Composition of Geographic-Based Component Simulation Models

by

William Boyd

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2018 by the
Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Ross Maciejewski
Mohamed Sarwat

ARIZONA STATE UNIVERSITY

December 2019

ABSTRACT

Component simulation models, such as agent-based models, may depend on spatial data associated with geographic locations. Composition of such models can be achieved using a Geographic Knowledge Interchange Broker (GeoKIB) enabled with spatial-temporal data transformation functions, each of which is responsible for a set of interactions between two independent models. The use of autonomous interaction models allows model composition without alteration of the composed component models. An interaction model must handle differences in the spatial resolutions between models, in addition to differences in their temporal input/output data types and resolutions.

A generalized GeoKIB was designed that regulates unidirectional spatially-based interactions between composed models. Different input and output data types are used for the interaction model, depending on whether data transfer should be passive or active. Synchronization of time-tagged input/output values is made possible with the use of dependency on a discrete simulation clock. An algorithm supporting spatial conversion is developed to transform any two-dimensional geographic data map between different region specifications. Maps belonging to the composed models can have different regions, map cell sizes, or boundaries. The GeoKIB can be extended based on the model specifications to be composed and the target application domain.

Two separate, simplistic models were created to demonstrate model composition via the GeoKIB. An interaction model was created for each of the two directions the composed models interact. This exemplar is developed to demonstrate composition and simulation of geographic-based component models.

ACKNOWLEDGEMENTS

The creation of this thesis was only possible thanks to the help of many people. I owe many thanks to my professor and advisor, Dr. Hessam Sarjoughian, for teaching me theories and skills, introducing me to modeling and simulation research and communities, and guiding my overall computer science education. His knowledge, patience, expectations, and wisdom have been a necessary source of encouragement throughout my education at Arizona State University. I thank Dr. Ross Maciejewski and Dr. Mohamed Sarwat for serving on my thesis committee.

The Medland team of the School of Human Evolution and Social Change at Arizona State University supplied an immeasurable amount of insight and knowledge into geographic historical simulations. In particular, I owe thanks to Dr. Michael Barton for showing me many details of using the GRASS software, Dr. Isaac Ulluh and Dr. Sean Bergin for helping me assemble and run multiple-software simulations, and Dr. Miguel Acevedo for working with me to integrate his vegetation simulation into other software simulations.

I owe thanks to my academic advisors, Christina Sebring and Arzuhan Kavak, for guiding me throughout the registration and course scheduling processes.

My colleagues in the Arizona Center for Integrative Modeling and Simulation supplied ample knowledge and friendship during my experience as a university graduate student. Soroosh Gholami, Abdurrahmin Alshareef, Chao Zhang, Masudul Quraishi, Mostafa Fard, Zhi Zhu, Moon Gi Seok, Xuanli Lin, Savitha Sundaramoorthi, and Gregg Scherer are all model scholars who have provided examples of how university research should be conducted.

I am grateful to the United States National Science Foundation for funding the research (NSF grant #DEB-1313727). This grant made it possible for me to continue my education and follow my dreams.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Simulations of real-world systems require abstractions of those systems. In the abstraction process, some system attributes are omitted, but others are delegated to external simulations and data. To create models of complex systems, it is often best to compartmentalize different aspects of systems as separate, autonomous models. These component models are each best specified with a separate set of specifications, operating with different rules and scales. Each model could be simulated independently using a simulation engine.

Each model, on which simulations are based, is defined to have a particular input and output specification. The input-output format includes more than programming language, data type, and value limitations. The frequencies, measurement units, and meanings of the variables play a significant role in a model. Some models have other specifications for I/O, such as spatial resolution or null value limitations. These unique specifications result in difficulty connecting different models, even for models that represent related systems. If one model requires vegetation density, in kilograms per square meter, as input, and another model outputs vegetation volume in cubic meters, the models could not be coupled directly.

Reconciliation of variable specifications must be done whenever the expected input of a simulation is not provided, with the same format, by the output of another simulation. While changing the simulations' I/O requirements may work in these situations, such changes are undesirable. Each simulation should function according to its own requirements, and should be tested based on the same requirements, and changing an input or output violates this. Forcing adherence to a common protocol places restrictions on component models that are inappropriate to their different scopes of operation.

Separate components can be used to provide inputs to a simulation given outputs from a different simulation. Such interaction components, called Knowledge Interchange Brokers or KIBs, are responsible for making conversions of data types, quantity, resolution, and frequency, between two models [1]. The KIB has a syntax and semantics that are separate from those of the simulations to which it connects. In this project, a design for a geographic KIB is developed that receives data from different agent-based and cellular automata simulations and provides the related inputs required in their models. A geographic KIB performs the basic operations required of interaction models while providing functionality that facilitates spatial operations.

Two different models were composed for a study of the effects of human activities on landscapes and vice versa. The vegetation simulation determines the types and density of vegetation cover over a given map area. The agent-based model (ABM) consists of farmers who decide where they will farm and what types of farms they will establish. The Landscape simulation focuses on the land itself, showing effects on the fertility and elevation of a landscape over time. The models are composed using separate KIB models for the different sets of interactions required.

1.1 Thesis Goals

This thesis project has the goal of creating a generalized design for a knowledge interchange broker for geographic models. The design includes Unified Modeling Language (UML) specifications and diagrams that show the structure and behavior of the KIB. The classes and methods of the KIB were then implemented in the Java 8 programming language. The generalized design and classes created for this project could be extended and applied to other models where interactions based on 2D geographic maps are desired. The process developed here to create geographic KIB objects could also be used to make other variations of interaction models for other systems.

To aid the development and testing of the geographic KIB model, exemplar models were created and simulated for use in conjunction with the KIB (See Figure 1). These geographic versions of predator-prey models are described in the Sample Models section.

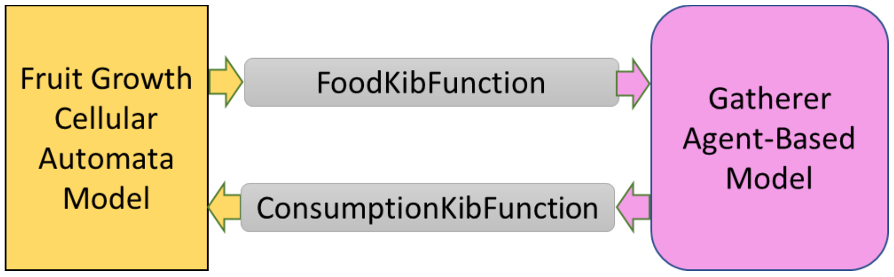

Fruit Growth Cellular Automata Model
FoodKibFunction
ConsumptionKibFunction
Gatherer Agent-Based Model

Figure 1: Composition of the Exemplar Models: The Fruit Growth and Gatherer Models.

Additionally, the geographic KIB design was to be applied to a set of simulations related to vegetation growth, landscape change, human decision-making, and interactions among them. While the content of the models is closely related, the simulators were not designed to meet each other’s I/O specifications, so one or more KIB Function units were required to allow their simulations to interact correctly.

1.2 Contributions

Previous KIB models have been created before this thesis. The GeoKIB differs from other KIBs in that it manages interactions involving spatial data formatted as cellular automata. Cellular automata represent changes to a geographic area over time [2]. For each time step, every map cell needs to read its inputs and produce its outputs as its state is determined. The updating format often varies greatly between different models. For example, one model of settlers may use the discrete-event representation provided by DEVS, while another vegetation model uses a strict, discrete-time update protocol. The interactions may include primitive data types, such as integer and floating-point values, but also includes entire maps of data, with values spread over cells of two-dimensional maps. Particular challenges arise when transferring cellular automata data between models, especially when the models use grids with different boundaries and cell sizes. The GeoKIB converts maps to different region specifications during simulations.

An important contribution of this thesis is the creation of a KIB design that can be modified for application to different sets of models. The design of the GeoKIB allows its general structure and functions to be extended for a particular set of interactions. The base GeoKIB model cannot be used as-is, but when configured, a GeoKIB models can be used for a variety of interactions. In particular, the ports of the GeoKIB can be customized for use with discrete time or discrete event models. The application of the GeoKIB for the exemplar models in this thesis show how this can be done.

Due to the need to compose simulations that do not all use the same simulation engine, the design of GeoKIB was made without the use of simulation engines. For this thesis, a Java-based implementation was chosen because of Java's strong object-oriented and strong typing capabilities. The design of the GeoKIB can be implemented in other programming languages or simulation engines, if necessary.

The GeoKIB was designed for the purpose of composing simulations that use the GRASS GIS system. Because of this, and because the cellular format of data used by the GeoKIB is similar to the grid-based format of GIS map data, the GeoKIB was made to be compatible with the GRASS functions. Instances of the GeoKIB have used GRASS operations for map visualization and data transfer. While the GeoKIB was designed specifically for the GRASS GIS, an interface allows a different GIS to be substituted if desired.

This unique set of capabilities and properties made it possible compose the Medland Modeling Laboratory (MML) with a separately designed vegetation growth model. These models were created by an interdisciplinary group of researchers for the School of Human Evolution and Social Change of Arizona State University. MML models the interactions of settlers and the landscape of the Iberian Peninsula. The vegetation growth model represents the growth of various species of plants native to the Iberian Peninsula. With the use of GeoKIB units, the two models were composed into a single model, in which vegetation amounts are provided to the MML model and human influences are provided to the vegetation model.

Use of synchronous reactive components for the models in Chapter 5 is a contribution.

1.3 Thesis Organization

Chapter 2 gives background information related to modeling, cellular automata, and model composition. Examples of prior research related to composition of cellular automata are cited.

Chapter 3 explains the approach used for heterogeneous composition of heterogeneous cellular automata for this thesis. Details of design and implementation of the interaction model, called the GeoKIB, are given. Chapter 4 gives descriptions of two exemplar models created to illustrate the process of model composition. Chapter 5 provides a detailed procedure for composing the exemplar models with the use of the GeoKIB. Chapter 6 concludes the thesis with a discussion of the findings and future work for the GeoKIB.

CHAPTER 2

BACKGROUND

This thesis was developed for use with simulations of progression over a geographical area. In particular, simulations of landscape, human agents, and vegetation needed to be composed. These simulations all represent data using cell-based structures for two-dimensional spatial data, along with GRASS GIS for data storage and visual representation. Some background is therefore provided for these simulations, cellular automata, and previous approaches used for model composition.

2.1 Medland Modeling Laboratory

Medland Modeling Laboratory (MML) is a simulator designed to represent the interactions between the landscape and humans living in the Iberian Peninsula [3]. The humans of a simulation are agents that determine the amount of farming, herding, and gathering they will do. Properties of the landscape are represented as geographical maps of data. The values stored within maps include village and farm locations, land fertility, and vegetation landcover. The GRASS GIS (Geographical Resources Analysis Support System Geographical Information System) [4] software suite is used to store and perform computations on these maps within the MML simulator.

A vegetation model was created by Miguel Acevedo of the University of Northern Texas which determines the numbers and properties of plants that grow in each area of a map for each species [5] [6]. The simulator requires a map of the terrain types of the simulated area for its initialization. During each simulation year, the vegetation simulation can accept a map of fire locations as an input at some time interval. The model gives, as outputs, geographic maps of values of properties of the plants. The properties in the output maps for each species include number of plants, basal area, and number of plants that died during each simulated year.

2.2 Cellular Automata

A variety of modeling methods have been used to represent spatiotemporal systems. A cellular automaton model is a representation of a system based on a tessellation of map cells over a rectangular cell space. All cells of the automaton have the same dimensions and the same specifications for input and output space, state space, and state transitions. State transitions for a single cell of a cellular automaton are dependent on the cell's inputs, the cell's state, and the state of the cell's neighbors [2]. The states of all the cells are updated with the same discrete-time behavior specification.

Papers by Meyer and Sarjoughian describes criteria and a specification for separate cellular automaton models that could be composed into a single model [7] [8]. The project described in this thesis is closely related to the idea of composable cellular automata. This project differs in that it emphasizes the process of model composition itself, and that the differences between the geographic representations, especially differences in spatial resolution, are handled within interaction models.

A variety of software tools are available for cellular automata models. One of these is called MASON (Multi-Agent Simulator of Neighborhoods) [9]. This tool is based on the agent-based modeling approach for creating, executing, and visualizing cellular automata models. Mathematica also supports run-time animation of cellular automata states, where cell behavior is determined by the mathematical formulas set for the model [10].

2.3 Related Work

Prior work has used different approaches for the modelling and composition of cellular automaton models. A few of these are presented here, along with a comparison of the approach used for this thesis.

2.3.1 Event-based Cellular Automaton Representations

Traditionally, the state of a cellular automaton is updated at discrete time steps. Some representations of cellular automata emphasize temporal changes. Peuquet and Duan describe a method of storing spatiotemporal GIS data primarily according to the time of change (event), rather than storing the full geographical state at every time interval. This organization of geographic data gives computational and conceptual advantages for geographic analyses of events, causes, and effects [11].

Each of the cells of a cellular automaton can be represented as a DEVS atomic model [12]. These atomic cells can be combined using their inputs and outputs with couplings to form cellular automata models that can have different topologies such as Von Neumann. Each cell can receive inputs from its neighbors, activating the external transition function of the cell. Wainer and Giambiasi developed a Timed Cell-DEVS model specification that uses DEVS to represent each of the cells of a cellular automaton [13]. This specification models the interactions of map cells while providing a mechanism to reduce the required computation for passive cells. The performance of cellular automaton simulations was improved in this manner. Compositions of Cell-DEVS models were created, but the assumption was made that all components of the compositions adhered to the same spatial resolution and timing system [7].

Parallel DEVS is also used to develop and simulate cellular automata [14]. A realization of this approach has been created and introduced to the DEVS-Suite simulator [15]. This CA-DEVS simulator, unlike other DEVS-based simulators, supports run-time forward and backward animation of cellular automata dynamics with superdense time trajectories.

2.3.2 Model Composition

Sarjoughian describes four approaches to composition of different models. With a mono modeling approach, all component models have the same formalism and specification [1]. The models can be composed without needing any formalism changes. As different systems are best represented by different models, restricting all component models to the same formalism can negatively affect the design of the models. For maximum modularity, different models should be designed independently of each other, resulting in natural differences between the models.

A super-formalism modeling composition approach involves packaging a model within another model with a different formalism [1]. For example, a discrete-time model can be encapsulated within a discrete event model. The model would then be seen as a discrete event model from an outside viewpoint. Once this approach is used to adapt all component models to one common formalism, that formalism can be used for the composition, as with mono-formalism modeling. The use of this approach must make considerations for the limitations in which formalisms can be packaged in this way.

Using the meta modeling approach to composition, a single model contains all the component models. The meta model must “interpret” the component models into its formalism, which becomes the standard for the composition [1]. This has some similarities to the super modeling approach, with the difference that meta modeling is the use of a single model to contain, control, and integrate all component models. Since the meta model must be designed to handle all of the specific component models, changes or additions to the models may be difficult to make, resulting in limitations of scalability.

The fourth approach to model composition is poly-formalism modeling. This approach uses a separate model, called a model interchange broker (KIB), to manage interactions between models that have different formalisms [1]. This approach allows interactivity between models without changing the component models or their interpretations. A KIB must be compatible with all connected models, resulting in some KIB models that do not follow the formalisms of any of the component models. As the design of a KIB depends on the models that are composed, a separate KIB must be designed for each type of interaction for composition.

Previous work for knowledge interchange broker (KIB) components emphasized the use of a KIB to control the interactions of combined models. Mayer and Sarjoughian describe the process of using a KIB to separate the agent and landscape operations of a simulation [16]. The KIB described here is different from the previous KIB. First, the KIB of this project is made to demonstrate a generalized design for interaction modeling for geographic simulations. This project includes designs and software that can be extended and applied to other interactions involving geographic maps. Second, the KIB used to combine the vegetation and MML simulations were applied to existing simulations, whereas previous KIB components were developed together with their simulations. Combining software simulations that were not designed to be compatible brings a different set of challenges.

Inokuchi, et al. explain a formulation for composition of cellular automata. Composition is defined for a single group (set of cells) in which the transition function for the composition of two cellular automata is equivalent to the functional composition of the transition functions for the component automata [17]. This type of composition may be useful when composing cellular automata that represent layered effects on a set of data. However, this definition of composition is not applicable to models that update independently on different data sets. In this approach, composition is defined for one cellular automata.

