME’s CSVSpatialDataProvider to represent the plate configuration being tested. To do so, we will tell the data provider what csv to load, that we are representing a grid_4 (a rectangular grid with four directions of movement), and that we want to determine solely by x-y coordinates. Since .csv files have no conventional rules to describe how entries relate to one another, we have to provide those rules ourselves. Fortunately, we can do this in a straightforward way using a simple Lua script (see the body of <neighbor_expr> tag in Figure B.7). The script takes in a pair of xy-coordinates, the first being the source plate and the second being the candidate neighbor plate. If either the x or the y coordinates are equal (but not both), and the opposite coordinates deviate by ± 1 , then the candidate is considered a neighbor. At this point the neighbor’s index is returned as well as a symbolic value that represents the direction of the neighbor. Figure B.7 has a full listing of the SDP entry on the OME control (.omec) file. A detailed discussion of Lua can be found in the book Programming in Lua [17].

```

<?xml version="1.0" encoding="UTF-8"?>
<ome_ctrl>
1) <spatial_provider library_path="CSVSpatialDataProvider" filepath="..."
    x_label="x" y_label="y" coverage_format="grid">

2)   <neighbor_expr language="lua" mode="compare_xy" grid_type="grid4">
      local center,x1,y1,neigh,x2,y2 =...
      --check to see if both cells are the same
      if x1==x2 and y2-y1==1 then
        return neigh,SDP_SOUTH
      end
      if x1==x2 and y1-y2==1 then
        return neigh,SDP_NORTH
      end
      if y1==y2 and x2-x1==1 then
        return neigh,SDP_EAST
      end
      if y1==y2 and x1-x2==1 then
        return neigh,SDP_WEST
      end
    </neighbor_expr>
  </spatial_provider>
  <coverage_mapping>
    <submodel name="Plates" instance_per_unit="1">
      <var sdp_name="x" mapto_name="x" init_dir="sdp_to_component"/>
      <var sdp_name="y" mapto_name="y" init_dir="sdp_to_component"/>
      <var sdp_name="index" mapto_name="id" init_dir="sdp_to_component"/>
    </submodel>
  </coverage_mapping>

3) <solver method="euler"/>
   <ome_model filepath="omeDemoModel.omem" start_time="0.0"
     stop_time="20.0" report_interval="1.0"
     relative_step_size="1.0"/>
</ome_ctrl>

```

Figure B.7.

A complete .omec file. Box 1) contains the mapping information for the SDP and any associated coverage files; in this case, the ellipsis would be replaced by either OMEDemo-LineCoverage.csv, OMEDemoSquareCoverage.csv, or OMEDemoHCoverage.csv for the line, square, or H-shaped coverages, respectively. Box 2) contains the tags instructing a coverage how to assemble itself, and which values to directly map. Together, boxes 1) and 2) contain the tags with the necessary information to map explicit coverage data into the System Dynamics model. Box 3) contains the information to carry out the simulation on the named model file, omeDemoModel.omem.

Once the configurations are in place, the Critters simulation can be run with ease. If the configurations and / or the number of plates change, only a single attribute will need to be changed in the SDP entry in the control file. Further down the line, if the data comes from a different source altogether, only the SDP implementation needs to be swapped; no modification to the model itself will be necessary. By abstracting explicit spatial information

out of the model and into the SDP, the implementation has become much more flexible and easier to maintain.

AppendixB.5. Sample Results

At this point, we can run the model with each of the coverages which will produce simulated results. Figure B.8 contains one set of potential results for each coverage.

Figure B.8. Resulting populations for each plate at the conclusion of the run using the time series provided by the attached table using the configurations provided in Figure B1. (a) shows the final plate populations for the line configuration, (b) shows the final plate populations for the square configuration, and (c) shows the final plate populations for the H configuration. The table “Electrified Plate for Time” shows which plate was electrified at a specific time step, with no plate being electrified for the initial condition (Time Step 0).

