UNIVERSITÉ DU LUXEMBOURG

Faculty of Science, Technology and Communication

A method for coordinating unrelated space-time topologies in multi-agent, multi-scale simulations

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Master in Information and
Computer Sciences

Author:
J. Luke Scott
luke@cronworks.net

Supervisor:
Prof. Pascal Bouvry

Reviewer:
Prof. Steffen Rothkugel

Advisors:
Dr. Grégoire Danoy
Prof. Ferdinando Villa
Ass. Prof. Ioannis Athanasiadis

September 2013

Contents

Abstract

Agent Based Modeling systems are being used in increasingly diverse applications every year. The data sources and agent models used in most of these systems operate under synchronized views of space and time, avoiding the need for mediation between scales. However, mediation must be done if systems are to incorporate models whose scales are not synchronous, an issue especially pertinent for fields relying on unrelated, externally provided data sets, as is the case with geostatistical or geopolitical systems. Here we document a framework, API, and proof of concept which provides the functionality needed to mediate between scales of heterogeneous agents. Our solution and its reference platform (Thinklab) use strict semantic rules and ontology-based modeling to achieve consistency and reusability.

Chapter 1

Introduction

This project seeks to develop an efficient, high quality, general purpose solution which can represent software agents with heterogeneous perceptions of space and time, and allow them to interact properly and efficiently in simulations. Agents with such heterogeneous scales have differing views of how space and time are segmented, categorized, and observed. The representation of internal state and observed state is affected by such views, which in turn affect how these agents interact with and respond to each other. As a proof of concept, this system will be contributed as the agent subsystem of the Thinklab modeling software stack, a semantically driven, open source infrastructure used for major environmental decision support systems.

1.1 Context

Agent systems in current research fall into two categories: those whose primary purpose is *problem solving*, and those whose primary purpose is *simulation*. In both types of systems, the characteristic traits of the system emerge from the interactions of individual agents; in many cases, complex system dynamics emerge from relatively simple agent definitions.

In a problem-solving system, the quality of the emergent behavior of the system is the measure of success. The selection of individual agents and their behavior during their life cycles is unimportant, as long as their behaviors lead to an accurate or effective solution. These systems generally operate under a (job shop) scheduling paradigm: a number of tasks, sometimes with deadlines, are allocated to a collection of resources to achieve sometimes competing objectives.

One example of problem-solving systems is a trading agent system, which searches for solutions to the multiple-objective, multiple-constraint Trading Agent problem [51]. This problem is a real-world scenario in which travel agencies, supply chain variables, auctions, and other constantly varying game scenarios must be navigated within time constraints. Another example is an ant-colony system for solving the Traveling Salesman problem.

In a simulation system, individual behaviors of the agents are chosen so as to match reality, and the consequences of behavior on the higher-level system are the object of study. Because of this difference, the system is designed with different constraints. Agents are modeled to correspond to a real-world behavior, and the system must implement that behavior as it is modeled. The system cannot select different agent types to provide better performance; rather, the system must provide a performant environment within which arbitrary agents can be defined with semantic accuracy and simulated with precision.

Because our system is built for general-purpose modeling, we are tasked with the latter requirements. We are required to ensure that the agents modeled in our system are a fair reflection of the real world. As a consequence, we are also required to accept data sources and agent semantics which reflect subjective views of space and time.

A speed/quality trade-off exists, as in most systems. The trade-off in our system can be managed by the modeler by selecting higher or lower quality scale mediation strategies, more or less detailed data sources, etc. Our goal is to fulfill these constraints efficiently by implementing an efficient modeling environment, rather than selecting the optimal agent type(s) for a given task.

1.2 Motivation

Problem solving and simulation systems differ in some ways but can share design and implementation techniques. The primary difference between the two is that the measure of success of a problem solving system is the accuracy of its final result, whereas a simulation system is measured by the accuracy of the individual behaviors of the agents in the system.

All agent simulation systems known to us at the time of writing use a view of space/time which is inherently consistent (see Figure 1.1). In systems with consistent space/time, agents which operate at different scales share common dimensional units and have a lowest common denominator (which is the base unit for the system). In this way, agents can easily translate values between themselves. No complex scale mediation semantics would be needed in these systems beyond simple scaling functions.

Feature	Existing Systems	Our Requirement
Heterogeneous Scales	use common base units	unrelated units/segmentation
Agents' Location in Space	generally cell-based	arbitrary (cells, polygons, etc)
Temporal Semantics	uniform time steps	arbitrary, per-agent durations
Scale Mediation	not necessary	automated, configurable solution
Subjective Agent Perception	not explicitly modeled	API should be provided

Table 1.1: Our itemized requirements as they currently exist in other systems, and as we would require

However, we would like to loosen this constraint so that our system is able to incorporate raw data or models using arbitrary scales. Strict semantics is the guide for agent definition, and accordingly agents may have different scales with no relation to each other. The representation chosen for space and time should not be forced to agree in simultaneously represented agents. It is possible that elevation data is represented using meters over a square-kilometer sample grid, rainfall data is aggregated in inches over square miles, and software agents reason and navigate using subjective categories over polygons defined in latitude/longitude units. In this case, no single cohesive grid or measurement system exists, even though they all represent semantically compatible concepts of *earth's surface*.

In the real world, as agents (robots) interact with each other, different internal representations are negotiated by automatic, usually unconscious, translation. Differences in perception are usually not obvious to the agents. Creating a *virtual* system where these differing internal representations are honored while computing observed states for all agents, while preserving accuracy and performance, is a difficult challenge and the main purpose of this work.

In Chapter 3, we identify three main components necessary for a multiple-scale system: semantics, scale mediation, and negotiation of agent perception. The scale mediation component is the one with arguably the most serious performance implications. Many academic fields have contributed thinking to the topic of multivariate data interpolation, from which we draw influence; 2D and 3D image processing, for instance, has been invested in by both industry and academia. The programming paradigm called Functional Reactive Programming (FRP) was an inspiration in shaping our thinking about time as a unique dimension with characteristics not shared in e.g. 3-dimensional space. FRP was also a rich ground for developing computational models that deal appropriately with potentially fatal combinatorial explosions.

