Complexities of Simulating a Hybrid Agent-Landscape Model Using Multi-Formalism Composability

Gary R. Mayer
Gary.Mayer@asu.edu

Hessam S. Sarjoughian
Sarjoughian@asu.edu

Arizona Center for Integrative Modeling & Simulation
Computer Science & Engineering Department
School of Computing and Informatics
Arizona State University, Tempe, Arizona

Keywords: agents, cellular automata, multi-modeling, multi-formalism, poly-formalism.

Abstract

Hybrid agent-landscape models are used as an environment in which to study humans, the environment, and their dynamics. To provide flexibility in model design, expressiveness, and modification, the environment models and human agent models should be developed independently. While retaining each model's individuality, the models can be composed to create a model of a complex, hybrid agent-landscape system. This should allow for a much more in-depth analysis of each model independently, as well as a study of their interactions. To create such a modeling environment requires a look beyond a simple interface between two models. It may require that the models' formalisms be composed, their execution be synchronized, their architectures be integrated, and a common visualization be created to provide a whole-system data view during simulation. This paper discusses the complexities of such an undertaking.

1 INTRODUCTION

Hybrid agent-landscape models are typically used to better understand the effects of human beings interacting with their environment. The term "hybrid" is used to imply that *both* the human and environmental sub-system models are developed to some higher resolution that enables a clear delineation between the two to be drawn, if not developed to the point that they might be separated and executed independently. However, the incorporation of a large amount of detail in a model often results in large numbers of interacting pieces – modeled systems whose interacting pieces bring about different kinds of complexities. Moreover, a new dimension of complexity is introduced when the human and environmental sub-systems are combined. Their interaction is bi-directional as the human and environment models act as both producer and consumer in the hybrid model. Yet, by maintaining a clear delineation of each sub-system, researchers are able to choose a formalism that best represents the sub-system to be modeled, complex or not, thus enabling a more intuitive mapping from domain knowledge to formalism (e.g., using an agent-based model to represent the humans and a cellular automaton for the environment). Furthermore, by maintaining independent models researchers are able to study the human population and the environment separately, as well as examine the interactions between the two methodically.

It is the intent of this paper to discuss an approach that facilitates these studies and presents some of the challenges associated with this approach. The underlying concept is to compose modeling formalisms instead of integrating models' inputs and outputs via interoperability concepts. The Mediterranean Landscape Dynamics (MEDLAND) project [1] is an on-going international and multi-disciplinary effort. One of its goals is to develop a laboratory in which to study humans, the environment, and their dynamics. In MEDLAND, humans are represented by agents and the environment is represented by a landscape model. The two models will be composed using a third model, the interaction model (see Figure 1). It is from the MEDLAND domain that examples are provided and previous research summaries are drawn.

Figure 1 shows a discrete-event, rules-based agent model composed with a discrete-time, cellular automaton landscape model. Note that there is no direct communication between the two composed models. Their communication is managed by an interaction model which handles the data transformation and control. This interaction model is complete with its own formalism and realization separate from the two composed models.

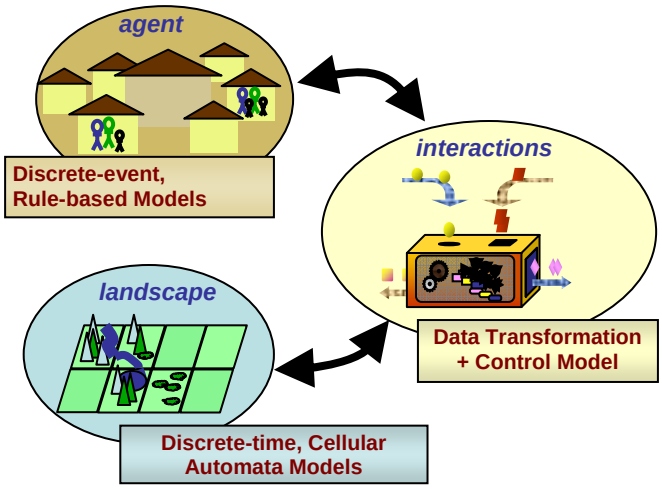

Figure 1. Proposed Agent-Landscape Model Architecture

2 BACKGROUND

2.1 Composability Problem

Using disparate modeling formalisms to describe different sub-systems has important benefits ranging from acquiring requirements to simulation experiments (see [2] for details). To achieve model composability, the main challenge centers on having appropriate concepts and methods to compose different model types such that both the disparate models and their interactions have well defined syntax and semantics. The composed models must be correct and valid – i.e., model specifications must be consistent as (i) correctness is ensured according to the domain-neutral modeling formalisms and (ii) validation is ensured according to the domain-specific model descriptions [3].