2.3.3 Multiresolution Modeling

Davis and Bigelow explain some advantages of low-resolution models when used in conjunction with high-resolution models. Lower-resolution models require fewer computational resources and data. Rules that are more understood or well-defined can often be applied to low-resolution models, and the results can be easier to comprehend and analyze at a higher level. Models for the different resolutions often have different domains of application and can be simulated concurrently, with data exchanged between the models [18]. In the case of spatiotemporal models, differences in resolution could refer to varying spatial precisions or to the types or numbers of measurements used. An interaction model is one approach to managing the interactions between the models without changing the resolutions of the models.

Kiester and Sahr proposed and implemented a methodology for a multi-resolution cellular automaton model. Its specification views cells from related models as neighbors in a hierarchical adjacency set, regardless of the differing resolutions and cell sizes. The implementation of the methodology requires that state transition rules for each cell be dependent on the percentage distribution of the states of the neighboring cells, rather than the number of neighboring cells that have the state. For example, in the Game of Life model [19], the transitions are redefined as dependent on the ratio of neighboring cells that are alive [20]. The ratio-based specification allows for interactions among models, with automatic adjustments for differing numbers of neighboring cells from other resolutions. The approach to interaction shows the assumption that all models use the same category of information greatly restricts the use of such an approach with heterogeneous models. In addition, working with a hierarchical adjacency set in this manner would most often require changing the component models and their transition rules to fit this ratio-based perspective of neighboring states.

Another approach to for multiscale modeling called Complex Automata (CxA) has been proposed in [21] and [22].A Complex Automata can be created as a composition of a set cellular automata of different spatial and temporal resolutions. Each cellular automaton of the set has a time step (Δt) and cell size (Δx) appropriate for its representation, along with temporal and spatial boundaries within the boundaries of the CxA. The CxA acts as a whole Cellular Automata that integrates CAs that have different spatial and temporal scales.. This is similar in purpose to this thesis, allowing models with different specifications and resolutions to be composed. This thesis differs in that the responsibilities of managing interactions are assigned to separate, dedicated interchange components. This difference in approach was chosen to allow interaction of models that were designed separately from the other models of the composition. While a well-designed CxA can allow more efficient composition of cellular automata components designed for aggregation, a technique was required that allows future models, with different temporal formalisms (such as discrete event), to be added to an existing system. Another difference is that in this thesis, two composed Cellular Automata models can have the same spatial dimensions. The CxA approach neither conceptualizes nor formalizes the Composable Cellular Automata model, which provides modularity at the level of individual Cellular Automata models for composition with component-based models [7].

CHAPTER 3

INTERACTION MODELING DESIGN

Object-oriented design principles focus on separating the design of complex systems into simpler components. The components can be combined to created the complete system. Decomposition facilitates simplicity of design, implementation, and testing, as each component holds a small subset of the whole system's complexity. While integration of the components is less difficult than designing the system as one component, integration is nonetheless a difficult task.

The purpose of the geographic knowledge interchange broker (GeoKIB) designed in this thesis is the composition of models with cellular spatial models. The GeoKIB is designed to allow component models to be combined into a single model when the component models have different specifications for input, output, and data structures. This design was created to be adaptable to both frequency- and event-based update types.

3.1 A Generalized Knowledge Interchange Broker Design

In the simplest case, a composition of simulations may consist of only two connected simulations. In this case, the function of the KIB can be integrated into the controller for the composed simulation. This would allow the same component to be responsible for controlling the individual simulations, their interactions, and the timing. However, if a composition involves more than two simulations, multiple KIB units may be required. Each of the simulations may connect to more than one KIB to regulate its interactions with each of the other components. Overlapping concerns prevent the KIB units from assuming responsibility for control of the component simulations and overall timing of the composition.

To prevent an overlap in control responsibilities, this project separates the functionality of handling interactions into instances of the KIBFunction class. Each instance of KIBFunction has the task of regulating a set of interactions between two simulations, where one simulation is designated as the input to the KIBFunction, and the other is the output. When two different simulations must communicate in multiple ways, multiple KIBFunction instances can be used as brokers between those same two simulations (See Figure 2). In particular, when two models must send information to each other, at least one KIBFunction instance is needed to regulate the data flow in each direction.

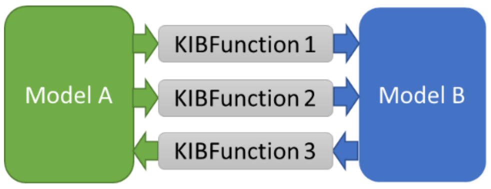


Figure 2: Example of Multiple KIBFunction Objects Used to Compose Two Models.

KIBFunction (shown in Figure 3) is an abstract class that needs to extended before it can be instantiated. In particular, the abstract methods state_transition() and is_transition_triggered() must be defined for any KIBFunction object. The state_transition() method sets the main state transition function of the KIBFunction object, including the acts of dequeuing inputs, transforming values, and enqueuing outputs. The method is_transition_triggered() returns a Boolean value that demines whether the state_transition() method should be called. As an example, is_transition_triggered() may be defined to return true only when a new input is available, indicating that state_transition() will not be activated when no input has been received.

For a KIBFunction object to have the ability to synchronize its timing with other components or the composed simulation, it needs the ability to read the time from an external source. For this purpose, the KIBFunction class depends on an object of the ISimClock interface to obtain the current time. An object that implements ISimClock is assigned to a KIBFunction object with the attach_sim_clock() method. KIBFunction then uses the method get_current_time() method from the ISimClock interface to obtain the current simulation time. Different simulations and KIBFunction objects may use the same or different clocks for their timings.

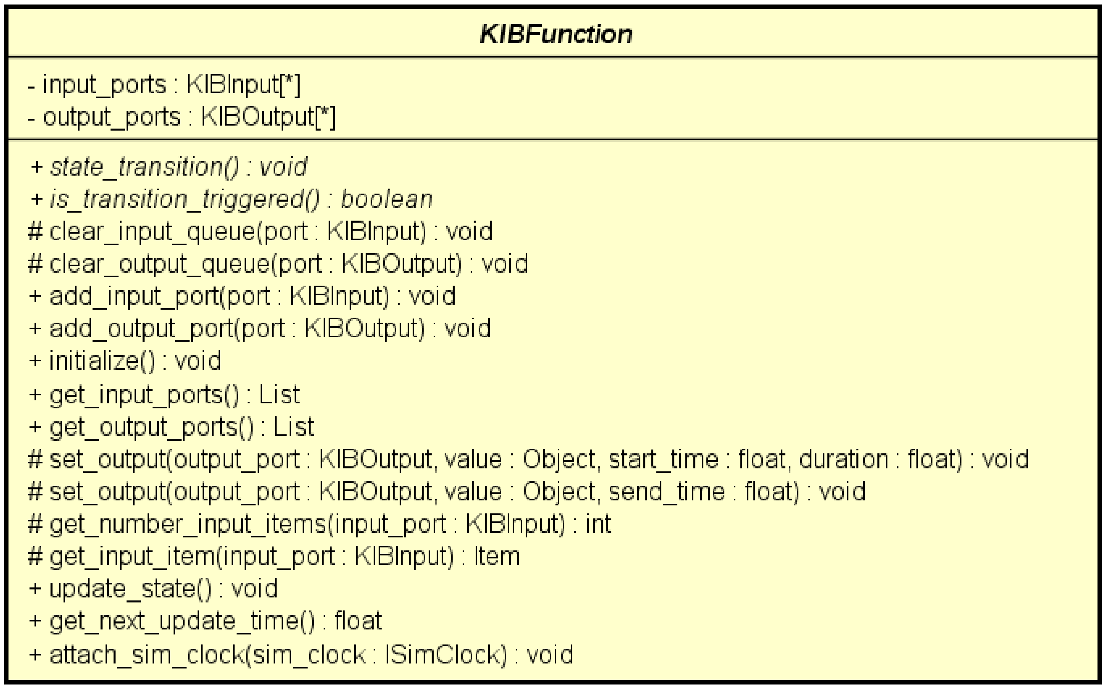

KIBFunction
- input_ports : KIBInput[*]
- output_ports : KIBOutput[*]
+ state_transition() : void
+ is_transition_triggered() : boolean
# clear_input_queue(port : KIBInput) : void
# clear_output_queue(port : KIBOutput) : void
+ add_input_port(port : KIBInput) : void
+ add_output_port(port : KIBOutput) : void
+ initialize() : void
+ get_input_ports() : List
+ get_output_ports() : List
# set_output(output_port : KIBOutput, value : Object, start_time : float, duration : float) : void
# set_output(output_port : KIBOutput, value : Object, send_time : float) : void
# get_number_input_items(input_port : KIBInput) : int
# get_input_item(input_port : KIBInput) : Item
+ update_state() : void
+ get_next_update_time() : float
+ attach_sim_clock(sim_clock : ISimClock) : void

Figure 3: UML Diagram of the Base KIBFunction Class.

A KIBFunction object receives inputs from dedicated input ports and sends outputs through assigned output ports (shown in Figure 4). Each of the ports is parameterized to specify the data type of the values that are sent through the port. In the shown diagrams, "ValueType" represents the associated data type. Each of the ports uses a queue to store value and retrieve values. Every value added to a port's queue has an associated simulation time. In KIBInput, the class for the input ports, each received value is linked to the time of receipt. In an output port, represented by the KIBOutput class, values are associated with a scheduled time for the output.

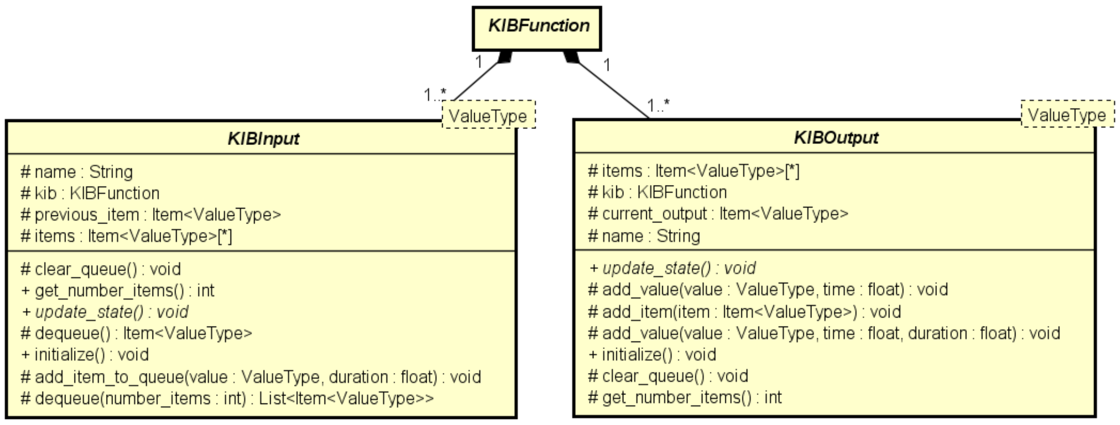

KIBFunction
1
1..*
ValueType
1
1..*
ValueType
KIBInput
# name : String
# kib : KIBFunction
# previous_item : Item<ValueType>
# items : Item<ValueType>[*]
# clear_queue() : void
+ get_number_items() : int
+ update_state() : void
# dequeue() : Item<ValueType>
+ initialize() : void
# add_item_to_queue(value : ValueType, duration : float) : void
# dequeue(number_items : int) : List<Item<ValueType>>
KIBOutput
# items : Item<ValueType>[*]
# kib : KIBFunction
# current_output : Item<ValueType>
# name : String
+ update_state() : void
# add_value(value : ValueType, time : float) : void
# add_item(item : Item<ValueType>) : void
# add_value(value : ValueType, time : float, duration : float) : void
+ initialize() : void
# clear_queue() : void
# get_number_items() : int

Figure 4: Input and Output Port Classes for the Interaction Model.

For both inputs and outputs, the class Item is used to tag values with a simulation time. The data type of Item (see Figure 5) is parameterized in the same way as KIBInput and KIBOutput. The types associated with a port object, Item object, and value of the Item are all identical. The value and time of an Item object are set by Item's constructor and cannot be altered. Optionally, a duration value can be set for the Item, which determines the length of time the value is available at the port. The default duration value of -1 indicates that the value does not have a limited duration and will remain available until replaced. The value, time, and duration of an Item may be read at any time with the use of the getter functions.

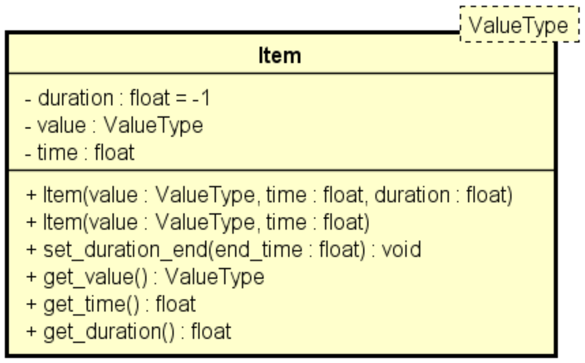

ValueType
Item
- duration : float = -1
- value : ValueType
- time : float
+ Item(value : ValueType, time : float, duration : float)
+ Item(value : ValueType, time : float)
+ set_duration_end(end_time : float) : void
+ get_value() : ValueType
+ get_time() : float
+ get_duration() : float

Figure 5: UML Diagram of the Item Class. This is used to associate input/output values with time and duration.

The class for KIBFunction's input ports, KIBInput (See Figure 6), is an abstract class, preventing it from being directly instantiated. This decision was made because the specific procedure for collecting inputs depends on whether the port should receive values passively or take inputs actively. The DEKIBInput (discrete-event knowledge interchange broker input) subclass of KIBInput is designed to accept values via its set_value() methods. In addition to enqueuing the given value, the method calls the update_state() method of the KIBFunction, which determines whether an action must be performed. The DEKIBInput class allows the KIBFunction to use discrete-event inputs, which trigger a reaction in the KIBFunction when an input is received.

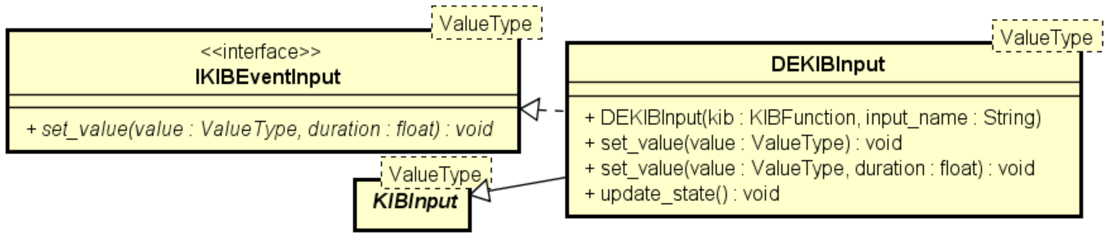

ValueType
<<interface>>
IKIBEventInput
+ set_value(value : ValueType, duration : float) : void
ValueType
KIBInput
ValueType
DEKIBInput
+ DEKIBInput(kib : KIBFunction, input_name : String)
+ set_value(value : ValueType) : void
+ set_value(value : ValueType, duration : float) : void
+ update_state() : void

Figure 6: Class Diagram for DEKIBInput. This class receives inputs using a discrete-event format.

The DTKIBInput (discrete-time knowledge interchange broker) class (See Figure 7) also inherits from KIBInput. DTKIBInput reads input values from a designated source at set time intervals. The source, from which the values are read, must be set using the class's set_source() method. The source must be one that implements the IOutput interface for outputs. DTKIBInput calls the get_value() method of the source to take an input value whenever the appropriate amount of time has passed. A DTKIBInput port may be useful if the KIBFunction needs to sample inputs at a rate different from the output frequency of the connected model.

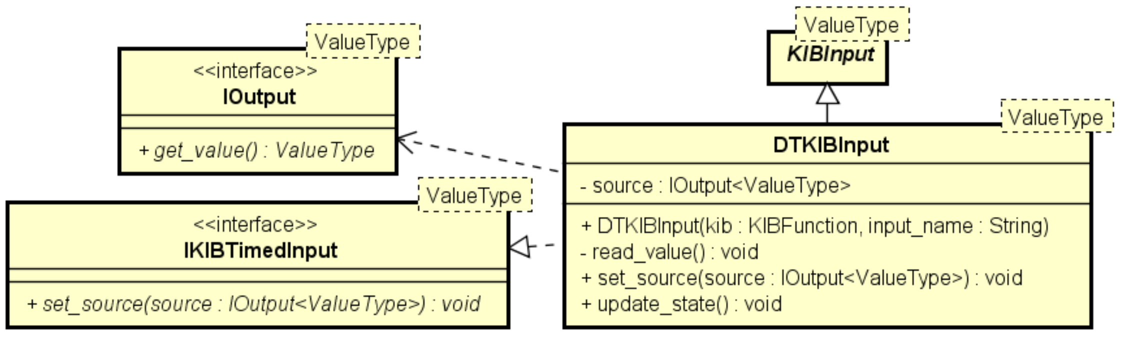

ValueType
<<interface>>
IOutput
+ get_value() : ValueType
ValueType
KIBInput
ValueType
DTKIBInput
- source : IOutput<ValueType>
+ DTKIBInput(kib : KIBFunction, input_name : String)
- read_value() : void
+ set_source(source : IOutput<ValueType>) : void
+ update_state() : void
ValueType
<<interface>>
IKIBTimedInput
+ set_source(source : IOutput<ValueType>) : void

Figure 7: Class Diagram for DTKIBInput. This class receives inputs using a discrete-time format.