Other agent-based system designs are discussed in Section 2.1, which includes ontologies of agent types that exist in these systems. Other academic fields that contribute to this topic are discussed in the following sections of this chapter.

1.3 Contributions

In this thesis we describe the following contributions to the current literature:

Modular Scale Mediation Strategy: By using a modular scale mediation mechanism, we can fully decouple agents' subjective scales (views of space and time, with other dimensions allowed but not implemented). This allows software agents of arbitrary design to interact with each other and with various real-world data sets without regard to the scales used internally by the others. By standardizing our system using formal ontologies, the scale conversions can be done in context-appropriate ways without the involvement of agent developers or field researchers.

Globally-Synchronous, Locally-Subjective Time Scales: Time is treated as a unique dimension which is allowed to differ between agents as the other dimensions are. The way we implement time also enforces forward causality and synchronous observations, while allowing agents' individual perceptions and computations to be completely decoupled from each other's. This has the advantage of providing a highly parallelizable computation paradigm with explicit synchronization points.

Circular Reference Avoidance: Our time period semantics treat the time-period boundaries as exclusive-start, inclusive-end, which is opposite of the predominant inclusive-exclusive model. By doing this, we introduce: 1) an approximation of the real-world temporal delays between cause and effect, while retaining the simplicity of a temporally-synchronized simulation; and 2) an outright avoidance of circular references in simulated environments.

Efficient and Flexible Agent-State Semantics: By dictating that agents make *observations* and decisions at specific instants in time and their decisions lead to *agent-states* over the intervals bounded by these instants, we provide a computational model which is very efficient and yet provides agent-state functions which can express complex, continuously-changing values that can be observed at any instant during the agent's lifetime.

1.4 Outline

The rest of this thesis is organized as follows:

In Chapter 2 we discuss related work in literature which informs the formulation of the various problems we have addressed and the approaches we have used in our design.

Chapter 3 is a statement of the various problems which must be solved to create multiple-scale simulations. We also describe two use case scenarios which exemplify the requirements that a system should fulfill, as well as being illustrative of the domain and typical uses for which Thinklab is currently used.

In Chapter 4 we present our semantic model and system design for addressing all of the elements from our problem statement. This chapter also gives our reasoning for choosing some unorthodox techniques, namely exclusive-inclusive time periods and globally-synchronous, locally-subjective temporal scales.

Chapter 5 describes how we have implemented the modules described in our design. We also discuss our proof of concept which demonstrates the functioning of the modules.

Chapter 6 concludes with a summary of our work, as well as future work which we believe could make our solutions more flexible and applicable beyond the use cases to which we have limited ourselves.

As is the case with any project which overlaps with multiple disciplines, in this project we are forced to choose terminology which fits our purposes, at the possible expense of abusing or stretching definitions in some of the overlapping disciplines. In the glossary we list the terms which may have varying interpretations depending on the background of the reader, as well as some terms whose definitions are simplified for the purposes of this thesis. In the latter case, we often draw finer distinctions between concepts in Thinklab than we express in this thesis because some implementation details and domain-specific considerations go outside the scope of this work.

Chapter 1: Introduction

Chapter 2

Related Work

This project aims to contribute to a semantic meta-modeling system, where the primary issue is scale mediation among heterogeneous agents and data sources [57, 58, 55, 56]. To accomplish our goal, consistent semantics had to be developed. Hence, this project primarily overlaps with others in the areas of semantic agent modeling and multivariate interpolation (which we refer to as *scale mediation*). Going forward, the target infrastructure (Thinklab) will be used for more complex scenarios, and performance and flexibility issues will most likely surface such that other areas of research are expected to be beneficial.

The rest of this chapter is organized as follows:

Section 2.1 discusses the agent modeling landscape in literature. This section places Thinklab into context and motivates its existence by explaining the assumptions in other systems with reference to our wish list for an ontology-based modeling platform.

Section 2.2 describes the Functional Reactive Programming paradigm, which is a primary inspiration for our semantic model of temporal observation and the progression of agent-states through time. We have also gained insight on how Functional-Reactive designs can lead to efficient implementation techniques for complex simulated worlds.

In Section 2.3, we discuss the field of Geospatial Interpolation, which most closely resembles our task of mediating between heterogeneous agent scales in simulations. This field is useful for us on two levels: 1) it is centered around the idea of combining geostatistical data sets to form holistic understandings, which is also Thinklab's *raison d'être*; and 2) it emphasizes that not just scales but also scale mediation strategies must be heterogeneous, which is to say that a scale translation approach cannot be a one-size-fits-all proposition.

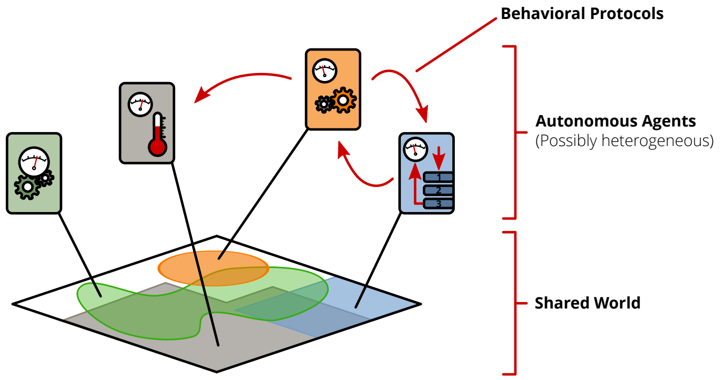


Figure 2.1: The main components of an agent modeling system

Continuing from above, we introduce Image and Signal interpolation in Section 2.4. This section is a simplification of the previous, because image and signal interpolation are done in well-controlled environments. The simplifications allow us to study idealized solutions (generally spline-based vector representation) as landmarks. We would like to approach the directness of these approaches in our own work.

Sections 2.5 and 2.6 describe related fields which we have not studied deeply but leave their application for future work.

2.1 Agent Modeling and Simulation

Thinklab is ultimately an agent modeling system, as the agent paradigm can be considered a generalization in the field of modeling that can encompass most other paradigms in use today. Here we describe Thinklab and some prominent Multi-Agent Systems (MAS) and the main ideas in MAS literature which had an impact on our work.