To achieve model correctness and enable model validation, a variety of issues must be considered. The authors group these to formalism and realization aspects (see Figure 2). The formalism focuses on model specification and execution – i.e., the former is for mathematical descriptions of the system and the latter is for the machinery that can execute model descriptions. The formalism modeling and execution layers are not specific to any one model instantiation; they're generalized for a class of models (e.g., discrete-time).

Important considerations for composability are the implications of what it means to inject data and control from an external source that may not have the same approach to model specification and execution. For instance, one model may have an innate concept of time, while the other does not. Consider, also, what it means for a discrete-event model to inject data into a discrete-time model not at predefined time steps. As a final example, take one model that uses a state-based approach. The modeler must ensure that an external input does not arbitrarily modify the state of that model. All state changes must be in accordance with the rules of that model's formalism else correctness and validity of that model is suspect.

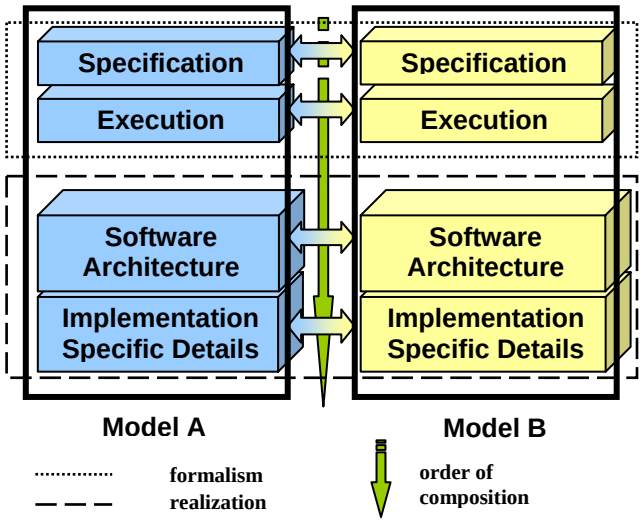

Figure 2. Model Layer Composition

The realization aspect deals with software architecture, design, and implementation. The software architecture and design are conceptual and detailed software specifications (e.g., described in the Unified Modeling Language (UML)) that can be forward engineered to specific programming language constructs (e.g., Java, C, Lisp). Examples of design considerations can be as simple as converting an integer from one model into a double for another. For distributed model systems, it includes such things as quality of service and handling synchronous versus asynchronous input and output.

The last layer under realization is titled "Implementation Specific Details". This refers to those elements of a model implementation that are not architecture related; they are more closely tied with the data. Specifically, how things like the scale and resolution of the model interact with other models. For example, an environmental model may employ millions of data elements; each at a 100 meter scale, while an agent model uses only a couple of hundred agents working at a 10 meter resolution. The modeler must consider the significance to an agent's movement possibly existing entirely within one cell of the environment model.

2.2 Multi-Modeling Approaches

In their books, Fishwick [4] and Zeigler and others [5], describe multi-modeling as the creation of a model composed of smaller models. Each sub-system model captures only a part of the whole system. The composition

of these sub-system models enables the modeler to create a much more complex representation of the whole system. For example, the Joint-MEASURE simulation environment integrates DEVS-C++ and GRASS models to simulate movement of vehicles on geo-referenced terrain [6, 7].

A taxonomy for multi-modeling using the same or different types of formalisms has been proposed [3]. This nomenclature offers four approaches to multi-modeling.

A *mono-formalism* refers to one in which there is a hierarchical composition of models into/from parts described within one syntax and semantics. For instance, a system model composed of all discrete-time models.

Super-formalism is a single formalism that supports describing two or more different types of models. It requires that the models be of the same family (e.g. system specifications). Note that the super-formalism approach forces a uniform execution approach and syntax on both models. For example, the Discrete Event & Differential Equation System Specification (DEV&DESS) [5, 8] can describe both a continuous model and a discrete-time model.