As with KIBInput, KIBOutput is an abstract class and must be instantiated from a more specific subclass. PassiveKIBOutput (See Figure 8) is a subclass of KIBOutput that allows its current output to be read at any time. Its get_value() method returns the value in the queue with latest scheduled time not later than the current time. If there is no value in the queue, or if the duration has expired on the most recent value, null is returned.

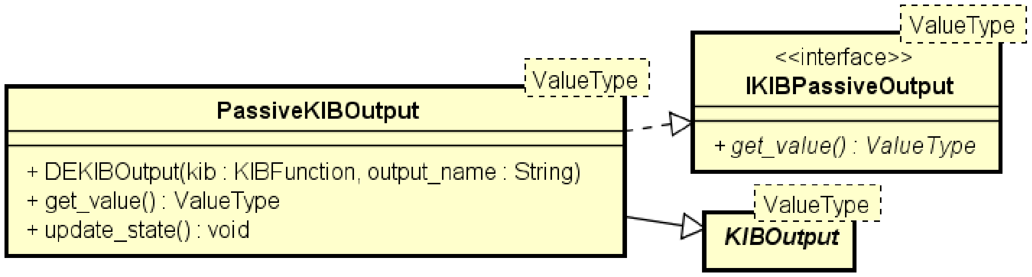


Figure 8: Class Diagram for PassiveKIBOutput. This allows outputs to be read by an external model.

ActiveKIBOutput (See Figure 9) is another subclass of KIBOutput that has the purpose of delivering values to external elements at scheduled times. This is slightly different from PassiveKIBOutput, which does not directly influence objects outside of the KIB. An ActiveKIBOutput object may be assigned any number of destinations to which it will deliver its values. The add_destination() method, which sets a destination for the ActiveKIBOutput, can be called multiple times to add more than one destination. When the simulation time reaches the scheduled output time for a value, that value is sent to every assigned destination. Each value is sent only once, so the "duration" of a value has no influence for ActiveKIBOutput. Values are sent by calling the set_value() method of each destination object, so each destination must implement the IInput interface.

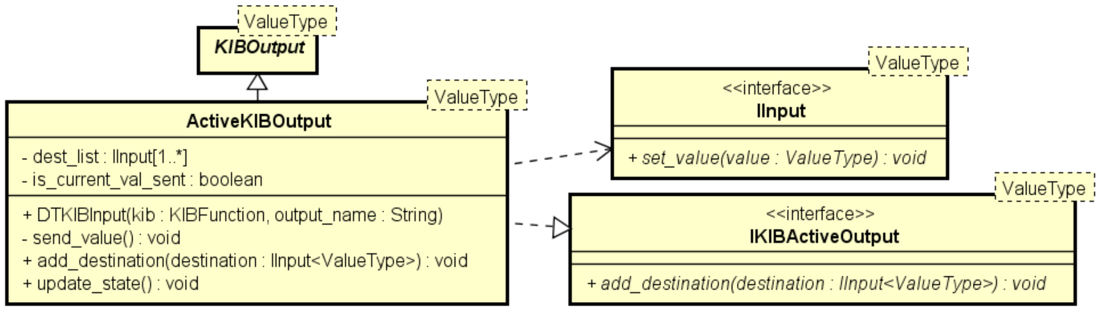
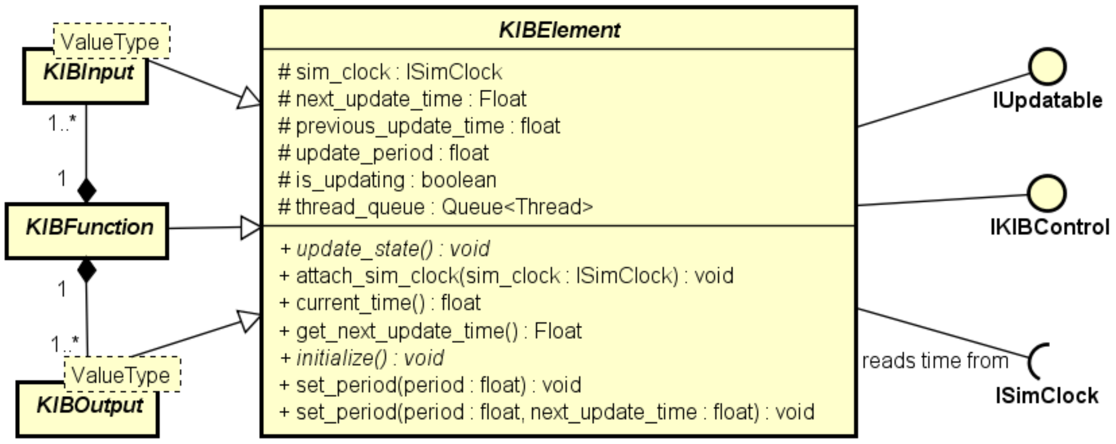

ValueType
KIBOutput
ValueType
ActiveKIBOutput
- dest_list : IInput[1..*]
- is_current_val_sent : boolean
+ DTKIBInput(kib : KIBFunction, output_name : String)
- send_value() : void
+ add_destination(destination : IInput<ValueType>) : void
+ update_state() : void
ValueType
<<interface>>
IInput
+ set_value(value : ValueType) : void
ValueType
<<interface>>
IKIBActiveOutput
+ add_destination(destination : IInput<ValueType>) : void

Figure 9: Class Diagram for ActiveKIBOutput. The class delivers output values to an external model.

All components of KIBFunction need access to simulation time and need to be updated. For this reason, KIBFunction, KIBInput, and KIBOutput all inherit from the KIBElement class (shown in Figure 10), which provides interfaces to set up and update the timing. A simulation clock can be attached to a KIBElement by using the attach_sim_clock() method. The get_current_time() method of the simulation clock is called to read the time, so the simulation clock must realize the ISimClock interface. When a clock is set for KIBFunction, the clock is automatically set for all of its current KIBInput and KIBOutput objects. There is no requirement for all KIBFunction elements to use the same simulation clock. The amount of time that should pass between required updates can be set with the set_period() methods.



KIBElement
# sim_clock : ISimClock
# next_update_time : Float
# previous_update_time : float
# update_period : float
# is_updating : boolean
# thread_queue : Queue<Thread>
+ update_state() : void
+ attach_sim_clock(sim_clock : ISimClock) : void
+ current_time() : float
+ get_next_update_time() : Float
+ initialize() : void
+ set_period(period : float) : void
+ set_period(period : float, next_update_time : float) : void
ValueType
KIBInput
1..*
1
KIBFunction
1
1..*
ValueType
KIBOutput
IUpdatable
IKIBControl
reads time from
ISimClock

Figure 10: Class Diagram for KIBElement. This superclass of all KIB elements is used for timing and synchronization.

The KIBElement method update_state() is used as a “wakeup” function. When the method is called, the object checks whether any updates are needed, performs any scheduled actions, then updates the attributes accordingly. Calling update_state() more than once does not cause any inconsistencies in the KIB’s functionality or state, so the method may be called at any time to ensure that the state is current. In the class KIBElement, update_state() is an abstract method, meaning that it has no “default” implementation and uses polymorphism to provide the implementation appropriate for the subclass.

The implementation of update_state() depends on the class and its specific purposes. In KIBFunction, update_state() first calls the update_state() method of every attached KIBInput port. Then, if the is_transition_triggered() method returns true, state_transition() is called. Finally, update_state() is called for each of the attached KIBOutput objects. To prevent unnecessary looping, the update_state() methods of KIBInput and KIBOutput do not call any object's update_state() method.

In DTKIBInput, update_state() uses its stored information about the time of the previous update, the current time, the update period, and the scheduled time for the next input to determine whether an input should be read at this time. If an input value is read and added to the input queue, the timing information of the DTKIBInput is updated to prepare for the next input. Using a similar procedure, the update_state() method of ActiveKIBOutput manages the times that outputs are sent to its destinations. In DEKIBInput, updates are only needed when an inputs are given, so update_state() does nothing. In PassiveKIBOutput, update_state() uses the scheduled time and duration of the items in the queue to update the value that will be read by the get_value() method.

3.2 Geographic Interaction Model Design

The design of the KIBFunction is not particular to any particular data type or class for inputs and outputs. The input and output ports can be parameterized to operate together with any set of models. While this is compatible with geographic models, it is best to extend the design of KIBFunction to address issues specific to geographic maps.

The basic data structure for values spread across a geographic area is a geographic map, represented as an array of values associated with map locations. For this project, the GeoMap class was created for this representation. GeoMap is closely associated with the MapRegion class (See Figure 11), which defines the boundaries and spatial resolutions for a set of GeoMaps. The north_bound, south_bound, west_bound, and east_bound attributes of MapRegion are boundary values, which may be latitude and longitude values or any relative measurements of location. The num_rows and num_cols values determine the sizes of each of the dimensions of the array in associated GeoMap objects. These attribute values are set during instantiation of a MapRegion object and cannot be changed afterward. If region changes are necessary during a simulation, the change must be represented by the creation of a new MapRegion object. The null_value attribute is the value designated with a lack of a value at any location.

Upon instantiation, a GeoMap object is associated with a MapRegion object. The MapRegion cannot be removed, switched, or altered during the lifespan of the GeoMap. The size of the two-dimensional array of values is based on the numbers of rows and columns in the MapRegion. GeoMap has methods to return the value of the cell at a set of index values or at a specified geographic location. The set_value() method can set the value map value at the specified row and column number. The is_cumulative_value() method reveals whether the GeoMap's values are cumulative values that would be added when map cells are aggregated (such as numbers of fruit), or average values that should not be collected over areas (such as rainfall density).

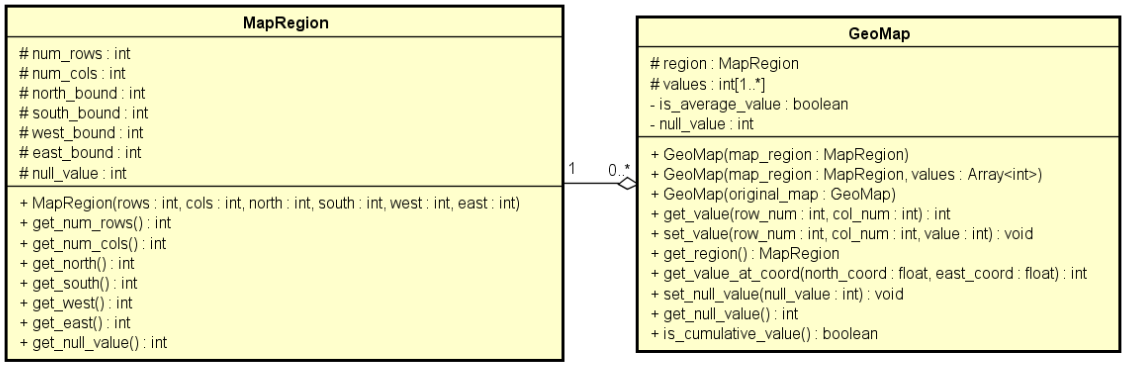

MapRegion
# num_rows : int
# num_cols : int
# north_bound : int
# south_bound : int
# west_bound : int
# east_bound : int
# null_value : int
+ MapRegion(rows : int, cols : int, north : int, south : int, west : int, east : int)
+ get_num_rows() : int
+ get_num_cols() : int
+ get_north() : int
+ get_south() : int
+ get_west() : int
+ get_east() : int
+ get_null_value() : int
1
0..*
GeoMap
# region : MapRegion
# values : int[1..*]
- is_average_value : boolean
- null_value : int
+ GeoMap(map_region : MapRegion)
+ GeoMap(map_region : MapRegion, values : Array<int>)
+ GeoMap(original_map : GeoMap)
+ get_value(row_num : int, col_num : int) : int
+ set_value(row_num : int, col_num : int, value : int) : void
+ get_region() : MapRegion
+ get_value_at_coord(north_coord : float, east_coord : float) : int
+ set_null_value(null_value : int) : void
+ get_null_value() : int
+ is_cumulative_value() : boolean

Figure 11: Diagram of the Classes MapRegion and GeoMap. These are used to represent a distribution of values over a geographic area.

The class GeoKibFunction, shown in Figure 12, inherits from the KIBFunction class and adds functionality specific to working with geographic transformations between models. A GeoKibFunction is assigned a MapRegion object for its inputs and another MapRegion for its outputs. The differences between the MapRegion objects is used as the basis for the spatial conversions that are performed within the class. As with its parent class, GeoKibFunction is an abstract class, meaning that it cannot be directly instantiated. The abstract methods from KIBFunction, state_transition() and is_transition_triggered(), are not defined in GeoKibFunction and must be defined in a specialized extension of the class.

GeoKibFunction provides a structure for the transformations between two different geographic standards. It also provides the spatial_conversion() method, which converts a GeoMap from the input region to the output region. The abstract method measurement_conversion() is a set of other operations that must be performed in the transformation between models. Such operations may be as simple as converting measurement units or as complex as predicting future values from a collection of relevant input values. An object may be attached to allow access to operations from a GIS, such as GRASS.

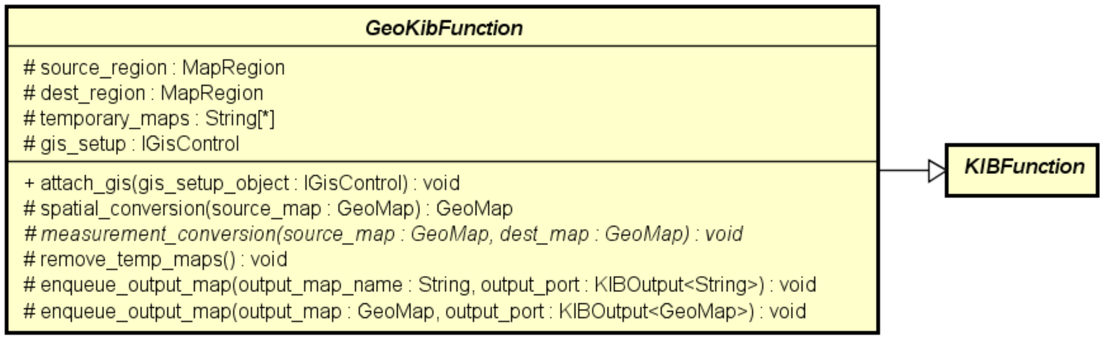

GeoKibFunction
# source_region : MapRegion
# dest_region : MapRegion
# temporary_maps : String[*]
# gis_setup : IGisControl
+ attach_gis(gis_setup_object : IGisControl) : void
# spatial_conversion(source_map : GeoMap) : GeoMap
# measurement_conversion(source_map : GeoMap, dest_map : GeoMap) : void
# remove_temp_maps() : void
# enqueue_output_map(output_map_name : String, output_port : KIBOutput<String>) : void
# enqueue_output_map(output_map : GeoMap, output_port : KIBOutput<GeoMap>) : void
KIBFunction

Figure 12: Class Diagram of GeoKibFunction. The class models interactions involving geographic maps and regions.

The spatial_conversion() method accepts one GeoMap object as a parameter, which must have the same MapRegion object assigned as the source_region attribute of the GeoKibFunction. Within spatial_conversion(), a two-dimensional array of values is created with the number of row and columns specified in the destination_region attribute of GeoKibFunction. Each element of the new array corresponds to a geographical area, with values that can be determined for its north, south, west, and east boundaries of a destination map cell. These boundaries can then be matched with the original GeoMap to find an overlapping area. Because of differences in resolution, the destination cell may not align with the boundaries of cells from the source map. The destination cell may overlap multiple cells from the source map in different area proportions, as shown in Figure 13.