Figure 2.1 shows the general components of an agent modeling system. Autonomous agents of possibly heterogeneous type, structure, and internal and external behavior occupy a shared, simulated world. Their interactions are mediated by the system through well-defined protocols. The agents' individual and group behaviors, the properties of the shared world, and the system protocols lead to emergent behavior which is more complex than what can be captured by deterministic models. This property of agent modeling systems makes it an effective tool for analyzing complex systems of interaction in the natural sciences, economics, sociology, etc.

2.1.1 Thinklab Modeling Platform

Thinklab is a *semantic meta-modeling* platform which aims to address the task of *integrated modeling* as a reconciliation of strong semantics with modeling practice. The rationale behind its development is to help achieve advantages (such as modularity, flexibility, validation, and integration of multiple paradigms and multiple scales) that have remained unrealized in modeling to this day. To achieve this goal, Thinklab keeps the *logical* representation of the modeled world distinct from the *procedural* knowledge that allows its simulation. The logical representation is modeled using concepts and relationships that comprise the Thinklab *abstract knowledge base*, built around a set of ontologies that provide a solid foundation to describe *what* the aspects of interest of the modeled world are, with no attempt to resolve *how* they may be simulated.

In the procedural knowledge base, models are defined that allow users to produce observations of the concepts contained in the abstract knowledge base. Models can consist of algorithms or datasets; from the Thinklab point of view, the two just represent different ways to observe a concept. Both the abstract and the model knowledge can be expressed using a compact domain-specific language.

Semantic meta-modeling uses the idea of *observation* as the unifying theme to define a general way to model physical objects and phenomena. A model is seen as a strategy to produce observations of a concept that comes from an accepted knowledge base. Compatibility of different models as components of the same computation is guaranteed by the semantic equivalence of the concept they model, established through standard machine reasoning. When supported by adequate infrastructure, this approach enables the integration of many modeling paradigms that are often used and described separately, for example spatially-explicit to process- and agent-based models, or probabilistic vs. deterministic models. This conceptualization provides a natural path to reach goals in modeling that have frequently been discussed, but not demonstrated so far to their full potential, including modular modeling, multiple-paradigm modeling, multiple-scale modeling and structurally variable modeling.

The Thinklab software stack consists of ThinkQL (TQL), a domain-specific language, and the Thinklab server, which performs model reasoning and processing. Various clients can be used; normally the system is used with the ThinkCap Eclipse client plug-in. Thinklab aims to allow uncoordinated extensibility of the model base and to be a modular, multiple-scale and multiple-paradigm modeling infrastructure.

The objects and processes in Thinklab are defined using the OWL ontology language [35] in its OWL-DL incarnation that is guaranteed decidable. Thinklab is delivered with an initial knowledge base consisting of established pre-defined ontologies for foundational concepts and for common concepts used in physical modeling and data processing. At the moment, the DOLCE [19] ontology provides the foundational basis for all specifications, and the SWEET [41] ontologies from NASA provide an initial base of knowledge that can directly be used for model building. DOLCE and SWEET have been provisionally aligned by the Thinklab developers.

The basic function of Thinklab is to simulate observations. The semantics of the *subject of observation* (equivalent to *agents* in multi-agent systems terminology) drives the observation process: *data* are the literal properties of the object, while *object properties* point to other objects. This corresponds closely to the OWL conceptual model and maps directly to the established description logics model. A simulation is generally driven by observing one top-level *root subject*, leading to a cascading series of sub-observations which collectively generate a simulated world.

When building a model, the root subject observed and any data known before any observation takes place (*abstracts* in DOLCE parlance, including extents of space and time where the object is located) constitutes a *context* for any future observation, whose result will be constrained to satisfy all the properties required for the result to be semantically consistent. Because the data usually contain space and time as topologies (e.g. grid cells or polygons for space; linear steps for time), any data properties whose state is defined indirectly (for example through numbers or categories) will have multiple values, in number corresponding to the Cartesian product of the different states implied by the topologies adopted. The Thinklab resolution engine uses machine reasoning to choose the best modeling strategy in a given context, using the models and data available in the procedural knowledge base under the guide of the relationships stated in the abstract knowledge.

The TQL language is used both to define the knowledge base using a `model` instruction, which provides semantic annotations of data and algorithms, and to interrogate it using an `observe` instruction, which creates observed objects that are the equivalent of output datasets.

Models can be composed according to context, for instance when a single context spans multiple areas (e.g. when the context spans both land and ocean). If different models are appropriate in different areas, then these models will be selected and combined to resolve the same observable concept for an optimal overall description.

To generate results from a modeling session, the `observe` action is called on a concept or an incomplete object, and Thinklab builds a dependency graph that corresponds to the observation strategy that can produce the requested values. The graph is then compiled into a scientific workflow which corresponds to the actual computation. (Scientific workflows are well documented in literature and are beyond the scope of this thesis [2].) The workflow is compiled into bytecode and run on a virtual machine to define the output states and sub-agents that compose the final dataset delivered to the modeler.

2.1.2 Other Agent Modeling and Simulation Systems

Many other agent-based modeling and simulation platforms exist today for a wide variety of both general and specific uses. A full review of all systems is beyond the scope of this thesis. Reviews of have been compiled for general-purpose agent systems [1], systems for analyzing land use and cover change [46], climate change [5], and general environmental informatics [3]. Other special-purpose systems exist; our reviews focus on environmental systems because Thinklab is currently employed in this domain.

Special-purpose agent systems are commonly built using general-purpose platforms, and so do not provide any re-usable features beyond what is available in the underlying platforms. Occasionally they are simple, custom-built systems, but these also provide very little re-usability. Custom-built systems will normally be effective at solving only the problems they are built for, and do not provide solutions or reusable system components for general problems such as arbitrary-scale semantics and translation.

The main general-purpose agent systems are: SWARM [22, 36, 53] and its derivative systems such as Repast [42, 43], JADE and other FIPA-compliant systems [6], NetLogo [54], and MASON [31]. Cougaar [21] is also used in many research projects, but it is primarily a problem solving system rather than a simulation system. It does not contain built-in concepts of time, space, or scale.