A *meta-formalism* describes mapping two disparate formalisms to a third, common formalism. It does not have the same-family model restrictions that are levied upon the super-formalism. However, expressiveness of the model formalisms must often be restricted according to the meta-formalism to ensure proper, multiple mappings. High-Level Architecture (HLA) [9] and Repast [10, 11] are examples of this.

Poly-formalism is the approach in which disparate formalisms interact via a third formalism while retaining their original formalisms. An example of this approach is the use of a Knowledge Interchange Broker (KIB) to compose Discrete Event System Specification (DEVS) and Linear Programming (LP) models [12].

The choice of which multi-modeling approach to use is situation dependant. It will likely be the modeler's opinion, motivated by the domain and requirements levied upon the system model (e.g., ability to described the model in one formalism or withstand the loss of expressiveness and/or domain specific details during mapping).

3 RELATED WORKS

Building a system model from multiple sub-system models is not a new concept. As such, there are a number of existing toolkits that perform some combination of agent-landscape modeling [13]. These toolkits have been examined during the course of this research and the most applicable are discussed here.

3.1 Swarm and Repast

Two well known toolkits are Swarm [14] and the Recursive Porous Agent Simulation Toolkit (Repast) [10, 11]. Both are discrete-event, multi-agent modeling systems using object-oriented software programming concepts and design. While Repast offers more features, including support for the Geographic Information System (GIS), both have some visualization capability. Both systems use a meta-formalism approach that requires the modeler to convert models into an object-oriented construct that complies with their simulator's specifications. As there is no formalism for either system, proof of correctness of the entire model is left up to the modeler. Further, neither system provides a formal method for conversion of any particular type of modeling formalism into their system. The use of the toolkit simulator, which uses a simplified definition of discrete event that allows multiple state transitions to result from a single event [15], is prohibitive to implementing a formalism within one of the toolkit environments. Without the ability to properly implement a formalism within these environments, correctness is a concern and composition of formalisms and models described within them are difficult to validate. Thus, the poly-formalism composition approach is not suitable in these environments.

3.2 Ptolemy

Ptolemy II [16] is a computational framework for embedded systems that focuses on concurrent systems. Ptolemy composes model domains through interactions and domain polymorphism. It does this by structuring the model domains as components and treating each as an actor under the control of directors using interface automata. It is super-formalism modeling with a strong software engineering emphasis [17]. While Ptolemy addresses multi-formalism modeling, it does so while focusing on interactions between embedded systems (each a unique domain). In the case of this project, the focus is on a single domain in which pieces of the domain are best described using a different modeling formalism. Furthermore, the discrete time domain within Ptolemy II is still experimental. The current model has strict requirements such as static scheduling and the requirement to know what will execute on the ports before the simulation begins. A restriction that is impossible to meet if the two composed models are to truly remain independent.

3.3 KIB

There have also been successful attempts at poly-formalism composability. In three other research projects, the third model that facilitates the composition is referred to as a Knowledge Interchange Broker (KIB) [18]. The KIB has been used to compose DEVS with Linear Programming in semiconductor supply/demand networks [12]; to compose DEVS with the Reactive Action Planner (RAP), an agent-based planner [19]; and to compose DEVS with Model Predictive Control (MPC) in semiconductor supply-chain manufacturing [20]. These three projects demonstrate two things. First, it is possible to compose models using the poly-formalism modeling approach. Second, the fact that three different projects exist and each composes DEVS with

a distinctly different formalism implies that the KIB is not generic to all formalisms. A KIB is a unique composition between two distinct formalisms. The research being done here is unique from the previous works in that it composes a discrete-event agent model with continuous processes represented by cellular automata.

3.4 MEDLAND

3.4.1 Agent-Landscape Model: Phase I

The initial agent-landscape model developed for the MEDLAND project uses a super-formalism approach created in DEVS. In that version, the agent is a discrete-event model and the landscape is a Cellular-DEVS, discrete-time model [5]. This approach was taken to enable a study of the agent model concept while reducing the complexity associated with interfacing models in different formalisms.