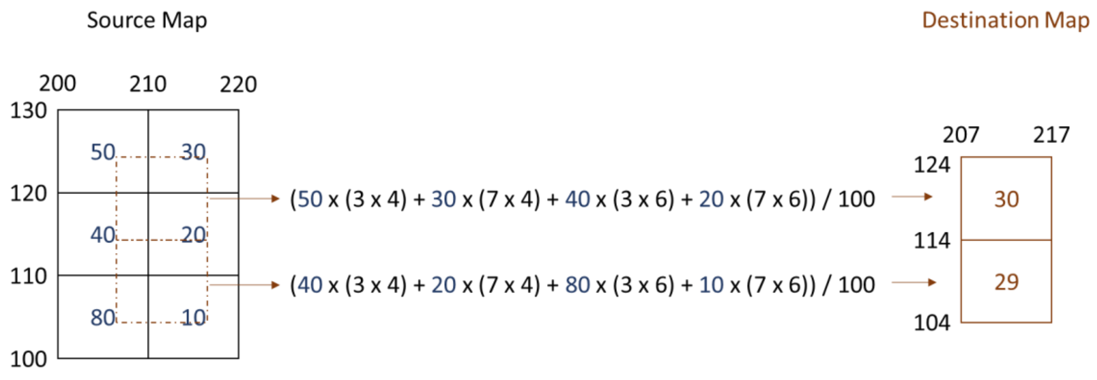

Source Map
Destination Map
200
210
220
130
120
110
100
50
30
40
20
80
10
(50 x (3 x 4) + 30 x (7 x 4) + 40 x (3 x 6) + 20 x (7 x 6)) / 100
(40 x (3 x 4) + 20 x (7 x 4) + 80 x (3 x 6) + 10 x (7 x 6)) / 100
207
217
124
114
104
30
29

Figure 13: Example of the Conversion to a Region with Differently Aligned Cell Boundaries.

The value of one destination cell in spatial_conversion() is determined by finding an average of the values from the corresponding (overlapping) area of the source map, weighted based on area sizes. For each of the source map cells, its value is multiplied by the size of the area of overlap between the source cell and destination cell. The products are added from all the corresponding source cells to create a weighted sum. Similarly, the sizes of the overlapping areas are added to find a total area. The weighted sum is divided by the total area to find the weighted average value for the destination cell. Finally, if the map is designated as having cumulative values, the weighted average value is converted to a cumulative value by multiplying by the ratio of the cell areas in the destination and source regions.

When analyzing the complexity of this algorithm, it is important to note that the information from the MapRegion can be used to determine the row and column numbers for a map. The value of each cell of the destination map must be determined. If the destination cell's boundaries do not align with those of a source cell, the destination cell may overlap some of the area of more than one source cell. Weighted sums must be accumulated for all of the areas of overlap, but the computations are not done more than once for any area of the source map. If the source map has r_s rows and c_s columns, and the destination map has r_d rows and c_d columns, the area of overlap will, at most, have $r_s + r_d$ rows and $c_s + c_d$ columns. Therefore, the computational complexity of the spatial_conversion() function is, in the worst case, $O((r_s + r_d) \cdot (c_s + c_d))$.

CHAPTER 4

EXEMPLAR MODELS

Since the purpose of this thesis project is to develop a method for the composition of spatiotemporal component models, sample models were required to refine and demonstrate the process. Two exemplar models models were created as potentially composable models. The first is a fruit growth model that provides the amount of fruit at different location cells during each time step. The second exemplar model is an agent-based gatherer model, in which the gatherers search an area for food. The exemplar models are separately developed, but they are conceptually related, making them good candidates for composition via GeoKIB.

4.1 Fruit Growth Model Overview

The fruit growth model developed for this thesis is a representation of the growth of fruit across a geographic area. The area is divided into distinct map cells based on the map region settings for the model. For each time step of the discrete-time model, the growth in the amount of fruit in a cell is determined as by multiplying a growth factor by the number of fruit within a given radius of the center of the cell. The amount of fruit can also be influenced by the external influences through an input to the model. The input map represents the change in the number of fruit, at each map cell, due to external influences.

4.2 Fruit Growth Model Specifications

The fruit growth model is shown as a synchronous reactive component in Figure 14. The influenceradius and growthfactor values are unchanging settings that influence the reproduction rate of fruit during each time step. The GrassMap fruitmap is a geographic map whose values indicate the amount of fruit across the model area. The fruitmap values are updated each time step. The input changemap is a map of values that specify the changes in the amount of fruit at each location. The values of changemap can be positive or negative, corresponding to fruit that is added or removed by external causes. The output GrassMap fruitamounts is made from the state GrassMap fruitmap, so its values are the same as the numbers of fruits over the area at the time of the output.

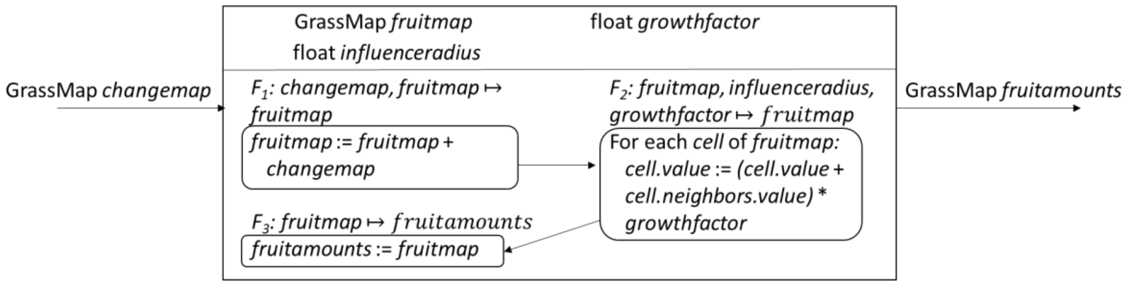

GrassMap *fruitmap*
float *growthfactor*
float *influenceradius*
GrassMap *changemap*
F_1: *changemap, fruitmap* ↦ *fruitmap*
fruitmap := *fruitmap* + *changemap*
F_2: *fruitmap, influenceradius, growthfactor* ↦ *fruitmap*
For each *cell* of *fruitmap:* *cell.value* := (*cell.value* + *cell.neighbors.value*) * *growthfactor*
F_3: *fruitmap* ↦ *fruitamounts*
fruitamounts := *fruitmap*
GrassMap *fruitamounts*

Figure 14: Specification for the Fruit Growth Model, Shown as a Reactive Component.

At each time step, the changes from the input changemap are added to the values of the fruitmap. After this change, the amount of fruit that grows during the time step at each map cell is found by multiplying number of fruit near the cell by the growthfactor value. Only fruit with a distance less than influenceradius from the center of the cell of interest are considered neighboring fruit, for purposes of finding the amount of new fruit that grows.

The amount of new fruit at each cell is added to the values of fruitmap. The fruitamount output is a copy of the current fruitmap GrassMap.

4.3 Fruit Growth Model Structure and Algorithms

The fruit growth model has a structure that allows tracking and manipulation of the amount of fruit distributed over the geographic area. The general class structure is shown in Figure 15. The GrowthModel class encompasses the overall fruit growth simulation. GrowthModel has attributes to track the state of the simulation, including the current simulation time, the last received input map, the time associated with the input map, and the map of fruit amounts as a GrowthMap object.

The GrowthModelSettings class contains the settings that can be set for a simulation. The settings are set by the class constructor but are not changed after instantiation. The other methods of the class only return the values of the settings of the class.

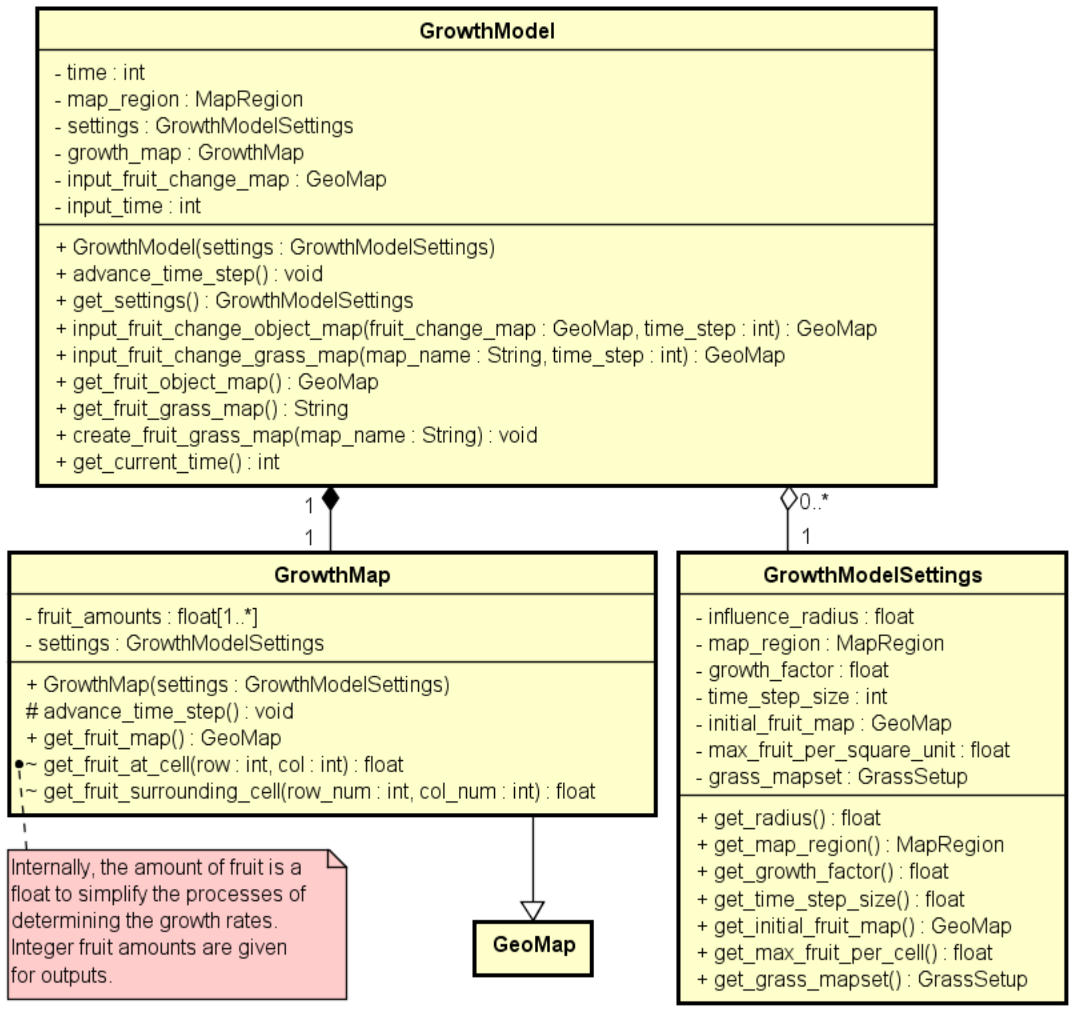

GrowthModel
- time : int
- map_region : MapRegion
- settings : GrowthModelSettings
- growth_map : GrowthMap
- input_fruit_change_map : GeoMap
- input_time : int
+ GrowthModel(settings : GrowthModelSettings)
+ advance_time_step() : void
+ get_settings() : GrowthModelSettings
+ input_fruit_change_object_map(fruit_change_map : GeoMap, time_step : int) : GeoMap
+ input_fruit_change_grass_map(map_name : String, time_step : int) : GeoMap
+ get_fruit_object_map() : GeoMap
+ get_fruit_grass_map() : String
+ create_fruit_grass_map(map_name : String) : void
+ get_current_time() : int
1
1
0..*
1
GrowthMap
- fruit_amounts : float[1..*]
- settings : GrowthModelSettings
+ GrowthMap(settings : GrowthModelSettings)
# advance_time_step() : void
+ get_fruit_map() : GeoMap
~ get_fruit_at_cell(row : int, col : int) : float
~ get_fruit_surrounding_cell(row_num : int, col_num : int) : float
GrowthModelSettings
- influence_radius : float
- map_region : MapRegion
- growth_factor : float
- time_step_size : int
- initial_fruit_map : GeoMap
- max_fruit_per_square_unit : float
- grass_mapset : GrassSetup
+ get_radius() : float
+ get_map_region() : MapRegion
+ get_growth_factor() : float
+ get_time_step_size() : float
+ get_initial_fruit_map() : GeoMap
+ get_max_fruit_per_cell() : float
+ get_grass_mapset() : GrassSetup
Internally, the amount of fruit is a float to simplify the processes of determining the growth rates. Integer fruit amounts are given for outputs.
GeoMap

Figure 15: Class Diagram of the Implementation of the Fruit Growth Model.

The GrowthMap class inherits from the GeoMap class, which allows interaction with GRASS. The class includes, as an attribute, a two dimensional array for the amounts of fruit currently at each map cell. Floating-point values are used, instead of integers, for storage and computations of fruit amounts within the GrowthMap class to simplify computations involving fractional growth factor amounts. Output maps produced by GrowthMap are of the integer values that result from truncating the decimal portions of the fruit amounts. No separate class is used for the map cells of GrowhMap because only one value is needed for each map cell.

The advance_time_step() method of GrowthMap is called to progress the state of the fruit amounts by one time step. For each map cell, the method calls the get_fruit_surrounding_cell() method to find the amount of nearby fruit, then multiplies the result by the growth factor to determine the amount of new fruit that grows at the cell. The amount of fruit near a given cell is found by incrementing across map cells, adding the number of fruit from cells whose distance from the given cell is less than the radius of influence. For cases in which the part of a cell is inside the circle of influence, a portion of the number of fruit is counted as near the given cell. The ratio of the fruit that is included is found as an approximation; the rectangular area that circumscribes the circle of influence is counted as the nearby area in any map cell.

4.4 Fruit Growth Simulation Setup and Use

A group of settings can be customized for a simulation of the fruit growth model. These settings include amounts for the simulation's growth factor, radius of influence, maximum fruit per square unit, and amount of time per time step. The map region, which includes the boundaries and resolution of the simulated geographic area, must also be set. The GRASS mapset can be set for interactivity with GRASS. The initial amounts for the fruit can be set as a map of values, or if not set, will be determined randomly over the simulation area.

To create a simulation of the fruit growth model in Java, the map region must first be set as an instance of the MapRegion class, using the following constructor.

```
public MapRegion(int rows, int cols, int north, int south, int west, int east)
```

To allow the use of GRASS, an object of the GrassSetup class must be set. In its constructor, the names of an existing GRASS mapset and its parent location must be given for the values of mapset and location, respectively. For gisdbase, the absolute file path of the folder containing the GRASS location must be given.

```
public GrassSetup(String gisdbase, String location, String mapset)
```

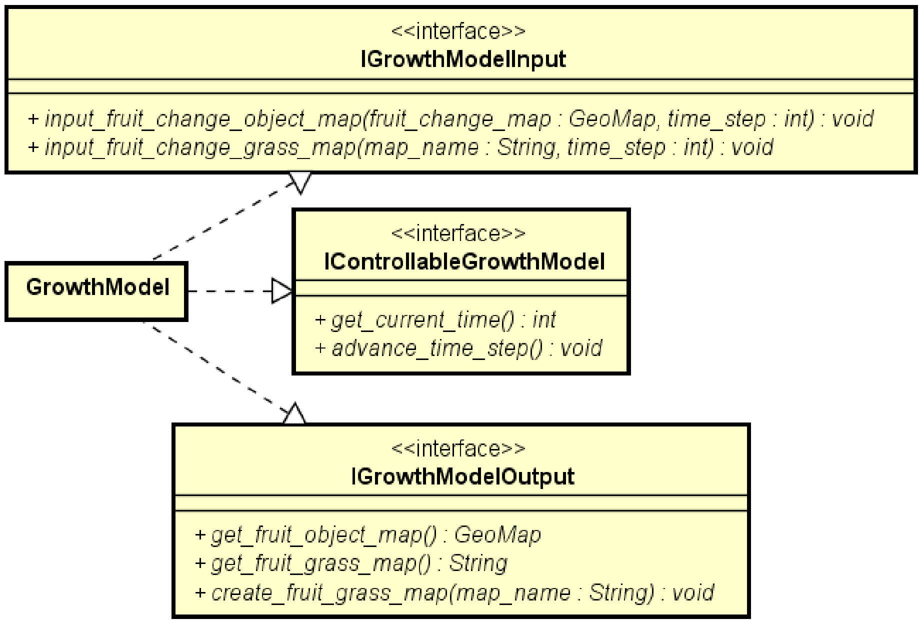
The set_grass_region() method of the GrassSetup object can be used to set the current GRASS region. Before the simulation for the fruit growth model can be instantiated, its settings must be set by creating an object of the class GrowthModelSettings, using the following constructor method.

```
public GrowthModelSettings(
        MapRegion map_region, float growth_factor,
        float radius_of_influence, int time_step_size,
        float max_fruit_per_square_unit,
        GeoMap initial_fruit_map, GrassSetup grass_mapset
)
```

If no GeoMap object is to be used for the initial fruit map values, a null value may be given for the initial_fruit_map argument. Once the GrowthModelSettings object has been initiated, its object can be used as the argument in the GrowthModel constructor, which will create and initialize the fruit growth simulation.

```
public GrowthModel(GrowthModelSettings settings)
```

After the simulation has been instantiated, GrowthModel’s interfaces (shown in Figure 16) may be used to provide inputs, control the simulation, and obtain outputs.