Thinklab was designed because no existing system contains all the features desired by the research team, and none would even be well suited to customizing for our purposes because of fundamental assumptions made, or core design principles used. Typical problems which exist in other agent systems are:

- They use I/O files instead of data stores, and other fixed, non-adaptive structures.
- Agent concepts which do not align with our purposes, whether too simple (FLAME), too complex (JADE), or with architectures that do not work well for us.
- Time and space are only abstract (machine) values without inherent semantics, or are not capable of expressing non-synchronous or relative values per agent.

- They are opaque in general, not lending themselves to usability or extensibility.

Thinklab is designed from the ground up to take shortcomings from existing systems into consideration and provide a good foundation of agent simulation features, a plug-in interface allowing future expansion, and an effective path to sharing of resources, reusability and modularity aimed to serve a distributed and diverse user community.

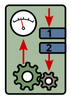
2.1.3 Ontology of Agents in Literature

This section is a catalog of the most common agent types that have been identified in literature [4, 8, 9, 44, 50]. Agents can be categorized by individual or group behavior, capability, construction, etc. We have chosen to categorize agents by the two most basic agent features: *reasoning* ability, and *cooperation* ability. The list below is ordered from the most simple to the most complex.

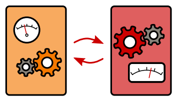
Information Carrier Agents have no reasoning ability; they exist to represent information or inanimate objects in the agent simulation environment. Athanasiadis [4] discusses the distinction between information carrier agents and *decision maker* agents, of which the remaining list is composed. In Thinklab, information carrier agents normally represent messages or collected field data (see Section 4.1.1 for more information).

Reactive Agents are the most simple type of agents capable of interacting with the world around them. They incorporate a *subsumption* architecture, using layers of reactive rules or rule sets. Rule filters are compared to inputs and/or agent state until a match is found; the actions associated with the matching rule(s) are then carried out. Layers can subsume, or override, each other based on priorities dictated at build time.

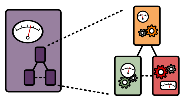
Deliberative Agents generally follow the Belief-Desire-Intention (BDI) architecture introduced by Bratman [8], which means they maintain state and can reason methodically; in general they can be considered Turing machines. *BDI* indicates these agents' semantic model: they maintain state as a set of *Beliefs*, goals and priorities as a set of *Desires*, and they attempt to achieve goals by executing *Intentions*. We focus more on agent capabilities than internal workings, so we refer to these agents simply as deliberative agents.



Hybrid (Layered Deliberative) Agents use a layered architecture to provide both reactive and deliberative functionality. A controller decides where to direct input and which of the behavior(s) to perform.



Social (Deliberative + Messaging) Agents know about and can communicate with each other. Their communication is often restricted by agent type or group boundaries. They can inherit hybrid/layered functionally if desired.



Organized (Deliberative + Messaging + Organization) Agents can communicate with and reason about each other, and can also form into social and organizational structures. These structures are well studied in literature, and lead to useful emergent behaviors, especially in problem solving or socioeconomic simulations.

In addition to this rather linear taxonomy, Nwana [44] differentiates based on mobility, describing agents as either *static* or *mobile*, as well as arbitrary attribute-based categories – namely *cooperation*, *learning*, and *autonomy*. We have incorporated these two classifications implicitly without making specific references to them. With respect to static vs. mobile, all agents in Thinklab are capable of existing at a specific location (or over a specific spatial region), in the sense that all agents can specify whether they exist over space and/or time. The specification of agent behavior in Thinklab is expected to be closely tied to their semantics as specified by the ontologies adopted; the relevant details of the project in this respect are not fully finalized at the moment. For the purposes of this work, any agents which have a spatial component are capable of moving, whether they choose to or not. As for arbitrary attributes, we defer to object-oriented software design to provide the modelers and developers with more fine-grained control beyond the categories above.

Russel [50] chooses not to categorize based on internal features and instead defines agents by the external contracts they fulfill, in terms of *perceptions* and *actions*, and presents optimization techniques by which they may do so. We define agents in similar external terms, but have chosen to categorize them by more traditional internal structures because these fit with pre-existing models that have been incorporated into Thinklab.

Primarily, the *information carrier* and *deliberative* agent types are used in our initial Thinklab prototype. This is not explicitly restricted, but the default agents we have built fall into these two categories. Raw data sources (such as topological maps) and messages are of the first type, and normal "agents" are generally the second type. At the time of writing, no social or organized agents had been created, but the mechanisms exist to create them if desired.

2.1.4 Ontology of Communication Structures in Literature

In Multi-Agent Systems, agent interaction patterns are arguably more important than agents' internal workings. Here we describe the most common patterns from literature; the review by Horling and Lesser was extremely useful here [24]. Other papers we reviewed were from Carley [10], Jennings [25], Macal [32, 33], and Parrot [47], but these named no communication structures not present in Horling and Lesser's work.

Hierarchies are simple, tree-like structures. Agents can communicate with their parents and children; agents higher in the structure have more visibility and influence.

Holarchies are like hierarchies with portions of the tree grouped together as *holons*. Holons share common characteristics, allowing functionality to be segmented. Also, connections can be made between holons across the structure for a more web-like arrangement. In their web-like form, holarchies can be compared to *small-world networks* [59].

Coalitions allow agents to arrange themselves into temporary, goal-based teams. Agents are expected to be selfish rather than altruistic or team-driven (cooperative), and so agents use the coalitions to meet their own needs. Each coalition is generally a flat structure internally, with the possibility of one leader or representative which coordinates tasks internally and communication externally.

Teams are similar to coalitions but with cooperative agents, long-running team membership, and long-running goals and priorities. Internally, teams can be structured using roles which may change over time.

Congregations are grouping structures similar to coalitions and teams, with a unique feature that congregations are not goal-driven. They are formed by mutual, *general* self-interest (rather than specific or temporary goal-oriented interests) and similarity in the members' capabilities. Congregations are long-lasting but dynamic, and communication between congregations is generally limited.

Societies introduce social contracts among grouped agents. The contracts (known as *social laws* or *norms*) define external behaviors of agents; agents may break the terms of a contract but will be sanctioned if they do so. Societies allow system designers to state goals and policies explicitly while still allowing a measure of flexibility and decentralization in agent interactions.

Federations are quasi-political structures in which agents delegate representation to a single leader of each federation. Leaders are expected to negotiate on their constituents' behalf to achieve cooperation with other federations, and the constituents are given a single, consistent contract by which to operate. Federations also allow these localized contracts to differ, so they are effective for integrating heterogeneous actors.