The model events are cyclic with each cycle representing a calendar year. The agents in this model represent households. Each household has a population that it uses to derive a need for food to survive and a labor force with which to manage land. The basic management actions are "cultivate" land, "fallow" land, and "release" land. The landscape cells each have their own soil value indicative of soil quality that ranges from 0 to a maximum of 5. When a landscape cell is cultivated, its soil quality reduces by one each cycle (to a minimum of 0). Each cycle that a landscape cell is fallowed, its soil value increases by one, up to the maximum value for that cell.

Each cycle, the agent assesses its current food requirement and compares it to an expected yield (derived from last cycle's yield and current population) to create a management plan. Conflict resolution is handled on a first-come, first-served basis and plans are revised as necessary until all food requirements are met or no additional land exists for cultivation. Any excess land held beyond what is need for cultivation is held in fallow.

The agents are given two goals. The first is simply survival. With only this goal active, the agents were able to reach a steady-state population quickly and demonstrated that cultivated and fallowed lands were swapped each cycle. The second goal, growth, allowed the agents to use large excesses of fallowed land and cultivate it. Since population growth is tied to the difference between yield and food need, this created large growth spurts in agent population. This growth continued until the agent population reached simulation boundaries [21].

While this approach worked, it also became obvious that the DEVSJAVA environment would be inappropriate for the landscape as the number of data elements increased. While efficiencies could be imparted to the Cellular-DEVS landscape model, it would be unlikely that the model could be made as robust as a continuous or discrete-time model devised using the Geographic Resources Analysis Support System (GRASS) [22, 23]. Cellular-DEVS does not scale as well as a geographical database management system such as GRASS. Furthermore, while Cellular-DEVS offers some visualization tools, the GRASS visualization tools offer a richer set of predefined features.

3.4.2 GRASS

GRASS is an implementation of GIS [23] that manages georeferenced information. As a geographical database management system, it is specifically tailored to efficiently examine and modify large geographical data sets. GRASS data (points, lines, polygons, or pixels) is stored in files referred to as *mapsets*. Each mapset may be either a *vector* or *raster* data model. Vector data models are entity models in which the data model represents a specific entity and the topology (relationship) between the data is either implicit or explicit depending upon what is represented. For example, a polygon stored as a vector would have details about each point in the polygon and the lines connecting those points, thus providing an explicit relationship between the data. The raster data model is a square, regular tessellation of continuous space. The topology in this discretization is implicit [24].

GIS implementations are capable of logical and numerical data analysis methods. These methods are not commutative (sequence matters). Thus, more complex command sets are stored in structured command files (*scripts*) and are referred to as 'models'. The scripts specify the order of function execution and to which data the functions apply [24]. These models have no predetermined specification for behavior, structure, or execution. These are left to the modeler's discretion when devising the scripts.

3.4.3 Agent-Landscape Model: Phase II

The next version of the model progressed towards a poly-formalism approach. The behaviors from Phase I were maintained. However, instead of using Cellular-DEVS for the landscape model, a GRASS landscape model was created and an interface was implemented within a component of the agent model. When agents interacted with the landscape, they sent DEVS message objects to the interface component. This component converted the message into GRASS scripts. The scripts were then executed and the resultant output was captured and returned to the requesting agent. Further, the DEVS simulator was used to execute the GRASS landscape model by implementing landscape updates as internal events within the interface component. This approach allowed DEVS to provide a schema to the execution of the landscape models in GRASS.

It is purposefully stated that this is a *progression towards* a poly-formalism approach rather than the approach itself. The reason for this is that there is a direct interface between the agent and landscape models. Also, there is currently no third model supporting this interaction. Furthermore, the interface is solely at the implementation-

level; the formalism-level is not yet realized. This phase provided a better understanding of the difficulties of exchanging data and control between the two systems while examining approaches for describing GRASS models using more formal methods (e.g., cellular automata).

Simulation experiments were conducted on the second agent-landscape model to ensure that all of the functionality enabled in Phase I still worked through the interface. A simulation visualization was created using GRASS visualization tools to show the current status of key data. The GRASS data visualization had three panels. The first showed the current land use – cultivated, fallowed, or wild. The next panel displayed soil quality. Colors changed to represent soil values, whose dynamics are related to cultivation. The last panel displayed the agent that is currently managing each landscape cell. Each agent was assigned its own color.