<<interface>>
IGrowthModelInput
+ input_fruit_change_object_map(fruit_change_map : GeoMap, time_step : int) : void
+ input_fruit_change_grass_map(map_name : String, time_step : int) : void
GrowthModel
<<interface>>
IControllableGrowthModel
+ get_current_time() : int
+ advance_time_step() : void
<<interface>>
IGrowthModelOutput
+ get_fruit_object_map() : GeoMap
+ get_fruit_grass_map() : String
+ create_fruit_grass_map(map_name : String) : void

Figure 16: Interfaces Implemented by the GrowthModel Class.

The input_fruit_change_object_map() or input_fruit_change_grass_map() method can be used to give an input map of the changes in fruit due to external influences as a GeoMap object or as the name of an existing GRASS map, respectively. The values of the input map may be positive, negative, or zero-value integers, indicating whether the fruit was added, removed, or not changed. Along with the input, the value of the time step during which the change is to be applied must be specified. The input map specifies the amount of change in the amounts of fruit during the time interval immediately preceding the specified time step.

The advance_time_step() method progresses the simulation by one time step. This causes the input map, if provided, to change the values of the amounts of fruit across the simulation map. Then, the growth of fruit from reproduction is determined and added to the current fruit amounts. The current simulation time may be obtained by using the get_current_time() method.

A GeoMap object of the amounts of fruit at each map cell can be obtained from the simulation with the use of the get_fruit_object_map() method. The create_fruit_grass_map() and get_fruit_grass_map() methods can be used to provide the same output map to the GRASS mapset.

4.5 Gatherer Model Overview

The gatherer model represents a set of gatherers that can look for available food across a landscape. A gatherer is a person or animal that needs food from the environment but does not directly cultivate the food. Time in the model progresses at evenly separated discrete steps. Each of the gatherers is associated with a geographic location. Food is represented only as the amount available in every location of the represented area.

During each time step of a simulation, each gatherer chooses an action based on the current state. When a gatherer finds an unclaimed area with more food than any other visible map cell, the gatherer will settle at the map cell, claiming it as a more permanent residence and becoming a settler. Gatherers, when searching for food, will not include settled map cells in their searches; they will only search for unclaimed land. Each map cell can be claimed by at most one gatherer. Settlers remain at their current location unless the amount of food at the cell drops to below a minimum threshold value. If there is insufficient food at the settled map cell, the settler will return to a wanderer status and search for an available location with more available food.

Wanderers, or gatherers that are not settled, move to different locations of the landscape depending on which available map cell has the most available food. Gatherers are assigned a radius of vision, which also represents the maximum distance a gatherer may travel during one time step. If a wanderer sees that an unoccupied map cell has more food than the current location, the wanderer will set that map cell as the destination for the next time step. The wanderers move to their destinations at the beginning of the next time step. This may result in more than one wanderer moving to the same location. If the wanderer does not see any available locations with more food than the current position, the wanderer will claim the map cell and become a settler. This process eventually separates the gatherers that collect at a location.

Gatherers can die from age or from lack of food. A "life expectancy" is assigned to the gatherers, indicating the maximum amount of time each gatherer may live. Any gatherer whose age exceeds this amount within a simulation dies automatically. The ages of gatherers at the start of a simulation are set randomly. Gatherers also have "appetite" and "satisfaction" state variables that indicate the amount of food required by the gatherer and the amount of time the gatherer can survive without food, respectively. If, in any time step, the amount of available food at a gatherer's current location is less than the gatherer's appetite, the satisfaction of the gatherer decrements. If the satisfaction value reaches 0, the gatherer dies. When a gatherer dies, it is removed from the simulation, and it loses its claim to the settled land.

In this model, reproduction is only done by gatherers that have been settled for a period of time. While settled, a gatherer produces new gatherers periodically, at a rate set by the simulation settings. A new gatherer is created as a wanderer at the same map cell as its parent with an age of 1. Wanderers do not reproduce before settling.

4.6 Gatherer Model Specifications

The gatherer model as a whole can be specified as a synchronous reactive component (Figure 17), using the notation described by Alur [23]. The model's state consists of status of all the settlers and map cells of the model. During each time step, the gatherer model must receive an input map indicating the amount of food available in each of the map cells. At any time, an output may be obtained from the gatherer model in the form of a map of the number of gatherers across the map.

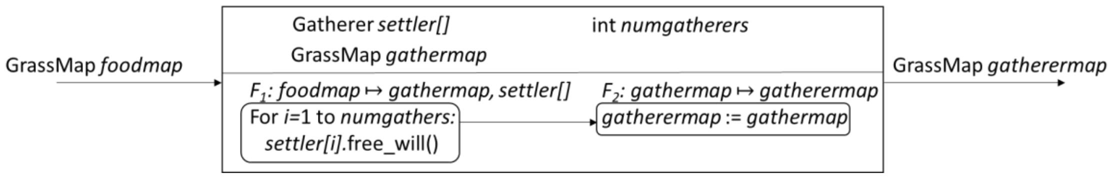

Gatherer *settler[]*
int *numgatherers*
GrassMap *gathermap*
GrassMap *foodmap*
GrassMap *gatherermap*
F_1: *foodmap* ↦ *gathermap*, *settler[]*
For *i*=1 to *numgathers:*
settler[i].free_will()
F_2: *gathermap* ↦ *gatherermap*
gatherermap := *gathermap*

Figure 17: Specification for the Gatherer Model, Shown as a Reactive Component.

Within the gatherer model, each of the gatherers may be shown as a different component, as shown in Figure 18. In this model, no inputs are shown. This should not be mistaken to mean that a gatherer is independent of outside influence, because the state of the currently occupied and nearby map cells influence the decisions and state of a gatherer. The only output of a gatherer is its current location, but the gatherer may directly influence the states of map cells.

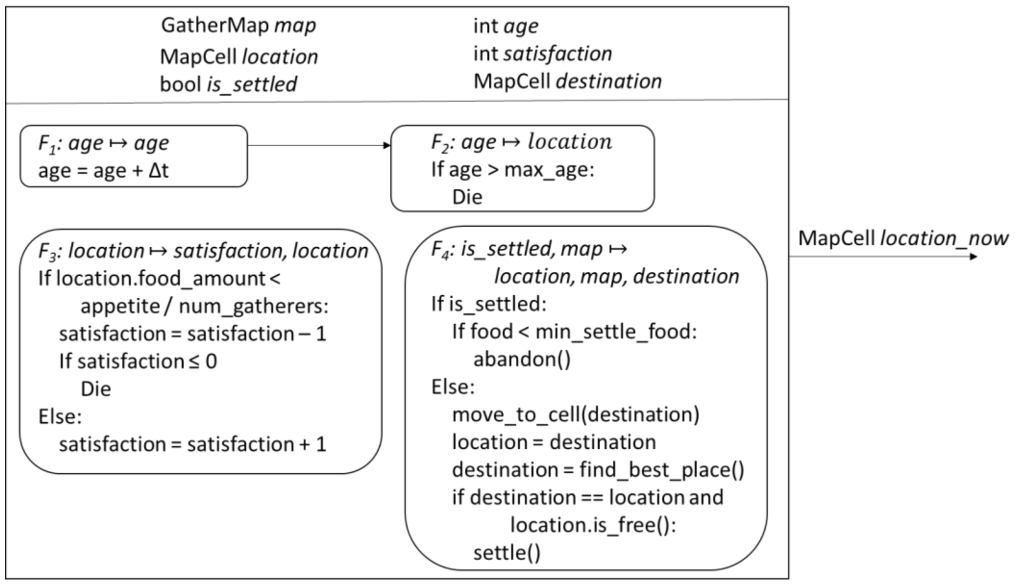

GatherMap *map*
MapCell *location*
bool *is_settled*
int *age*
int *satisfaction*
MapCell *destination*
F_1: *age* ↦ *age*
age = age + Δt
F_2: *age* ↦ *location*
If age > max_age:
Die
F_3: *location* ↦ *satisfaction, location*
If location.food_amount < appetite / num_gatherers:
satisfaction = satisfaction – 1
If satisfaction ≤ 0
Die
Else:
satisfaction = satisfaction + 1
F_4: *is_settled, map* ↦ *location, map, destination*
If is_settled:
If food < min_settle_food:
abandon()
Else:
move_to_cell(destination)
location = destination
destination = find_best_place()
if destination == location and location.is_free():
settle()
MapCell *location_now*

Figure 18: Reactive Component Representation of a Gatherer in the Gatherer Model.

4.7 Gatherer Model Structure and Algorithms

The implementation of the gatherer model is based on the use of geographic data structures for the current simulation state and for the model’s inputs and outputs. The model settings, map region, GRASS mapset settings, and current model state are stored as objects within the object of GathererModel, which represents the current simulation. The GatherMap object is divided into GatherMapCell objects, each of which holds the Gatherer objects and information about the current amount of food. The GatherMap class inherits from the GeoMap class of the Geographic package, and inputs and outputs of the model are also represented as GeoMap objects. Because of this, all of the map objects of GathererModel have the ability to import and create GRASS data. The class structure for the gatherer model is shown in Figure 19.

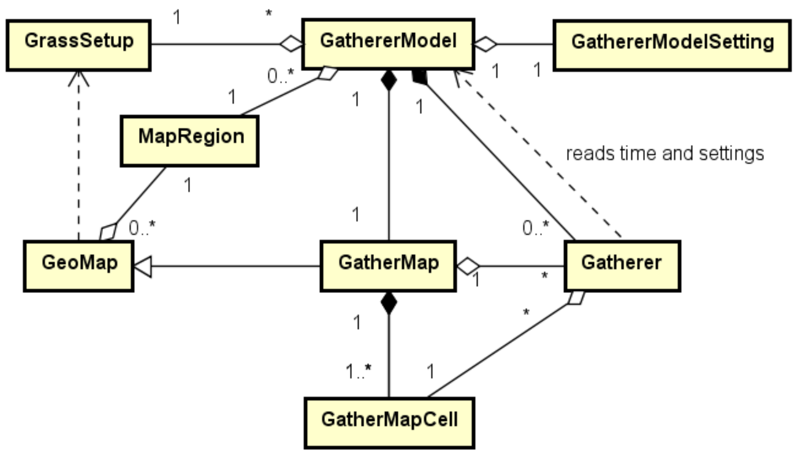


Figure 19: Overall Class Diagram of the Implementation of the Gatherer Model.

The GathererModel class (See Figure 20) is responsible for tracking the model state, accepting inputs, providing outputs, and progressing the simulation. A list of all gatherers is stored by GathererModel, allowing signals to progress to be given to the gatherers.

A cell of the GatherMap class (See Figure 21) stores the list of gatherers at that location, the amount of food at the location, and the settler of the location. The GatherMap class gives access to information from its cells, as well as providing access to GRASS functions through its inheritance of the GeoMap class.

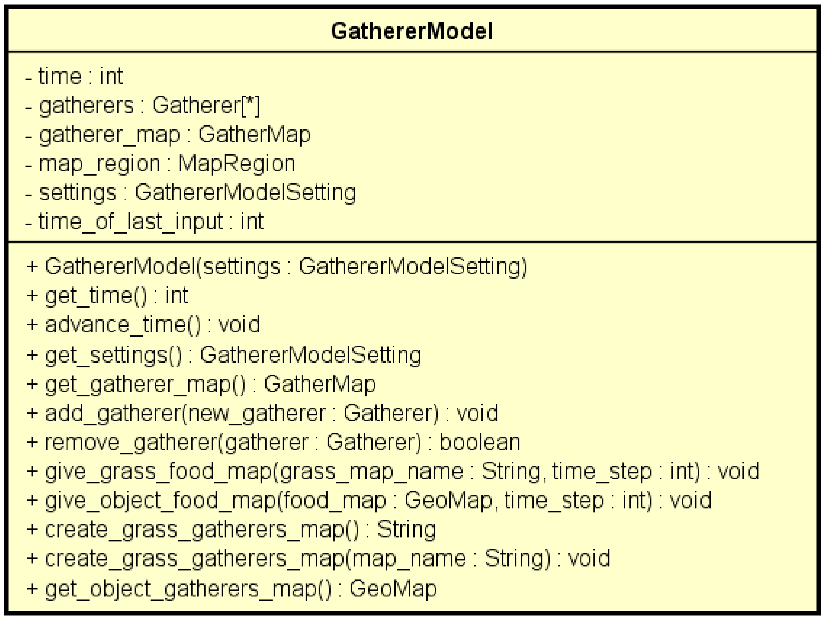

GathererModel
- time : int
- gatherers : Gatherer[*]
- gatherer_map : GatherMap
- map_region : MapRegion
- settings : GathererModelSetting
- time_of_last_input : int
+ GathererModel(settings : GathererModelSetting)
+ get_time() : int
+ advance_time() : void
+ get_settings() : GathererModelSetting
+ get_gatherer_map() : GatherMap
+ add_gatherer(new_gatherer : Gatherer) : void
+ remove_gatherer(gatherer : Gatherer) : boolean
+ give_grass_food_map(grass_map_name : String, time_step : int) : void
+ give_object_food_map(food_map : GeoMap, time_step : int) : void
+ create_grass_gatherers_map() : String
+ create_grass_gatherers_map(map_name : String) : void
+ get_object_gatherers_map() : GeoMap

Figure 20: UML Diagram of the Main GathererModel Class.

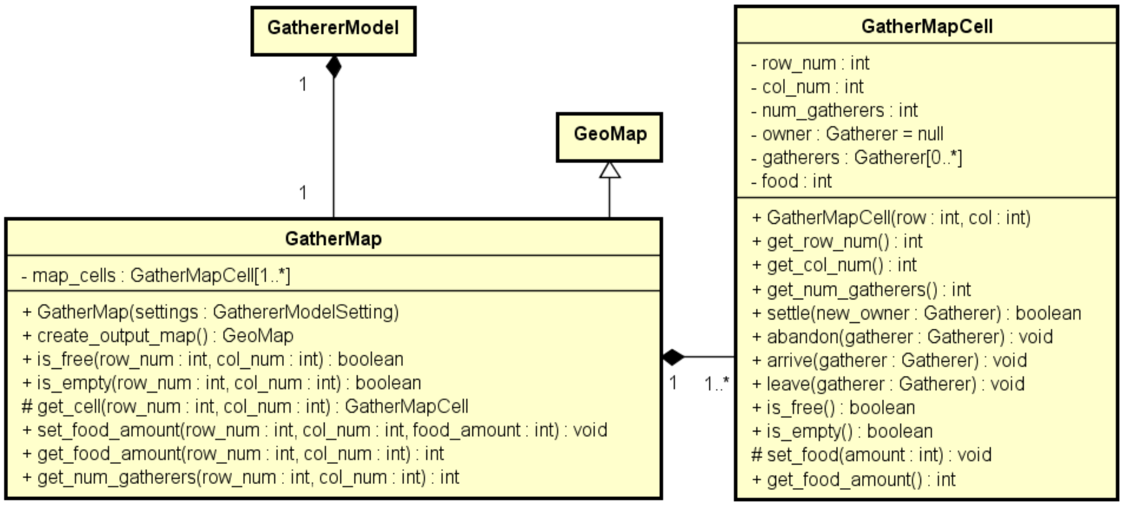

GathererModel
1
1
GeoMap
GatherMap
- map_cells : GatherMapCell[1..*]
+ GatherMap(settings : GathererModelSetting)
+ create_output_map() : GeoMap
+ is_free(row_num : int, col_num : int) : boolean
+ is_empty(row_num : int, col_num : int) : boolean
# get_cell(row_num : int, col_num : int) : GatherMapCell
+ set_food_amount(row_num : int, col_num : int, food_amount : int) : void
+ get_food_amount(row_num : int, col_num : int) : int
+ get_num_gatherers(row_num : int, col_num : int) : int
1
1..*
GatherMapCell
- row_num : int
- col_num : int
- num_gatherers : int
- owner : Gatherer = null
- gatherers : Gatherer[0..*]
- food : int
+ GatherMapCell(row : int, col : int)
+ get_row_num() : int
+ get_col_num() : int
+ get_num_gatherers() : int
+ settle(new_owner : Gatherer) : boolean
+ abandon(gatherer : Gatherer) : void
+ arrive(gatherer : Gatherer) : void
+ leave(gatherer : Gatherer) : void
+ is_free() : boolean
+ is_empty() : boolean
# set_food(amount : int) : void
+ get_food_amount() : int

Figure 21: Class Diagram of the State Map for the Gatherer Model.