Matrix Organizations introduce the idea of *categories of influence*. Each direction of an agent matrix corresponds to a group within the population; rows may be peers in a social group while columns represent professional team (with a manager in the top row leading the team). In these structures, agents prioritize influences and commitments from various directions to guide their decisions.

Compound Organizations is a term which acknowledges that agent interactions do not fit cleanly into categories, but are usually some combination of the above ideas along with system-specific needs. Different structures may be blended, superimposed, or structured hierarchically. Horling and Lesser mention that some combinations work especially well together, for instance teams with hierarchical internal structures.

In Thinklab, no explicit communication structures have been created. We anticipate that developers will need a fine-grained set of such specifications, and again the core semantics will be the guide to selecting specific types of agents, communication capabilities and behaviors based on inheritance of agent identity from core semantic types. In our current use case scenarios communication structures are of obvious importance (for instance, both scenarios involve government/family interactions, suggesting at least a hierarchy). As Horling mentions, some structures emerge out of patterns of behavior, rather than being dictated from the top down. Our government/family actors observe each other directly as a basis for their interactions, and so could be described as having a simple emergent communication structure.

Messaging Systems

Agent coordination systems normally employ specialized agent roles to facilitate agent interaction, for instance *yellow pages* (directory agents) [24]. Some systems allow messaging based on a blackboard system, in which all agents can see all messages and can respond as desired [21, 23].

The type of messaging system used in an agent platform is for the most part orthogonal to the interaction system(s) employed, but some agent organization structures will imply certain messaging characteristics. For instance, any kind of team- or coalition-based structure would suggest that the messaging system can be partitioned by team, group, or specialty; structures with privileged roles would likely require visibility constraints in the messaging system; etc.

A closing mention should be given to FIPA standards which apply to messaging, and imply certain agent coordination structures, such as *contract nets* [6]. FIPA and FIPA-based systems do not directly enforce agent structures, but their usage does fit roughly within the categorizations given here.

2.2 Functional Reactive Programming

Functional Reactive Programming (FRP) is a programming paradigm which grew out of the need for efficient and intuitive representations of objects and actions in computer animation. It is an improvement over discrete-representation techniques which tightly couple computation and (real-world) display of images and animations. FRP begins with the basic assumption that time and other dimensions are continuous, and should be explicitly treated as such [13, 15].

Historically, computer animation had been done using frames of raster images to represent objects in a way that resembles the operation of computer monitors, which use pixels to display images that are refreshed at some discrete time interval. When animating raster representations of objects in space, all values of all locations on the monitor are re-computed, even if few of their values change at any given time step. This model of computer animation is computationally very expensive.

Pixel image formats and frame-based animation mimic the computer monitor in a software form. At the lowest level, image display software must operate using discrete forms if it is to interact with a computer monitor, but there is no limitation to how an object must be represented in software. The only requirement is that the representation can be converted into an appropriate format for display, and it must unambiguously describe the object.

In reality, objects move fluidly through time with no correlation to the arbitrary time steps imposed by screen refresh rates, processor or memory frequencies, discrete command steps, etc., and they are also composed of continuous shapes rather than pixels. The goal of FRP was to decouple the task of representing fluid objects from any specific representational scale by introducing layers of abstraction. It exposes these qualities of the real world explicitly, rather than hiding them behind artificial representations of pixels and time steps. Discretization can be introduced as needed during rendering, so that an object's observed values are synchronized with screen refresh rates and pixel density.

The main primitives used in FRP are *behaviors* and *events*, introduced in work by

Elliott et al. which preceded FRP [14, 12]. Time in FRP is treated as a total order (any two instants in time must be comparable) but there is no requirement that it be represented using discrete time steps of equal size or even absolute scalar values [13].

Behaviors and events were originally imagined in an idealized way that resisted efficient implementation, especially using an imperative language. However, Elliott's later paper sheds light on implementation by introducing the *push/pull* design, an elegant solution for dividing behaviors into their *discrete* and *continuous* components and treating each one separately (see Section 2.2.1)[15].

FRP specifically acknowledges that some phenomena may be arbitrarily short, and therefore may avoid detection in discrete environments with coarse sampling rates. It uses the term *events* to describe these phenomena which are essentially discrete momentary occurrences. One implementation of FRP used Interval Analysis [52] to reduce the computational effort in event detection, which is essentially a technique for detecting periods of time where events will provably not happen. These time periods are not analyzed for events, and instead computational effort is focused on periods of time where the events might happen.

2.2.1 Push-Pull FRP

Push-Pull FRP [15] is an implementation of FRP which takes more optimization details into consideration. It treats data differently according to whether it changes discretely or continuously: discretely-changing, instantaneous events are *pushed* to any component of the system which is affected (much like an Observer design pattern [18]), and between these discrete events, continuously-changing values are *pulled* by the observer on demand.

In Push-Pull FRP, time-varying values are segmented into non-reactive, continuous *time functions* which occupy the time periods between discrete, momentary *reactive values*. Real-world activity is modeled using *reactive behaviors* which are composed of these two components.

Push-Pull FRP treats discrete changes in a manner similar to event-driven programming, and so is a good basis upon which to design a semantic model which can be implemented in an object-oriented, imperative language (Thinklab is written mainly in Java). Code written in an event-driven system has the purpose of *responding to events* in a system, rather than *dictating an order of execution*, as is the case in script-based or procedural programs. Event-driven programming and FRP's discrete event semantics are also similar to the Observer design pattern [18].

Push-Pull FRP treats discrete changes as events which cannot be known in advance and are handled as they occur. This concept also informs our treatment of discrete events in a simulation (see Section 4.4.5).

2.3 Geospatial Interpolation

The problem of translating between arbitrary scales takes many different forms in various research areas. One area of research is geospatial interploation, which must be used for some scale translations in our use case scenarios. Mitas and Mitasova [38, 37] give a very concise overview of interpolation strategies in a geostatistical setting. The strategies they describe allow observed phenomena in d-dimensional space which is collected in irregular sample patterns to be interpolated into a target representation with arbitrary resolution.