4 INTERACTION MODEL

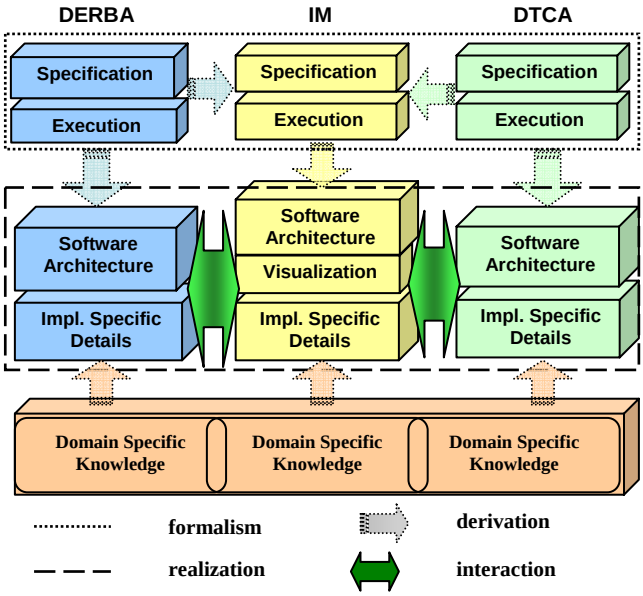

Figure 3. Proposed Hybrid Model Architecture Using a Poly-Formalism Approach

The on-going effort involves research and development of a discrete-event *interaction model* (IM) that composes a DEVS discrete-event, rule-based agent (DERBA) model with a GRASS discrete-time cellular automata (DTCA) landscape model (see Figure 3). This version completely segregates the agent and landscape models. Figure 3 shows that the formalisms of the two composed models (DERBA and DTCA) will be used to derive the formalism of the IM. Next, the realization of all three models is derived from their formalisms. It is within the realization of each model that the interactions between them occur through data mapping, aggregation/disaggregation, and control passing. Note that the IM has an additional component, a visualization layer, attached to its software architecture layer.

Visualization is an important tool for supporting simulation experimentation. The evaluation of executing models, particularly for complex large-scale domains, is invaluable for researching hybrid agent/landscape dynamics. The visualization layer is attached to the software architecture because appropriate ways to probe the composed models, while still maintaining their independence, must be considered in order to retrieve data dynamically. Despite the connection to the software architecture, the visualization layer is unlikely to be derived from the formalism.

The remainder of this section discusses the individual layers of the IM (as shown in Figure 3) and the challenges associated with the design and implementation of that layer.

4.1 Formalism

4.1.1 Specification Interaction

The interaction model must respond to input from both composed models and, in the case of the discrete-event agent model, may not know the exact timing of such events. Therefore, it makes sense to consider a discrete-event modeling formalism for the IM. However, it should be noted that the discrete-event specification for the agent is rich enough to also specify discrete-time agents and the discrete-time cellular automata may also represent a discretized continuous model. Thus, when designing the IM, it should be kept in mind that by using a discrete-event interaction model to compose any of these, a problem arises.

The poly-formalism modeling approach moves the details of the domain and formalism of each model and; therefore, the ability to interact as well, into the interaction model. This removes any domain-specific knowledge of the composed models from each other. However, it implies that the discrete-event interaction model will then inject any data into either model as an event. Since the agent and landscape models may have different time deltas, the time at which an event is injected may not align with a regularly scheduled discrete-time event. This poses a problem if any of the functions within a discrete-time model are time-delta dependant. For example, a landscape soil erosion model assumes that its values are updated once every 10 cycles. The value of 10 is explicitly used as a time delta to revise data values each time the function is run. The agent model updates every 1 cycle and, somewhere between the soil erosion model's update, the agent makes a modification that impacts that soil erosion model and requires that soil values be updated immediately. Running the soil erosion model would erroneously cause the model to update its time by 10 and cause it to be out of synch with the rest of the modeled system.