An object of the Gatherer class (See Figure 22) is responsible for maintaining the state and parameters for a gatherer during its lifetime. The settings are copied from the model settings, rather than accessed during the gatherer's lifetime, to allow variation in gatherer settings for other variations of this model.

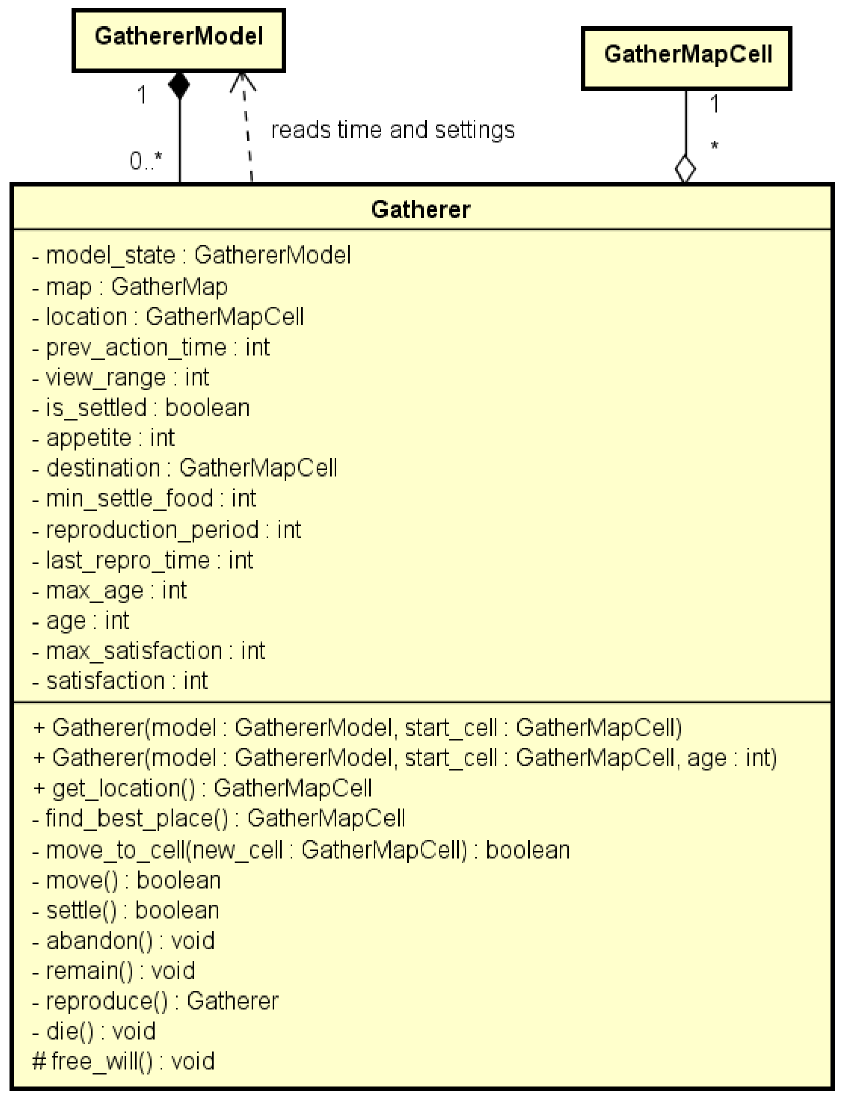


Figure 22: Diagram of the Gatherer Class for the Gatherer Model.

The free_will() method for each of the gatherers in a simulation is called during every time step. The method advances the state of the gatherer, then determines what the gatherer chooses to do during the time step. Pseudocode for the method is given below. All values refer to the conditions at the gatherer's current location.

```
free_will():
    age = age + $\Delta t$
    if age > max_age
        die
        return
    if food < appetite / num_gatherers
        satisfaction = satisfaction – 1
        if satisfaction $\leq 0$
            die
            return
    else
        satisfaction = satisfaction + 1
    if settled
        if food $\geq$ min_settle_food
            remain        // Stay at this map cell.
        else
            abandon       // Give up the claim to this map cell.
    else        // If the gatherer is not settled.
        Move to destination
        Set destination to best available visible map cell (with the most
            food)
        if current location is not settled and has the most food
            settle
```

4.8 Gatherer Simulation Setup and Use

The gatherer model includes settings that can be changed for different simulations or situations. These settings include map boundaries and resolutions, length of a time step, and the number of gatherers at the start of the simulation. For the gatherers, settings can be changed for the appetite, life expectancy, reproduction period, visual range, maximum satisfaction, and minimum fruit required at a map cell before a gatherer will settle there.

In the Java implementation of this model, an object of a separate class, named GathererModelSetting, hold the settings for a simulation. An object of GathererModelSetting can be created by using a public constructor.

```
public GathererModelSetting(
    int num_rows, int num_cols,
    int north_bound, int south_bound, int west_bound, int east_bound,
    int num_gatherers, int appetite, int reproduction_period, int max_age,
    int min_settle_food, int view_range, int max_satisfaction,
    int time_increment, int end_time, GrassSetup grass_mapset
)
```

The map region, or set of boundaries and spatial resolutions, can be set using a constructor of the MapRegion class.

public MapRegion(int rows, int cols, int north, int south, int west, int east)

An object representation for GRASS mapset support can be created by a constructor of the GrassSetup class. The gisdbase must be the absolute path of a file directory. The location and mapset are a currently existing GRASS location and mapset within the gisdbase directory.

public GrassSetup(String gisdbase, String location, String mapset)

The GrassSetup object can be used within the simulation to create or import information from GRASS rasters, containing spatial data over the simulation area. Once all settings have been established for the simulation, the simulation itself can be instantiated using the settings.

public GrowthModel(GrowthModelSettings settings)

Once the simulation is created, input maps for the amount of food can be given using the give_object_food_map() method to give a GeoMap object, or using the give_grass_food_map() method to import a map from GRASS. The associated time step must be specified with the given input, and the time step should correspond with the next time step that will be simulated. The advance_time() method runs one time step of the simulation. An output map of the number of gatherers at each map cell can be obtained as a GeoMap object with the use of the get_object_gatherers_map() method. The same output map can be sent directly to the simulation's GRASS mapset by using either the create_grass_gatherers_map() method. The current simulation time can be obtained by calling the get_time() method. Interfaces containing these methods are shown in Figure 23.

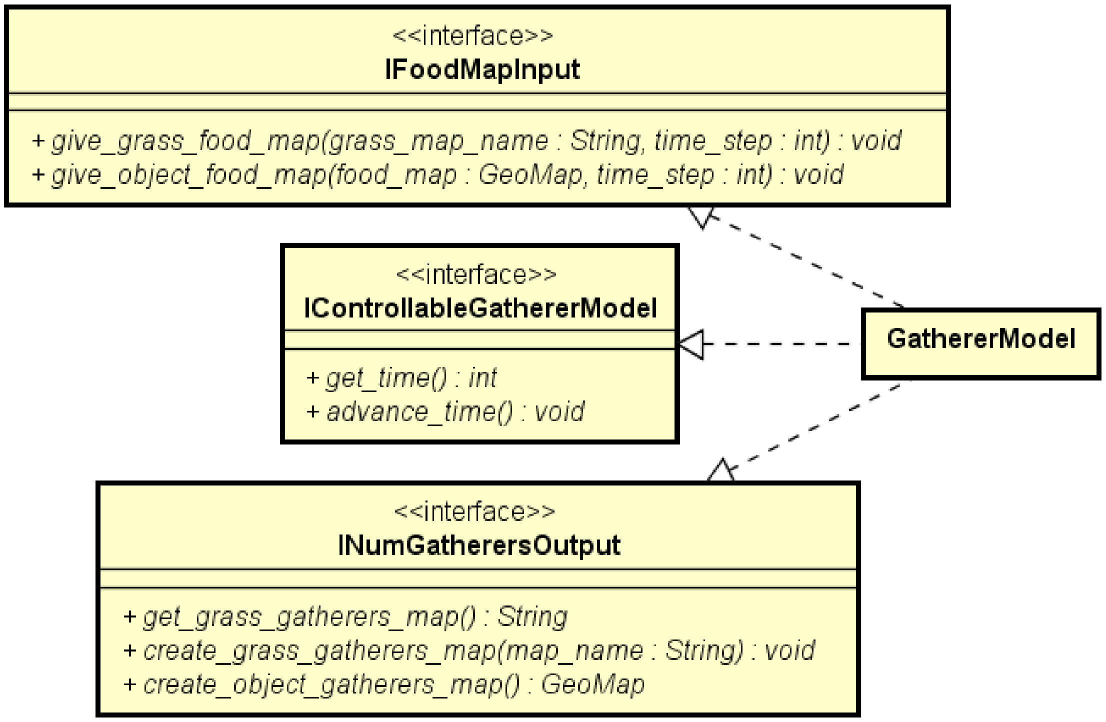

<<interface>>
IFoodMapInput
+ give_grass_food_map(grass_map_name : String, time_step : int) : void
+ give_object_food_map(food_map : GeoMap, time_step : int) : void
<<interface>>
IControllableGathererModel
+ get_time() : int
+ advance_time() : void
GathererModel
<<interface>>
INumGatherersOutput
+ get_grass_gatherers_map() : String
+ create_grass_gatherers_map(map_name : String) : void
+ create_object_gatherers_map() : GeoMap

Figure 23: Interfaces Implemented by the GathererModel Class.

CHAPTER 5

MODEL INTEGRATION

Creating a composition of models involves defining the characteristics of the composition, setting up the component models and interaction models, configuring the connections, and synchronizing the execution times. The fruit growth and the gatherer exemplar models were created for the purpose of verifying that all of these steps could be performed successfully using the interaction model created for this project. Additionally, the GRASS GIS is used to show the output maps produced during simulations. While the use of GRASS is not required for state transition or model transfer in this composed simulation, GRASS allows for complex geographic computations, as well as graphical displays of geographic data.

Composition of the fruit growth model and the gatherer model requires that the interaction specifications be defined. The gatherer model requires an input of the amount of food available at each location of the map. This input is specified to be the same as the number of fruit at the same location, taken from the fruit growth model. The input to the fruit growth model is the change, at each location, in the number of fruit due to external influences. This is determined by multiplying the number of gatherers at the location by the rate of consumption by -1. In the composition, it is assumed that different regions and time steps are used by the different models, so these differences must be handled in addition to the specified interactions.

As the exemplar models were defined in separate classes that allow for external control, the models can be instantiated and set up together in a function external to both models.

The different regions for the models were set using the constructor for MapRegion. The constructor for the fruit growth model, GrowthModel, also allows the other configurations for the model to be set, including the growth factor, radius of influence, maximum fruit per area, and length of the time steps. Settings for the gatherer model can be set in the constructor for GathererModel, including the amount eaten per time period, initial number of gatherers, maximum age, time required to reproduce, maximum vision and travel distance, and length of the time steps.

For the components that deal with interactions between the fruit growth model and gatherer model, extensions of the GeoKibFunction class need to be made. The class FoodKibFunction is a subclass of GeoKibFunction that takes a map of the number of fruit as input and produces a map of the amount of available food as output. Since these amounts are equal, no measurement conversions need to be made within FoodKibFunction, but differences in region and time scaling must still be addressed. The fruit growth model outputs the current number of fruit, so there is no case in which multiple inputs should be combined by addition or any other operations. Therefore, if multiple input maps are received before an output map needs to be sent, the output map is processed from only the most recently received input. The input map is processed through the spatial_conversion() method to adapt the input values to output region. If multiple outputs are needed before new inputs are received, the resulting map is output multiple times.

ConsumptionKibFunction is a class designed to handle the delivery of maps to the fruit growth model that tell the net change in the number of fruit caused by external influences. This class inputs the number gatherers over the map from the gatherer model. Gatherers that remain in an area for a larger amount of time will consume more fruit from the area, so input maps are added over their areas using the GeoKibFunction method map_sum() if received before an output must be created. The measurement_conversion() method of ConsumptionKibFunction multiplies each of the map's values by the amount of food eaten by a gatherer per simulation cycle, and by -1 (because the fruit change is negative). Using these methods, the state_transition() method sums the inputs, performs the measurement_conversion(), then uses spatial_conversion() to make adjustments to the region. The resulting map is given to the output port. If no inputs are received when an output is requested, all the values for the output map are 0.

Although not necessary for this particular composition, all the different types of port objects were used to demonstrate their uses. A DEKIBInput and ActiveKIBOutput were attached to the FoodKibFunction, and A DTKIBInput and PassiveKIBOutput were attached to the ConsumptionKibFunction. The ActiveKIBOutput and DTKIBInput each needed to be connected to the GathererModel class to work correctly. This required the implementation of the interfaces IInput and IOutput, the dependencies of ActiveKIBOutput and DTKIBInput, respectively. A subclass of GathererModel that implements the interfaces was created specifically for this purpose. Another possibility would have been to create a separate class that would implement the interfaces for use with GathererModel, but this would result in an unnecessary additional object during the program execution. The simulation manager for the composed model needed to explicitly give values from the GrowthModel to the DEKIBInput, and from the PassiveKIBOutput to the GrowthModel.

Exchanges of values to the GathererModel were handled automatically by the port objects of the GeoKibFunctions.

A class for the simulation clock of the composition, named TimeTracker, was created for the composed simulation. Both GeoKibFunction objects, as well as the composed model, use the TimeTracker object to determine the current time and synchronize operations. The clock was attached to the GeoKibFunction objects using their attach_sim_clock() methods. References to both GeoKibFunction objects were also given to the TimeTracker, allowing it to automatically update the GeoKibFunction objects when the simulation time changes.

During a simulation of the composed models, the simulation manager must determine when each of the component simulations must be progressed. Each component simulation can be queried for the time step for which it was last run. Information about the time of the last step and length of the time steps can be used to determine the time to run the next step for each component. The next time step for the composition is the least of the next times for the components. At each time step, the simulation manager finds the components whose next scheduled time step is equal to the current time, and commands those components to progress for one time step. It is important to note that this synchronization protocol is only possible for components that can be externally controlled to progress one step at a time.

The use of GRASS within the simulation was achieved through a few connections that allowed GRASS operations to be called within a Java environment. A set of scripts, written in Python, are used to access GRASS commands using the GRASS API. The GRASS scripts are executed by using statements for the Windows command line. Functions in Java create these statements that direct GRASS to perform specific operations, such as adding a map to a GRASS mapset or performing a specific computation over a set of maps. GRASS allows visual representations of the geographically distributed values to be displayed (See Figure 24).

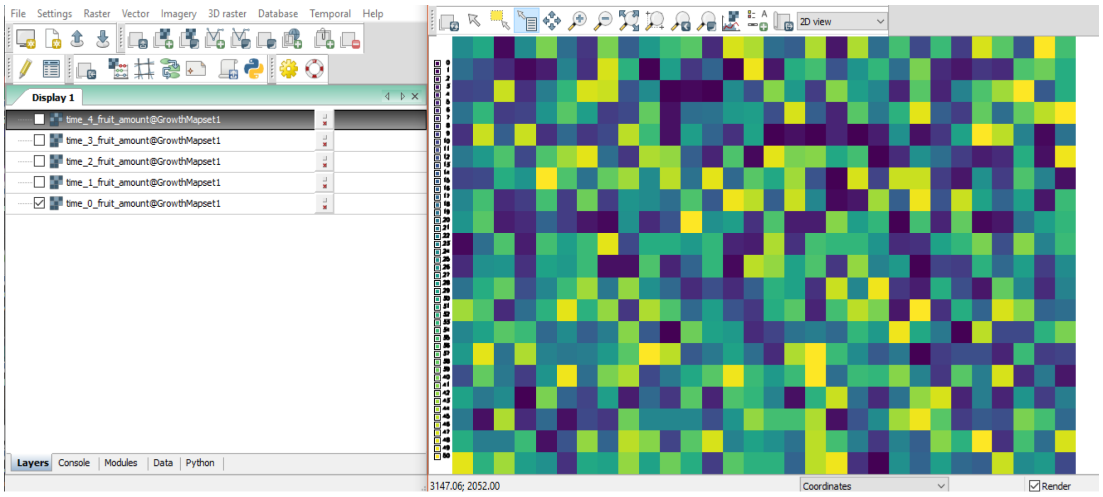
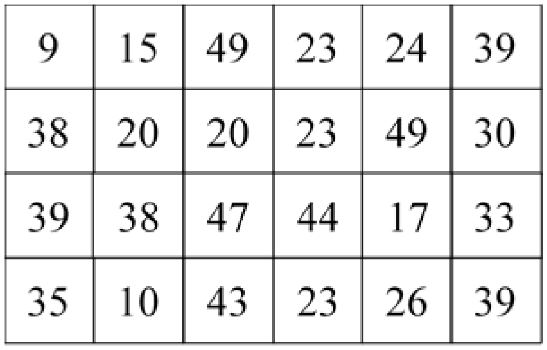

File Settings Raster Vector Imagery 3D raster Database Temporal Help
Display 1
time_4_fruit_amount@GrowthMapset1
time_3_fruit_amount@GrowthMapset1
time_2_fruit_amount@GrowthMapset1
time_1_fruit_amount@GrowthMapset1
time_0_fruit_amount@GrowthMapset1
Layers Console Modules Data Python
2D view
3147.06; 2052.00
Coordinates
Render

Figure 24: Example of a Display Produced by the GRASS GIS Software.