In their words, the interpolation problem can be described as: given the N values of a studied phenomenon $z_j, j = 1, \ldots, N$ measured at discrete points

$$r_j = (x_j^{[1]}, x_j^{[2]}, \ldots, x_j^{[d]}), j = 1, \ldots, N$$

within a certain region of d-dimensional space, find a d-dimensional variate function $F(r)$ which passes through the given points; that is, that fulfills the condition:

$$F(r_j) = z_j, j = 1, \ldots, N$$

Mitas and Mitasova mention that multiple-scale modeling will require interpolation techniques which can incorporate arbitrary scales, and they propose further work into more general solutions as well as drawing inspiration from the image-processing field. They suggest that wavelet techniques are a group of approaches with good potential in multivariate environments, a conclusion we have also come to. Their suggestion would imply that image interpolation and vector representation are good resources from which to draw, which we have done.

Various interpolation categories covered in their work are described in the following subsections:

2.3.1 Local Neighborhood Approaches

The intuition behind these approaches is straightforward and practical: Any given data point will probably have a stronger relationship with the data points in its vicinity than with points further away. This makes a lot of sense in geostatistics (e.g. rainfall or other weather patterns, elevation, pollution, etc.) and has broad support in the physical sciences in general.

But there are also computational and semantic benefits. These approaches allow *segmentation* of the interpolation process; they allow the problem to be broken into separate computational tasks, enabling parallel computation. Segmentation does not guarantee that the result of an interpolation can be represented in a modular and concise fashion. For instance, although IDW can be implemented as a localized process able to define a totally smooth surface from input points, it does not generate a result which can be represented concisely.

Inverse Distance Weighted Interpolation (IDW)

IDW generates interpolated points by computing a weighted average of input data points. The weights in the weighted average are proportional to the distance and scaled by some exponent p (usually $p = 2$).

To reduce computational expense incurred for points which are far away from the point being generated (and therefore not weighted very strongly), a localized neighborhood can be defined from which to sample: either the closest n points, or the points that fall within a radius r or a grid $([x-a \dots x+a], [y-a \dots y+a])$.

The most simplistic version of IDW would be to consider only the closest input point for each computation ($n = 1$), resulting in a d-dimensional step function for $F(r)$. Evaluating $F(r)$ at any point $r = (x_i^{[1]}, \dots, x_i^{[d]})$ would return the closest $z_j = F(x_j^{[1]}, \dots, x_j^{[d]})$.

For two-dimensional space, a generated data point $f(x, y)$ would be based on some number of input values $v(x_i, y_i)$ in the neighborhood of (x, y):

$$\begin{aligned} f(x,y) &\approx \frac{\sum_i w_i v(x_i, y_i)}{\sum_i w_i} \text{ where } w_i \propto (distance_i)^p \\ &\approx \frac{\sum_i \left(\sqrt{(x-x_i)^2 + (y-y_i)^2}\right)^p v(x_i, y_i)}{\sum_i \left(\sqrt{(x-x_i)^2 + (y-y_i)^2}\right)^p} \end{aligned}$$

This formulation unfortunately cannot be represented as a vector or a set of continuous curves; instead, specific points must be generated from this estimation based on the input data. This has implications which make it less desirable for our purposes (see Section 4.2.2).

Also, IDW without modifications will lead to flat spots at each generated point, which has the effect of disturbing derivative (slope) information. Modifications have been proposed by many authors for working around this problem and preserving an approximation of slope. Another shortcoming is that curves generated by IDW will not generally pass through the sampled points, but rather approximate them. A curve is based on a weighted average of some number of local points, and therefore will never pass through a local maximum or minimum point. This can be a minor or very serious down side, depending on the application.

Natural Neighbor

This technique begins with a *Voronoi tessellation* of the input data points. An interpolated point is generated as a weighted average of values from nearby regions. The weights are generated by computing a hypothetical tessellated region around the point as it would exist if it were included in the data set. Weights are equivalent to the proportion of the newly generated region which would lie in each of the previous regions. These weights are used to compute a weighted average of the corresponding values from each of the previous regions which overlap with this new hypothetical region:

$$F(r_j) = \sum_{i=1}^{n} w_i F(r_i)$$

This is an arbitrary-dimensional approach and generates smooth (rubber-sheet like) surfaces. Additional blended gradient information computed from the data points can be incorporated to smooth further while preserving first and second order derivatives. These characteristics make it a good general-purpose interpolation mechanism for geostatistics and other fields.

The most straightforward version of this technique is two-dimensional linear interpolation between neighboring points spaced evenly in the x direction.

Triangulated Irregular Network (TIN)

This approach generates triangle-shaped surfaces from individual data points by connecting all trios of neighboring points. The points can be chosen by various means; Delaunay triangulation selects points such that the circumcircle around any triangle does not contain any other point. The surfaces represented by the triangles are used as bivariate (or multivariate, in arbitrary dimensions) functions which can be solved for an interpolated value $F(r_i)$. TIN can also employ non-linear techniques to blend the edges of the surfaces together by incorporating the derivatives (first or first and second) of the surfaces in the result. This will also retain differentiability of the result.

TIN techniques perform well computationally because they are localized and therefore parallelizable, however they do not give the highest quality results. Because the triangles are exact, continuous representations of the interpolation, this approach is one which satisfies our preference for *continuous intermediate representation* (Section 4.2.2).

Rectangle-Based Methods

What Mitas and Mitasova describe as *Rectangle-Based Methods* are included in this thesis as *spline* techniques. These techniques are well-represented in image processing and interpolation fields. Splines represent d-dimensional shapes as vectors with curvature that comes from generating functions. These are discussed in Section 2.4, with attention given to wavelet techniques commonly used to scale raster images up or down. Using wavelets is conceptually similar to Fourier-based approaches, in that wave-shaped generating functions form a basis for discretizing a continuous signal by storing function parameters. Later, re-constructing that signal from the stored parameters is a matter of plugging the parameters into the generating functions, possibly integrating many sub-signals together.

Spline methods can optionally apply tension to the vectors that are given, as a mechanism for smoothing the result. One specific algorithm from Mitas' and Mitasova's prior work [40, 39] uses a smoothing and tension approach which preserves all orders of derivatives, so that mathematical analysis on the resulting data set can be performed. This can be useful in geostatistical sciences where such derivatives are important, for instance using runoff and erosion models in which slope influences water flow characteristics.