There are three approaches to managing this situation. The first requires that all composed models meet the discrete-event specification. However, this creates a problem for continuous functions within the models. By representing a continuous function within a discrete-event model, the modeler must handle discontinuity within the continuous function. This is a problem whose solution is still being studied using approximation mappings of continuous functions to discrete-events (quantizations) [5]. A second alternative is to allow the interaction model to inject the event during a model's next, regularly scheduled event time. The problem with this approach is that one model may execute much more frequently than the other and is likely to be dependant upon the data in the other model being current. This leads to further complications as the modeler must not only specify how concurrent actions injected into a model at the same time are managed, but how actions injected across multiple time frames between the model's update are controlled. The third approach levies a caveat on all discrete-time models that interface with the interaction model. The caveat states that any model in which an external model may have an impact must not contain a function that explicitly anticipates the time delta between function executions. This last approach allows the most flexibility throughout the system with the least complexity.

4.1.2 Execution Control

DEVS decouples models from their execution. There is an explicit model specification and a simulator specification. This enables a DEVS model to be run on different DEVS simulators, all of which have an innate sense of time. A GRASS model, on the other hand, is closely tied to its execution through the scripts and functions. Timing is provided by manually injecting the time as a variable or through external programs. This raises the question of how a modeler composes a model with a simulator and a sense of time with one that has no formal execution schema or timing. A solution under evaluation is to use a DEVS model to provide the GRASS model with timing. The time delta for each landscape model can be provided to the DEVS model upon initialization and the DEVS model could act simply as a clock that calls an execution script for each GRASS model.

The execution of the models as a system can be handled in two ways. The first is using a centralized control scheme. With this approach, the system has a single control scheme that exercises all three models. The second approach is a decentralized control scheme. This means that the interaction model simulator would send control messages and data to individual model simulator/executor, which would then exercise their respective models. Given that the agent and interaction models are both in DEVS[1] and the landscape model requires a simulator with an innate sense of time to automate it, the centralized control scheme seems the logical choice with minimal additional overhead.

4.2 Realization

4.2.1 Software Architecture

The software architecture must account for two main obstacles – disparate software languages and constructs, and hardware resource needs. The DEVS models are being created in DEVSJAVA, a Java language implementation of DEVS in which all models are object-oriented constructs. GRASS modules are written in C. Scripts (and functions) may be written in any scripting language such as Bash or Python depending on the system functionality that the modeler requires (e.g., file management, use of regular expressions, etc.). To run a GRASS script, a DEVSJAVA component uses the Java `Runtime.exec()` command to execute it.

Each GRASS module is independent and, therefore, has its own interface but the modules do not continuously run. They accept input, return output, and terminate. The output from the modules is only provided to the standard output stream (and, sometimes, standard error stream). Thus, to get return data, a program must capture and parse the data from the standard output buffer and then insert that data into a DEVS message object on the appropriate port. This approach is complicated by the fact that GRASS, being an open source project initially developed during the early 1970's during the time of command-line interfaces, has output that is typically preformatted for ASCII viewing and each module outputs a different format. The GRASS community is working on standardizing such variations.

The second issue that the software architecture must prepare for is hardware resources. This is because it is unknown how the two models will run together on a single machine, even if multi-processor enabled. Further, as the number of landscape cells grows and the number of agents grows, run-time memory may become a limitation. So, the architecture needs to provide an efficient approach to run both models and prepare for a possible distributed architecture. In this case, each model, the agent and the landscape, will reside on a separate computer. It must then be decided where the interaction model will reside.

4.2.2 Visualization

Visualization also plays a role in the IM software architecture. The intent is to provide a unified, synchronized data visualization with key data elements from both models displayed in a comprehensive manner. It can not be assumed that the only data that the researchers will wish to see are those that are being passed between the two composed models. Therefore, some method by which the interaction model can dynamically retrieve data during simulations must be devised. A Model-View-Controller (MVC) design pattern is being considered as the foundation for the

[1] And, therefore, decoupled from any specific simulator.

visualization architecture. This pattern separates the (data) model from the visualization component and the controller which manages both. The data model in this case is the data resulting from formatting, aggregation, and/or mapping of composed model data. The use of this pattern will allow the implementation of a controller that compliments the hybrid model framework while enabling flexibility for visualization tools that support the data under study.