CHAPTER 6

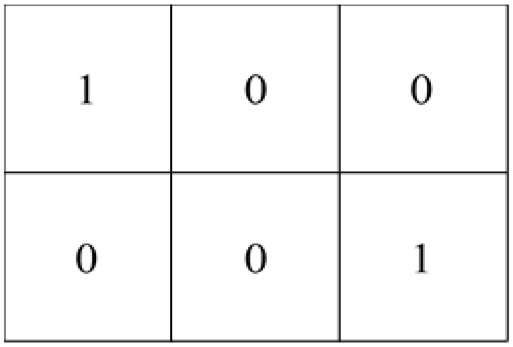
DEMONSTRATION OF INTERACTION MODEL EXECUTION

A demonstration is shown here of the interactions between the exemplar models via the GeoKIB. For this scenario, the growth model and the gatherer model are run in a joint simulation. The regions for the two models have same outer boundaries, but different resolutions. The simulation takes place in a 30km x 20km area, with west and east boundaries labeled 300 and 330, and south and north boundaries labeled 200 and 220. The simulation area for the fruit growth model is divided into 24 square cells that each have size 5 x 5. The same area in the gatherer model is divided into 6 square cells of size 10 x 10. Each gatherer eats, when available, four fruit per time step. A gatherer can see cells up to two spaces away and can move two spaces within a time step. The initial states of the models are shown in Figure 25.



9	15	49	23	24	39
38	20	20	23	49	30
39	38	47	44	17	33
35	10	43	23	26	39

(a) Initial numbers of fruit in the fruit growth model.



1	0	0
0	0	1

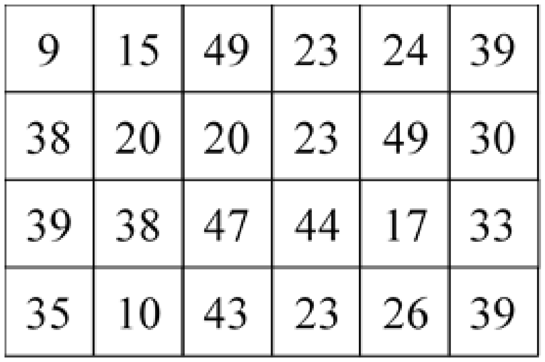
(b) Initial numbers of gatherers in the gatherer model.

Figure 25: Initial Values for the Combined Simulation.

The state of the combined simulation is updated at integral time steps. In this simulation, the state of the gatherers is updated every time cycle, but the state of the fruit growth is only updated for even numbered time steps. Time 0 is the start time of the simulation, and changes to the state occur after, not during, time 0. To avoid cyclic data dependencies, data from a component model can only be used by the other component model at a time later than the time of data production. That is, the number of fruit at time 2 influences the gatherer model at times 3 and 4, but not at time 2. This requirement allows the results of the simulation to be independent of the order of execution of the component models.

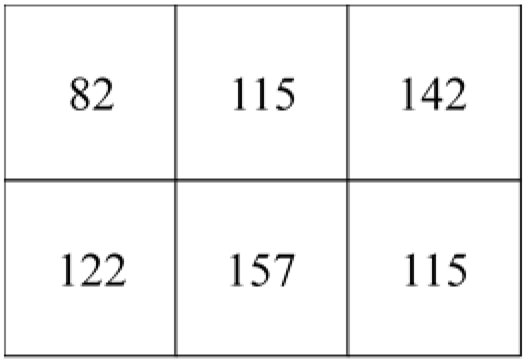
At the beginning of the simulation (time 0), the KIB takes the map information about the number of fruit from the fruit growth model. The Food KIB Function uses the data to create, as outputs, maps of available food for the gatherer model for times 1 and 2. The KIB must convert the spatial resolution to a format appropriate for the gatherer model. The food is cumulative data, so the values in the cells of the output maps are obtained from sums, not averages, from the corresponding input areas, as shown in Figure 26. Since the fruit growth model is not updated before time 2, identical maps are produced by the Food KIB Function for times 1 and 2.

As the fruit growth simulation has a period of 2 time units, its first iteration of execution is at time 2. The Consumption KIB Function is responsible for providing data to the fruit growth model concerning the changes in the number of fruit due to external influences, including the amount eaten by the gatherers.



9	15	49	23	24	39
38	20	20	23	49	30
39	38	47	44	17	33
35	10	43	23	26	39

(a) Initial numbers of fruit from the fruit growth model at time 0.



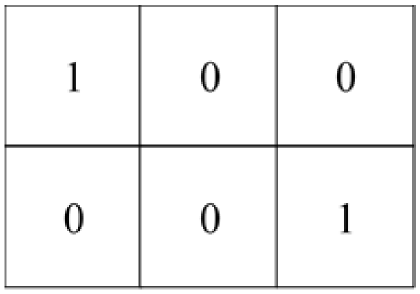
82	115	142
122	157	115

(b) Food maps created by the Food KIB Function for times 1 and 2.

Figure 26: Conversion of Spatial Resolution by the Food KIB Function.

Also at time 0, the KIB receives a map that tells the locations of the gatherers. The Consumption KIB Function produces maps that tell external changes to the number of fruit at each map cell, where negative values indicate that fruit was removed. The Consumption KIB Function has a discrete-time input port (DTKIBInput) connected to the gatherer simulation. The output port of the Consumption KIB Function is a passive output port (PassiveKIBOutput) connected to the fruit growth simulation. Since the Consumption KIB Function receives values more frequently than it sends values, the Consumption KIB Function needs to collect multiple inputs maps and assemble them before each of its values is sent.

At time 1, the first time step of the gatherer simulation is executed. During this step, the two gatherers examine their surroundings to find the cell with the largest amount of food. The cell with 157 food has more food than any other cell, and it is visible to both gatherers. The cell becomes the target of both gatherers. They each decide to move to the cell during the following time step. Because they do not move during time 1, the output from the gatherer simulation (shown in Figure 27) is the same as the output at time 0.



1	0	0
0	0	1

Figure 27: Numbers of Gatherers Output by the Gatherer Simulation at Time 1.

The Consumption KIB takes this map of numbers of gatherers during time step 1. The Consumption KIB Function has the inputs it needs to create its output, a map of the changes in fruit amounts, for use by the fruit growth simulation at time 2. The Consumption KIB Function must complete a few steps to create each of its outputs.

1. Collect and store each output from the gatherer simulation. Each output is a map of the numbers of gatherers, using the spatial resolution of the gatherer simulation.
2. Recognize when the appropriate time has arrived to create an output map. The outputs must be created after all relevant inputs (See Figure 28) have been collected, and before the fruit growth simulation runs for its next time step. The first output of the Consumption KIB Function in this example occurs at the beginning of Time 2, before the fruit growth model has run for its first time step.

2. .

1	0	0
0	0	1

(a) Numbers of gatherers at time 0.

1	0	0
0	0	1

(b) Numbers of gatherers at time 1.

Figure 28: Input Maps Processed by the Consumption KIB Function at the Start of Time 2.

3. Recognize when the appropriate time has arrived to create an output map. The outputs must be created after all relevant inputs have been collected, and before the fruit growth simulation runs for its next time step. The first output of the Consumption KIB Function in this example occurs at the beginning of Time 2, before the fruit growth model has run for its first time step.
4. Combine input maps received between outputs by adding the values at corresponding locations over each input map. The summation gives the total number of gatherer visits per location, in the gatherer simulation's spatial resolution.
5. Create a map of fruit change by multiplying the gatherer visits by the gatherers' appetite (fruit eaten per time step), multiplied by -1. The map indicates the changes in the amounts of fruit between updates in the fruit growth simulation (shown in

Figure 29). The values are all negative to indicate that the fruit was removed by the gatherers.

2	0	0
0	0	2

(a) Sum of input gatherer maps for times 0 and 1.

-8	0	0
0	0	-8

(b) Change in fruit at time 2, using the spatial resolution of the gatherer simulation.

Figure 29: Creation of the Fruit Change Map by the Consumption KIB Function at the Start of Time 2.

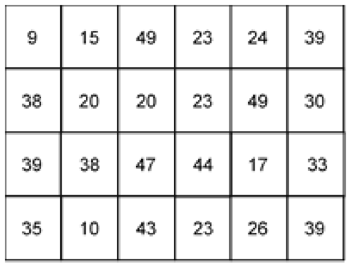
6. Perform a spatial resolution conversion on the fruit change map to convert to change its region from that of the gatherer simulation to that of the fruit growth simulation (See Figure 30).

-2	-2	0	0	0	0
-2	-2	0	0	0	0
0	0	0	0	-2	-2
0	0	0	0	-2	-2

Figure 30: Change in Fruit at Time 2, Converted to the Spatial Resolution of the Fruit Growth Simulation.

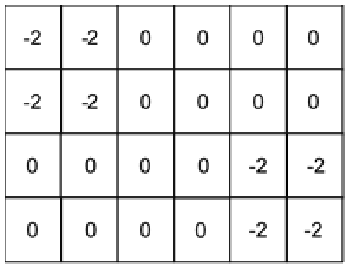
7. Make this output map, of the change in fruit, available at the output port of the Consumption KIB Function. This output is ready for use by the fruit growth simulation.

At time 2, the fruit growth simulation can proceed to execute its first simulation step, once it receives as input the map of changes in fruit from external influences. During this step, the values from the fruit change map are added, and the number of fruit is then updated based on an exponential growth model. Within the model, the growth in the number of fruit at each cell is influenced by the number of fruit in surrounding cells. The resulting amounts from the fruit growth are shown in Figure 31.



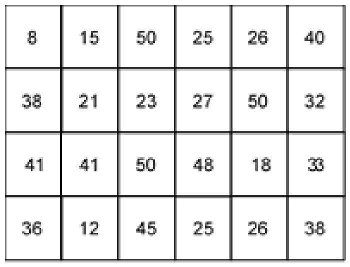
9	15	49	23	24	39
38	20	20	23	49	30
39	38	47	44	17	33
35	10	43	23	26	39

(a) Numbers of fruit at time 0.



-2	-2	0	0	0	0
-2	-2	0	0	0	0
0	0	0	0	-2	-2
0	0	0	0	-2	-2

(b) Change in fruit at time 2 (from external influences).

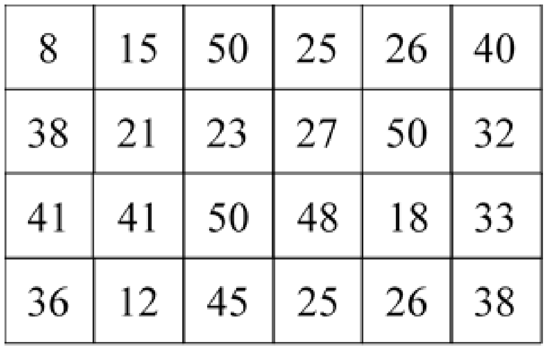


8	15	50	25	26	40
38	21	23	27	50	32
41	41	50	48	18	33
36	12	45	25	26	38

(c) Numbers of fruit at time 2, after the fruit growth simulation time step.

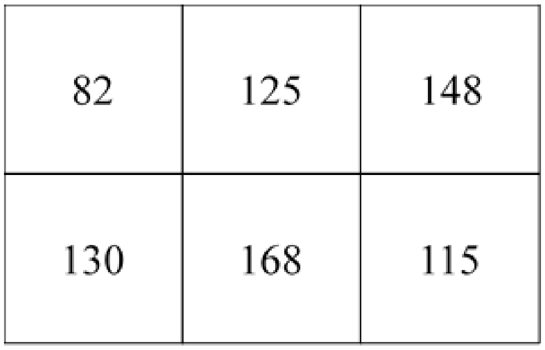
Figure 31: Time Step of the Fruit Growth Simulation.

The map of numbers of fruit is given as input to the Food KIB Function, which will use the input to create food maps for the times 3 and 4 (See Figure 32).



8	15	50	25	26	40
38	21	23	27	50	32
41	41	50	48	18	33
36	12	45	25	26	38

(a) Numbers of fruit from the fruit growth model at time 2.



82	125	148
130	168	115

(b) Food maps created by the Food KIB Function for times 3 and 4.

Figure 32: Conversion of Spatial Resolution by the Food KIB Function.

With this process, the fruit growth model and gatherer model interact without changes to their individual specifications. With similar procedures, the KIB can run combined simulations when the fruit growth simulation runs more frequently than the gatherer simulation, or when the spatial resolution of the gatherer simulation is higher than that of the fruit growth simulation. The exemplar models are not concerned with the differences in resolution or types of values from other models; each model sees inputs and outputs in its own resolution.

CHAPTER 7

TESTING AND RESULTS

The principles of the Geographic Knowledge Interchange Broker (GeoKIB) were applied to connect the two exemplar models: the fruit growth and gatherer models. In order to verify the GeoKIB's ability to compose models at different resolutions, a set of different simulations using the models was run. The different simulations differ in the relative spatial and temporal resolutions of the two exemplar models. Distances within simulations represent meters. All simulations of the set use the same set of spatial boundaries, with south boundary at a value of 1000, north boundary at 1320, west boundary at 5000, and east boundary at 5640. Each simulation time step represents one day. Within the simulations of the fruit growth model, the number of fruit at each cell increases by an amount determined by multiplying the number of fruit in the surrounding 5 meters by 0.010. A maximum fruit density of 0.3 fruit per square meter is allowed. Gatherers can live for a maximum of 30 days, and the 1000 initial gatherers have an age of 10 days. A gatherer can reproduce and create another gatherer after staying settled for 5 days. Each of the gatherers needs to consume 4 fruit per day. A gatherer will die if there is not enough available food for 3 days.

Initial values were provided to the exemplar models as maps that indicate the locations of fruit and gatherers over the simulated landscape. To remove stochasticity from the simulations, the same map of fruit locations and the same map of gatherer locations were used to create the initial state for each of the simulations. In cases where a model had a different spatial resolution than the map of initial locations, the locations were converted to create a state most consistent with the given initial locations using the model's resolution.

The first of these simulations has both models use the same spatial and temporal resolutions. Both models use a map resolution of 32 rows and 64 columns, for a total of 2,048 cells. The models each update once per simulation time step during the 100 total time steps. For this simulation, the GeoKIB's ability to manage resolution differences is not necessary, but its ability to translate the differences in the meanings of values is still utilized, specifically to convert numbers of gatherers to amounts of fruit eaten. The numbers of fruit and gatherers from this simulation are shown in Figure 33.