Other benefits to smoothing are to make a rendered image more visually pleasing, or even to improve accuracy in cases where interpolation using general algorithms alone results in jagged or bumpy intersections, or reducing aliasing patterns.

2.3.2 Geostatistical Approaches

These approaches generally inherit from *Kriging Methods*, a family of interpolation methods developed for geostatistics in which certainty can be established for all generated data points along with the generated points themselves. This type of algorithm will generate a curve, and so can be rendered or incorporated into models directly, but Kriging methods are unique because they also generate confidence intervals which express the range of possible values based on statistical certainty.

However, Kriging methods are computationally expensive, requiring specialized resources for large data sets [28]. Using them should therefore be restricted to cases where their unique benefits are needed.

2.3.3 Variational Approach

This technique minimizes the sum of the deviations and maximizes the smoothness of the result. In some applications of this technique, mathematical problems may be introduced which prevent accurate analysis of a resulting surface. This is similar to the *Spline methods with tension* mentioned in this section, but uses a different representational primitive.

2.3.4 Application-Specific Approaches

These domain-specific approaches can take many forms.

Stream Enforcement is an example of how domain-specific metadata can be used to tune or refine the results of a more general algorithm. As is the case with other examples from Mitas' and Mitasova's work, stream enforcement is a technique used in geostatistics. It superimposes the shapes of known streams and rivers over a topological map being generated from surveyed sample points. The algorithm converting sample points into a smooth surface (especially using triangulation techniques mentioned here) may make assumptions that dam-shaped land masses exist in valleys when they actually do not. The assumption is reasonable; two sample points which are very close together will often be in fact joined by a mass of land. But in valleys with steep slopes on either side, this assumption does not hold. Stream maps which are superimposed over the sample sets are domain-specific metadata which inform the algorithm that these land masses should not be created between two points divided by a stream.

Metadata techniques like this are effective ways of introducing domain-specific knowledge into a general algorithm. They are ways for algorithms to incorporate additional information, in this case that a stream runs through a valley being rendered, to help select between interpretations of the original data.

2.4 Image and Signal Interpolation

Image interpolation uses general multidimensional interpolation approaches to convert from one pixel representation either into a continuous representation or another pixel representation. Often times, the algorithm used will generate a vector-based (continuous) intermediate result. This topic can be seen as a simplification of Geospatial Interpolation, because image and signal interpolation are done in well-controlled environments. Input values lie strictly within known boundaries and represent phenomena with very consistent physical properties. Representational formats are primarily matrix-based, using regular sized units, and generally only one interpolation strategy is used in each setting.

De Boor's review of splines [11] covers interpolation techniques as well as splines and spline types in general. This book is a *de facto* reference for vector-based representation of data points, and is not strictly an image-representation study. It focuses on splines, which are extensively used in image interpolation, so we include it here.

Kopf [29] presents a new technique for generating high quality vector images from low-resolution bitmaps. His work is also an example of *domain-specific vectorization*, a general category of adaptation procedures by which a general interpolation approach can be tuned by inferring beliefs about what the low-resolution image represents. More on domain-specific tuning is given in 2.3.4.

Image interpolation is a good resource for our work because the underlying algorithms tend to be general. They deal with color values distributed over multi-dimensional space (whether two- or three-dimensional), which is not very different than arbitrary state values as a function of multi-dimensional space and time. Many of the underlying processes used in image interpolation will work in arbitrary dimensions (even beyond four-dimensional space-time), so this field is rich with resources for our work.

Image interpolation generally uses the appearance of an image as the standard for quality. In some cases, a certain amount of accuracy or a specific trait is required. For instance, in elevation interpolation where differentiability is needed, these techniques may not apply. Image interpolation is a good starting point, providing procedures in general cases, and it is also a good platform for developing the components of an interopolation solution.

2.4.1 Lanczos Resampling

Lanczos resampling applies a sinc function to coded signals above the *Nyquist frequency*, from the Nyquist–Shannon sampling theorem [26]. This is the frequency which is double (half the wavelength of) the highest-frequency input one wishes to encode. Using Lanczos resampling results in a very accurate representation of the original data, but requires high-resolution inputs.

The sinc function is a type of wavelet, and so it provides two essential features for large-scale, general-purpose interpolation: *segmentation* and *continuous representation*. These are discussed in Section 3.2.1.

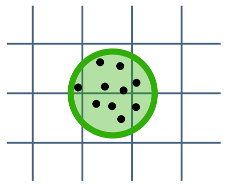


Figure 2.2: An agent performing ten consecutive randomized queries

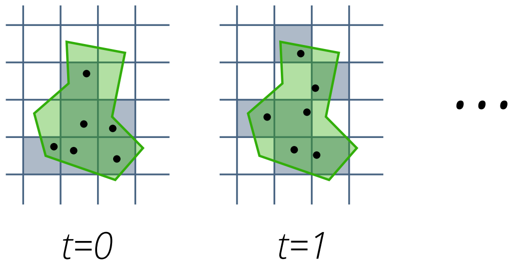


Figure 2.3: An agent may perform fewer queries per step over the course of many steps

2.4.2 Oversampling (Monte Carlo)

One approach used in image interpolation is oversampling, a Monte Carlo technique for computing a weighted average value for a cell whose boundaries do not correspond cleanly to boundaries in the input data. Relating this type of process to geostatistical and other data interpolation tasks is straightforward. Given an input which has an unrelated space/time grid, an agent may take random samples within the range in which it is interested and average the values to determine an appropriate aggregate value.

The oversampling technique addresses the problem of overlapping cells in the following way. Every time an agent queries a data source with a different scale (i.e. overlapping cells), it does so by selecting a random point within its range. That random point is then projected onto the data source and the cell within which it lies is the cell that is queried.

An example of a single value being generated by ten samples is shown in Figure 2.2. This will produce a weighted average of the underlying data which overlaps the agent's area. (Note that not all overlapping cells will necessarily be chosen during any given number of random selections.)

In more complex cases, where an agent overlaps with many cells of underlying data and must generate aggregate information, a less intensive version can be used by averaging some number of random samples per time period (see Figure 2.3). In each time step the agent may choose fewer than one sample from each underlying cell, but over time all cells will be represented proportionally to their overlap with the agent.