Component	Type	Description
OMERuntime	Library	Base library that handles all the tasks necessary to run a simulation. Such tasks include loading model data, event management (namely starting, stopping, and pausing), and resource management.
OMEDraw	Library	An auxiliary library that is used to handle tasks not directly related to the simulation (storing comments and descriptions, parsing Stock-Flow positioning information, lists of values to display, etc.).
CSV SDP (CSVSpatialDataProvider)	Library	A backend for the Spatial Data Provider which is intended to read in spatially explicit data from comma-separated value (.csv) files.
OMEEngine	Executable	Command-line tool for running an OME simulation and saving the results.
ModelClassBuilder	Executable	Command-line tool for converting an OME model's logic into C++ code suitable for compiling into a dynamic library, which can then be used to greatly reduce a simulation's run time.
UniversalConverter	Executable	Command-line tool for generating OME models from external tools (such as Vensim or Simile).
OMESimRunner	Executable	A Graphical tool for running OME simulations and displaying the results in an interactive fashion

Figure 1. OME architectural boundaries diagram and description table updated from [5]. Solid arrows represent required dependencies for each module to function properly. Dotted arrows represent optional linkages whose dependencies are defined in an OME control (.omec) file. The box <compiled model> represents the update logic of the model compiled into binary form, whereas <plugin> represents any interface between OME and an external tool; since these two entries are stand-ins, no additional details are provided in the description table.

Figure 2.
A simple Stock-Flow diagram (a), and the contents of an OME control file (b) and model definition file (c).

(a)

```

<autonomous_process
  name      = 'OMEProcess'
  path      = 'OMEAdapter.dll'
  initInfo  = 'omeDemoModel.omec'
  ...
/>

```

(b)

```

<coverage_mapping>
  <submodel name="Plates" instance_per_unit="1">
    <var sdp_name="x" mapto_name="x" init_dir="sdp_to_component"/>
    <var sdp_name="y" mapto_name="y" init_dir="sdp_to_component"/>
    <var sdp_name="index" mapto_name="id" init_dir="sdp_to_component"/>
    <var sdp_name="population" mapto_name="Pop" update_dir="component_to_sdp"/>
    ...
  </submodel>
  ...
</coverage_mapping>

```

Figure 3.

Example entry for including OMEAdapter in an Envision .envx project file (a), and the sample mappings from the Simple critter model .omec control file (b). The control file (omeDemo-Model.omec) is referred to in the “initInfo” attribute in (a), while the coverage mapping entries in (b) state that each plate will initialize its x, y, and index components from the coverage provided by Envision, and that Envision’s coverage will be updated with the population information for each plate after each time step. While the contents of these tags are complete, the ellipses (...) represent the possibility of additional attributes being present in (a) and additional tags being present in (b).


```

<neighbor_expr language="lua" mode="compare_xy" grid_type="grid4">
local center,x1,y1,neigh,x2,y2 = ...

if x1==x2 and y2-y1==1 then
    return neigh,SDP_SOUTH
end

if x1==x2 and y1-y2==1 then
    return neigh,SDP_NORTH
end

if y1==y2 and x2-x1==1 then
    return neigh,SDP_EAST
end

if y1==y2 and x1-x2==1 then
    return neigh,SDP_WEST
end

</neighbor_expr>

```

Figure 4.

Neighbor Expression used with the SDP for the Simple Critter Model for use with the CSVSpatialDataProvider backend; the neighbor expression is used to describe how to interpret the CSV containing spatial information since there is no standard for organization spatial data within the CSV file format. The attributes in the neighbor_expr tag specify Lua as the script language, that comparisons should be against values from the x and y value columns, and that the coverage type is a square grid where the four edge neighbors are considered. The embedded Lua script (black text) tests to see if two cells are adjacent, and returns the neighbor index and the direction to the neighbor if adjacency is found.

Figure 5.

Present division of user interaction and autonomous work for an OME workflow starting with a Model file originating from an external tool. Rounded boxes represent files and/or products generated during a workflow, while diamonds are operations that take place. All shapes on the left are responsibility of the user, while all shapes in the box on the right are the responsibility of OME itself. The user's responsibility largely revolves around the management of intermediate files that are necessary to produce simulation results from a model definition.

Figure 6.

Division of user interaction and autonomous work for an OME workflow after proposed changes are implemented: replacing the OME model declaration with the standard XMILE format, and the incorporation of automated model logic compiling using an embedded C++ compiler. Rounded boxes represent files and/or products generated during a workflow, while diamonds are autonomous operations that take place. Only the shapes on the left side of the diagram require direct user interaction: A XMILE model declaration must be provided, and the Simulation Results must be interpreted by the user. Most items designated as the user's responsibility in Figure 5 can be moved into OME's space once the necessary changes are made, simplifying the user's burden.

Table 1

Example records for Simple Critter plates. The first column is the unique id for a given plate, while the second and third columns are the x- and y-coordinates, respectively. This information is used to lay out plates in a specific spatial pattern

index	x	y
1	0	0
2	1	0
3	2	0
4	0	1
5	1	1
6	2	1
7	0	2
8	1	2
9	2	2