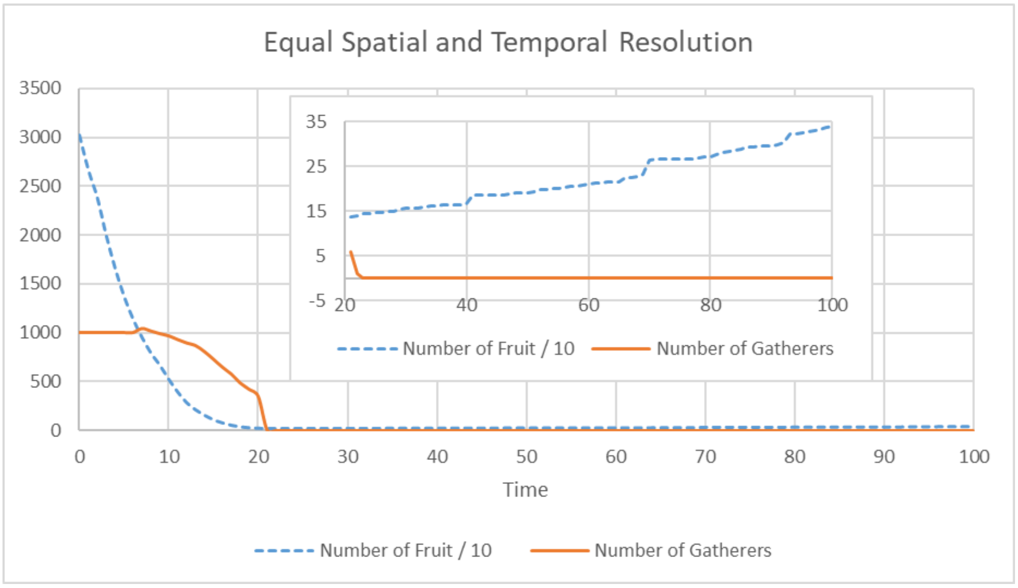


Figure 33: Results of a Simulation with Equal Spatial and Temporal Resolutions for the Exemplar Models.

Other simulations had varying differences in spatial resolution between the exemplar models, while keeping the temporal resolutions equal. In these simulations, one model had the higher resolution with 32 rows and 64 columns, while the other model represented the same area with fewer cells. Configurations for these simulations are shown in Table 1.

Fruit Number of Cells/Gatherer Number of Cells Ratio	64:1	16:1	4:1	1:1	1:4	1:16	1:64
Fruit Number of Rows	32	32	32	32	16	8	4
Fruit Number of Columns	64	64	64	64	32	16	8
Gatherer Number of Rows	4	8	16	32	32	32	32
Gatherer Number of Columns	8	16	32	64	64	64	64

Table 1: Configurations of Simulations with Different Relative Spatial Resolutions.

A few simulations retained the same spatial resolutions for the two models while using different time steps between updates for each of the models. In these, the state of one of the models was updated during each simulated day, while the other was updated only after a given number of days had passed. Configurations for simulations with varied temporal resolutions are shown in Table 2.

Fruit Number of Time Steps/Gatherer Number of Time Steps Ratio	10:1	5:1	2:1	1:2	1:5	1:10
Fruit Time Step Size	1	1	1	2	5	10
Gatherer Time Step Size	10	5	2	1	1	1

Table 2: Configurations of Simulations with Different Relative Temporal Resolutions.

Finally, some simulations were run in which both spatial and temporal differences existed between the two exemplar models. These simulations included all combinations with spatial area resolution factors of 4 or 16 and temporal resolution factors of 2, as shown in Table 3.

Fruit Number of Cells/Gatherer Number of Cells Ratio	4:1	4:1	16:1	16:1	1:4	1:4	1:16	1:16
Fruit Number of Time Steps/Gatherer Number of Time Steps Ratio	2:1	1:2	2:1	1:2	2:1	1:2	2:1	1:2
Fruit Number of Rows	32	32	32	32	16	16	8	8
Fruit Number of Columns	64	64	64	64	32	32	16	16
Gatherer Number of Rows	16	16	8	8	32	32	32	32
Gatherer Number of Columns	32	32	16	16	64	64	64	64
Fruit Period	1	2	1	2	1	2	1	2
Gatherer Period	2	1	2	1	2	1	2	1

Table 3: Configurations for Simulations with Differences in Both Spatial and Temporal Resolutions.

Because the GeoKIB was designed to be flexible regarding the differences in spatial and temporal resolutions, all of these simulations could be run without changing configurations for the GeoKIB for each simulation. The GeoKIB was able to read the region information, including boundaries and cell sizes) from the connected models to determine which map cells of the different regions are associated. The GeoKIB needed inputs that indicated the sizes of the time steps for each of the models, and the GeoKIB automatically determined how to manage the differences in data input and output timings.

In all of the simulations, the large numbers of gatherers, along with the limited number of fruit, resulted in a limit in the growth of the populations. At time 20, the original generation of gatherers die, leaving only those "born" during the simulation. As gatherers only reproduce after settling, and only one gatherer can settle in one cell, the number of surviving gatherers after time 20 is small when compared with the initial population size. Additionally, the gatherers quickly consume all of the fruit available wherever they collect.

Once the fruit is gone from an area, the gatherers of the area die if they cannot see another source of food.

Results of interactions between gatherers and fruit are apparent at all resolutions. The number of fruit decreases over time wherever gatherers are present, and gatherers tend to collect in areas with large numbers of fruit. As all of the interactions are managed by the GeoKIB, this shows the success of the GeoKIB in exchanging information between the models. Figure 34 shows the effects of changing spatial resolutions on the number of gatherers.

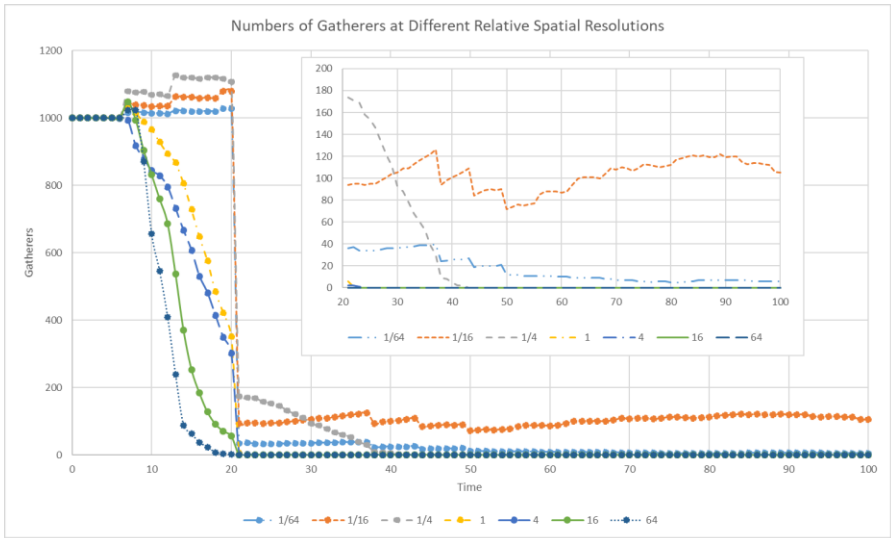


Figure 34: Number of Gatherers During Simulations with Different Spatial Resolutions for the Gatherer and Fruit Growth Models. Both the gatherer and fruit growth models have temporal resolution of 1 update per day. The key shows the number of gatherer cells divided by the number of fruit growth cells.

The differences in relative spatial resolutions resulted in some differences in results. In particular, the behavior of the gatherers was especially sensitive to spatial resolution. At lower relative resolutions of the gatherer model, each map cell of the gatherer model corresponds to multiple cells of the fruit growth model. This results in more fruit available at each of the gatherers' cells, making each of the cells more sustainable. However, the lower number of cells for the gatherers also results in fewer locations to settle and faster depletion of food. The gatherers had largest populations when each gatherer cell encompassed 16 cells of the fruit growth model.

A limitation of this approach to multi-resolution modeling became apparent. When models rely on other models for data, differences in resolution can result in missing or insufficient information for one of the models. For example, differences in temporal resolution force one of the models to rely on outdated data from the other model, with the GeoKIB forced to extrapolate some results to the current time step. When the gatherer model was updated 5 times as often as the fruit growth model, the GeoKIB needed to supply information to the gatherer model from as much as 5 days previous to the current day. Then, after the fruit growth model was updated, it was impacted by all 5 days of influences from the gatherers at once.

The fruit growth model and gatherer models were created only as samples for use in developing and testing the GeoKIB. A study of predator-prey models, population models, or vegetation growth models would result in more detailed models. Such models could also be composed using the same techniques, depending on the external inputs required by each model and the data produced within each model.

CHAPTER 8

CONCLUSION

One or more interaction models can be used to represent and regulate the interactions between models with different configurations. When models communicate via transfer of geographic data, the data can be transformed to account for differences in boundaries and resolution, as well as the differences in measurements and timing protocols. Although there is no one model that is appropriate for the representation of all types of interactions, the use of inheritance has proven useful in allowing the reuse of operations for a variety of purposes. In particular, algorithms that transform values, whether cumulative or average values, to different geographic regions can be applied to different applications with little or no alteration.

While the interaction classes can be extended and applied to other applications, the designs could be expanded in future projects. The GeoKibFunction class of this project is designed to work with integer values, but configuration options could be added that allow different data types for a geographic map. The KIB units have shown to operate well for single-threaded systems, but additional work is needed to ensure that model components and KIB units could function correctly in multithreaded systems while maintaining data integrity and preventing and resolving deadlocks. Constraints and aesthetic improvements can be added to aid software and model designers who compose models using the GeoKibFunction class. Connections to other GIS systems can be established, similar to the connections that allowed the use of GRASS for this project. Such changes would facilitate the application of the principles of a knowledge interchange broker in regulating the interactions between many different models that have geographic components.

Cellular Automata model composability supported with the geo-referenced Knowledge Interchange Broker lends itself to be realized in component-based modeling and simulation tools such as DEVS-Suite simulator. Geographic-based component simulation modeling is an approach that provides greater modularity for application domains involving multiple levels of spatiotemporal abstractions.

REFERENCES

[1] H. S. Sarjoughian, "Model Composability," in Proceedings of the 38th Conference on Winter Simulation, 2006.

[2] S. Wolfram, "Statistical mechanics of cellular automata," Reviews of modern physics, vol. 55, no. 3, pp. 600-642, 1983.

[3] C. M. Barton, I. Ullah and A. Heimsath, "How to make a barranco: Modeling erosion and land-use in Mediterranean landscapes," Land, vol. 4, no. 3, pp. 578-606, 2015.

[4] M. Neteler and H. Mitasova, Open source GIS: a GRASS GIS approach, New York: Springer Science & Business Media, 2013.

[5] H. Sarjoughian, W. Boyd and M. Acevedo, "Challenges of Achieving Efficient Simulations Through Model Abstraction," arXiv.org, 2017.

[6] M. Acevedo, W. Boyd, H. Sarjoughian, I. Ullah and C. Barton, "Integrating Socio-Environmental Models: Vegetation, Agriculture, and Landform Dynamics," in 9th International Conference on Environmental Modeling and Software, Fort Collins, Colorado, U.S.A., 2018.

[7] G. Mayer and H. Sarjoughian, "Composable cellular automata," Simulation, vol. 85, no. 11-12, p. 735–749, 2009.

[8] G. R. Mayer and H. S. Sarjoughian, "A Composable Discrete-Time Cellular Automaton Formalism," in Workshop on Social Computing, Behavioral Modeling, and Prediction., Phoenix, AZ, USA, 2008.

[9] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan and G. Balan, "Mason: A multiagent simulation environment," Simulation, vol. 81, no. 7, pp. 517-527, 2005.

[10] S. 1. Wolfram, The Mathematica Book, Cambridge: Wolfram Median, 1996.

[11] D. J. Peuquet and N. Duan, "An event-based spatiotemporal data model (ESTDM) for temporal analysis of geographical data," International journal of geographical information systems, vol. 9, no. 1, pp. 7-24, 1995.

[12] B. P. Zeigler, H. Praehofer and T. G. Kim, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd ed., San Diego: Academic press, 2000.

[13] G. A. Wainer and N. Giambiasi, "Application of the Cell-DEVS Paradigm for Cell Spaces Modelling and Simulation," SIMULATION, vol. 76, no. 1, pp. 22-39, 2001.

[14] C. Zhang and H. S. Sarjoughian, "Cellular Automata DEVS: A Modeling, Simulation, and Visualization Environment," in Proceedings of the 10th EAI International Conference on Simulation Tools and Techniques, Hong Kong, China, 2017.

[15] ACIMS, "DEVS-Suite Simulator," Arizona State University, [Online]. Available: https://acims.asu.edu/software/devs-suite/. [Accessed 10 11 2019].

[16] G. R. Mayer and H. S. Sarjoughian, "Building a hybrid DEVS and GRASS model using a composable cellular automaton," International Journal of Modeling, Simulation, and Scientific Computing, vol. 7, no. 1, pp. 1-31, 2016.

[17] S. Inokuchi, T. Ito, M. Fujio and Y. Mizoguchi, "A formulation of composition for cellular automata on groups," IEICE TRANSACTIONS on Information and Systems, vol. 97, no. 3, pp. 448-454, 2014.

[18] P. K. Davis and J. H. Bigelow, "Experiments in Multiresolution Modeling (MRM)," RAND, Santa Monica, CA, 1998.

[19] M. Gardner, "Mathematical Games - The fantastic combinations of John Conway's new solitare game "life"," Scientific American, vol. 223, no. 4, pp. 120-123, 1970.

[20] A. R. Kiester and K. Sahr, "Planar and spherical hierarchical, multi-resolution cellular automata," Computers, Environment and Urban Systems, vol. 32, no. 3, pp. 204-213, 2008.

[21] A. G. Hoekstra, J.-L. Falcone, A. Caiazzo and B. Chopard, "Multi-scale Modeling with Cellular Automata: The Complex Automata Approach," in 8th International Conference on Cellular Automata for Research and Industry, Yokohama, Japan, 2008.

[22] A. G. Hoekstra, E. Lorenz, J.-L. Falcone and B. Chopard, "Towards a Complex Automata Framework for Multi-scale Modeling: Formalism and the

Scale Separation Map," in Fifth International Conference on Computational Science and its Applications, Kuala Lumpur, Malaysia, 2007.

[23] R. Alur, Principles of Cyber-Physical Systems, Cambridge, Massachusetts: The MIT Press, 2015, pp. 13-36.

[24] G. Godding, H. Sarjoughian and K. Kempf, "Application of combined discrete-event simulation and optimization models in semiconductor enterprise manufacturing systems," in 2007 Winter Simulation Conference, Washington, DC, USA, 2007.

APPENDIX A

SETUP AND USE OF THE INTERACTION MODEL

Using GeoKibFunction

Application of the GeoFunction to a set of models can be done in a few steps. First, the GeoKibFunction must be extended to create a class designed to handle transformations between two specific models. The abstract methods, state_transition(), is_transition_triggered(), and measurement_conversion(), must be given concrete definitions.

After creating a concrete class for the interaction function, the class must be instantiated in a scope that allows access to the models it will connect. GeoFunction is dependent on a simulation clock that implements the ISimClock interface, so this should be resolved by attaching a clock object. This can be done with the method attach_sim_clock(). For this project, a class for a simulation clock was created named TimeTracker. TimeTracker automatically updates attached objects as the simulation time changes. The FoodKibFunction and TimeTracker objects were attached to each other in this way.

```
simTimer = new TimeTracker(0, deltaTGrowth, deltaTGatherer);
FoodKibFunction foodKib =
        new FoodKibFunction(growthRegion, gatherSimRegion);
foodKib.attach_sim_clock(simTimer);
simTimer.add_updatable_object(foodKib);
```

The input and output port objects for the GeoKibFunction must be instantiated separately. The number and type of port objects to use depend on the models that will be connected and the method with which values will be transferred. For input ports, the DEKIBInput class is best if inputs will be given to it, and the DTKIBInput class is best for taking inputs directly from another model. For output ports, PassiveKIBOutput allows output values to be taken at any time, and ActiveKIBOutput can be used to give values directly to another model. The GeoKibFunction object must be specified as an attribute in the constructors of the ports, attaching the ports to the GeoKibFunction object. For DTKIBInput and ActiveKIBOutput objects, other models need to be attached to allow automated communication, and the period of data transfer needs to be set.

```
DEKIBInput<GeoMap> foodKibInput = new DEKIBInput<GeoMap>(
    foodKib, "fruit_amount"
);
ActiveKIBOutput<GeoMap> foodKibOutput =
    new ActiveKIBOutput<GeoMap>(foodKib, "food_amount");
foodKibOutput.add_destination(gathererModel);
foodKibOutput.set_period(deltaTGatherer, 0);
```

Compilers and Execution Environment

The following software was used as parts of the environment under which the sample simulations and interaction software were executed.

Java 1.8.0_71

GRASS GIS 7.4.0

Python 2.7.13

Windows 10 Home

The software created in this project is expected to operate correctly when using any version of GRASS GIS 7, any version of Python 2.7, and any version of Java 8 (1.8) or later.

Software Development Tools

The following tools were used for the creation of the software and documentation of this project.

IntelliJ IDEA Community 2017.2.5

JetBrains PyCharm Community Edition 5.0.1

Astah UML 7.2.0/1ff236

Eclipse Neon.2 Release (4.6.2)

Git Version 2.8.2.windows.1

Github Internet service (https://github.com)

Microsoft Excel, Word, and PowerPoint