4.2.3 Implementation Specific Details

The implementation specific layer focuses mainly on scalability from an execution (performance) perspective. To reiterate the example given above, an environmental model may employ millions of data elements; each at a 100 meter scale, while an agent model uses only a couple of hundred agents working at a 10 meter resolution. To be resolved is the relation between the agent and the landscape. Is it a 1-to-1 relationship where an agent can individually impart a unique action on each landscape cell or 1-to-many, where an agent's actions are applied to a group of landscape cells? How does the difference in number of elements within each model effect things like wall clock time to execute the model through a specific cycle? As posed above, what does it mean for an agent to move $1/5^{th}$ of the way through a landscape cell?

Given that each of the composed models has no domain knowledge of the other, the resolution for all of these questions must be handled in the interaction model. The resolution issue may be handled using a data mapping that is configurable at initialization. An approach to the scale issue is to allow the interaction model to maintain its own map such that it keeps track of agent locations using a finer granularity than the landscape model. The timing issue is managed through synchronous/asynchronous event handling and defining in what order the models are executed. This too may be a configurable parameter at initialization.

4.2.4 Usability

There is one additional problem that arises as a result of the research domain and the models' intended users. The MEDLAND project is targeting social scientists with little or no formal programming skills. By using DEVSJAVA, we are introducing an unfamiliar programming language with unfamiliar, object-oriented constructs as compared with scripting languages. Since this system is meant to become an operational laboratory environment, it is necessary to minimize the difficulty of revising agents in order to maintain the flexibility to research more broad topics. The challenge truly comes in not only providing this flexibility, but ensuring the model behaviors are not changed to the point that the model itself is no longer correct.

The agent model is currently under revision to incorporate more details than were originally provided under the top-down models created for the first and second phases. This version is building the agent from the bottom up based upon ethnographic data. Once examination of the data is complete, the group will make decisions on where to abstract the data in the development of the new model. This redesign provides an opportunity to also revise the agent model's structure.

The current agents in the agent model are single components. Rules, needs, capabilities, etc. are all contained within a single model. Thus, the only capability that can be provided to the researchers while being certain that correctness is not compromised is the ability to modify parameters. One approach being considered is to reduce the agent model to component pieces, each with minimal distinct behaviors. These pieces can then be configured at simulation initialization. Each piece would then maintain its specific, correct behavior with modifiable parameters. DEVS component ports could be keyed such that specific outputs could only be coupled to specific input ports, etc. to improve usability. The overall agent behavior would be modified based upon which components were coupled.

5 CONCLUSIONS

Composing two disparate modeling types is not an easy task. The modeler must first decide upon a multi-modeling approach, taking into consideration the system requirements and domain. Next, the impacts to the specification and execution of the model formalism should be considered. Within the realization of the model, the modeler should consider the software architecture, visualization, and implementation specific layers that come into play as well. Each of these has its own constraints.

Many research issues can be studied using each of the multi-formalism approaches discussed in Section 2.2, above. It will likely be a mixture of the modeler's preference and the requirements levied on the model that determine which approach to use. However, if it is important to the modeler that each sub-system retains its formalism expressiveness to provide the best description of the sub-system model and be loosely coupled with its composed model to ensure the flexibility to make changes with minimal impact to the other model, then the poly-formalism approach is suitable for achieving correctness of the modeled system. In essence, the poly-formalism modeling approach affords the modeler two levels of generality. First, at the formalism level, creating an IM allows any system containing the same class of models (DERBA and DTCA) to be composed. Note that realization details may dictate some IM changes. Second, at the realization level, the IM can compose any DEVSJAVA DERBA model with any GRASS DTCA model. At this level, only the implementation specific details within the IM may have to be adjusted.

While there are many issues presented within this paper, none are insurmountable as described above. The information provided for possible solutions reflects our

research into this problem as viewed from agent-landscape modeling and simulation. The reader may find different solutions based upon their own requirements and domain specifics.

Acknowledgements

This research is supported by National Science Foundation Grant #BCS-0140269. We thank the MEDLAND team members including Eowyn Allen, Ramón Arrowsmith, Steven Falconer, Patricia Fall, Helena Mitasova and, in particular, Michael Barton and Isaac Ullah for their support with GRASS.

References

[1] MEDLAND. 2005. "Landuse and Landscape Socioecology in the Mediterranean Basin: A Natural Laboratory for the Study of the Long-Term Interaction of Human and Natural Systems". http://www.asu.edu/clas/shesc/projects/medland/ (accessed December 2006).

[2] Davis, P.K. and R.H. Anderson. 2004. "Improving the Composability of DoD Models and Simulations". *The Journal of Defense Modeling and Simulation*. 1(5), pp. 5 – 17.

[3] Sarjoughian, H.S. 2006. "Model Composability". In *Proceedings of the 2006 Winter Simulation Conference*, pp. 149-158, Monterey, CA, USA.

[4] Fishwick, P.A. 1995. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice-Hall, Inc. Englewood Cliffs, NJ.

[5] Zeigler, B.P., H. Praehofer, and T.G. Kim. 2000. *Theory of Modeling and Simulation*: *Integrated Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed. Academic Press. San Diego, CA.

[6] Hall, S. 1998. Personal Communication.

[7] Hall, S. 2005. "Learning in a Complex Adaptive System for ISR Resource Management". Spring Simulation Conference.

[8] Zeigler, B.P. 2006. "DEVS&DESS in DEVS". DEVS Integrative Modeling & Simulation Symposium. Huntsville, Alabama.

[9] IEEE. 2000. *HLA Framework and Rules*. IEEE 1516-2000.

[10] North, M., T. Howe, N. Collier, and J. Vos. 2005. "The Repast Symphony Development Environment". In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*. Chicago, IL, USA.

[11] North, M., T. Howe, N. Collier, and J. Vos. 2005. "The Repast Symphony Runtime System". In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*. Chicago, IL, USA.

[12] Godding, G., H.S. Sarjoughian, and K. Kempf. 2004. "Multi-Formalism Modeling Approach for Semiconductor Supply/Demand Networks". In *Proceedings of Winter Simulation Conference*, pp. 232-239, Washington, D.C., USA.

[13] Parker, D.C., S.M. Manson, M.A. Janssen, M.J. Hoffmann, and P. Deadman. 2003. "Multi-Agent Systems for the Simulation of Land-Use and Land-Cover Change: A Review". *Annals of the Association of American Geographers*. 93(2), 314 – 337.

[14] Swarm. 1996. *Swarm Simulation System*. Swarm Development Group. Available from http://www.santafe.edu/projects/swarm/ (accessed December 2006).

[15] Minson, R. and G.K. Theodoropoulos. 2000. Distributing RePast agent-based simulations with HLA. *Concurrency and Computation: Practice and Experience*. John Wiley & Sons, Ltd., pp. 1 – 22.

[16] Ptolemy Project. 2006. *Ptolemy II*. Available from http://ptolemy.berkeley.edu/ptolemyII/main.htm (accessed December 2006)

[17] de Alfaro, L. and T. Henzinger. 2001. "Interface automata". In *Proceedings of the 8th European Software Engineering Conference*. Vienna, Austria.

[18] Sarjoughian, H.S. and J. Plummer. 2002. *Design and Implementation of a Bridge between RAP and DEVS*. Internal Report, Computer Science and Engineering, Arizona State University. pp. 1 – 26.

[19] Sarjoughian, H.S. and D. Huang. 2005. A Multi-Formalism Modeling Composability Framework: Agent and Discrete-Event Models. In *The 9th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pp. 249 – 256. Montreal, Canada.

[20] Sarjoughian, H.S., et al. 2005. Hybrid Discrete Event Simulation with Model Predictive Control for Semiconductor Supply-Chain Manufacturing. In *Proceedings of the 2005 Winter Simulation Conference*, pp. 255-266, Orlando, FL, USA.

[21] Mayer, G.R., H.S. Sarjoughian, E.K. Allen, S. Falconer, and M. Barton. 2006. Simulation Modeling for Human Community and Agricultural Landuse. In *Proceedings of the 2006 Spring Simulation Conference*, pp. 65-72, SCS. Huntsville, AL, USA.

[22] GRASS. 2004. *Geographic Resources Analysis Support System*. http://grass.itc.it/ (accessed December 2006).

[23] Neteler, M. and H. Mitasova. 2004. *Open Source GIS*: *A GRASS GIS Approach*, 2nd ed. Springer Science + Business Media, Inc. New York, NY.

[24] Burrough, P.A. and R.A. McDonnell. 1998. *Principles of Geographic Information Systems*. Oxford University Press Inc. New York, NY.