2.5 Signal Theory

Signal theory deals with discrete representation of continuous signals, and is in general a study of two dimensions: one-dimensional discrete values being transmitted across one-dimensional time through some communication channel with known properties. It studies the accuracy and effectiveness of reducing continuous, real-world phenomena to discrete representation. Higher-dimensional data can be represented by reducing it to a series of one-dimensional discrete value samples.

Signal theory can be employed in our work to reason about the accuracy and effectiveness of our observation and scale mediation techniques, and about the theoretical limits of the discrete representation of observable phenomena, and the reliability of re-constituting or re-generating information from a discrete representation.

The limitation of signal theory as it applies to us is that it is generally a study of one-dimensional phenomena. Signal theory is at its core is *data representation and transmission*. The data can represent arbitrary dimensions if the representational format allows it; signal theory does not place a restriction on the dimensionality of the data, *per se* but it does inherently enforce data to be represented as a discrete-valued time series as an intermediate step. Still, it could be said that we have the same limitation: computers represent all data in physical, binary, memory- or disk-based formats, which are inherently one-dimensional, and so our techniques are at some level strings of discrete values as well.

2.6 Collision Detection

There is a rich body of work on collision detection, with contributions from both industry and academia, especially in the field of video game animation. The field of collision detection is broad; even the problem is stated differently in different contexts. For each form of the problem, many strategic approaches exist. The broadest subcategories within collision detection are *dynamic* (generally *a priori*) analysis by solving closed-form time functions for object motion, and *static* (generally *a posteriori*) analysis of objects at pre-rendered locations [16].

Many approaches involve partitioning space by bounding boxes, cones, etc., and identifying non-overlapping regions of objects for which collision computations can be skipped [16, 27, 30]. This is the spatial equivalent to (temporal) Interval

Analysis (IA), an optimization for event detection which has been leveraged by FRP (Section 2.2). IA is used as an *a priori* technique, and so is equivalent to dynamic collision detection approaches. Although we do not use IA directly, it has informed our strategy for event semantics. Similarly, we would like to use dynamic approaches to find the most efficient techniques for detecting collisions in Thinklab.

In contrast, static techniques deal with spatial positions already generated during some time step in a simulation or animation. These are the more common algorithms in the field, and unfortunately do not apply very well to our formulation of the problem. We do not anticipate their use being very effective in Thinklab, but by our judgment there should be no barrier to implementing these types of algorithms using our system.

We provide a good interface for incorporating collision detection algorithms, but leave their implementation for future work (see Sections 4.4.4 and 6.1.1).

Chapter 3

Problem Statement

Thinklab is an ontology-based semantic meta-modeling and simulation platform. A simulation in Thinklab is a series of observations by agents, which at a system level can be seen as raw data exchanges. The *subject* of an observation is also an agent; we call the states that are observed *agent-states*, which are composed of property-value pairs over specific time periods. Agents experience observations subjectively by *perceiving* the properties of other agents, and may respond to the observations by making decisions and taking actions. The decisions made and actions taken by an agent may result in *agent-state transitions* (changes in the states of observable properties of the agent), or they may result in *collisions* with other agents (a general term we use to refer to agent-state transitions forced upon agents which are not a result of their own internal decisions). Because agents and their decisions are observable by other agents, the process of a Thinklab simulation can be expressed through a graph. Although we describe states and state transitions, agents are more complex than state machines, so we will use state machine terminology and concepts without pretense of rigor.

Observation is the main process we deal with in this thesis. An agent observes another agent at a specific instant and makes decisions that affect its states over a specific temporal duration; we also allow for continuous observation, probabilistic reasoning, and continuous, time-varying states in a restricted interpretation. These special cases fit within a semantic framework of instantaneous decisions with specific temporal effect (see Section 3.1).

Before this work, non-synchronous scales had seen limited treatment in agent systems, and were not incorporated into the reference implementation of this work (Thinklab). What are now referred to as *agents* had been only capable of observing and being observed in a fixed time instant. Some probabilistic representation had been introduced in the form of Bayesian network data accessors, but actual behaviors could not be carried out in a temporally dynamic simulated

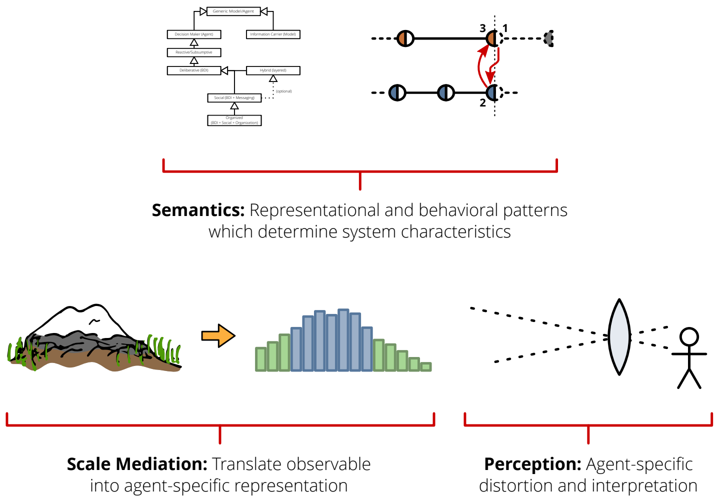

3
1
2
Semantics: Representational and behavioral patterns which determine system characteristics
Scale Mediation: Translate observable into agent-specific representation
Perception: Agent-specific distortion and interpretation

Figure 3.1: The three main problem categories within multi-scale simulation

world. The work presented here allows time to move forward, and for agents to interact while maintaining subjective views of space and time.

Related work reviewed by the authors did not contribute meaningfully to the design of a truly multi-scale simulation environment. Scales in other systems are expected to be ultimately synchronous in all dimensions. According to Balbi [5], most systems are either spatially non-specific (deferring spatial reasoning and interaction to the implementation), or assume a single synchronous grid of space. All reviewed systems had a single, fully synchronous time schedule with time steps of equal duration.

The problem of multi-scale integration breaks down into three main categories (see Figure 3.1): the *Semantics* of reasoning about scale, both in terms of the software API and the agents' reasoning abilities at runtime; the *Scale Mediation* required when the agents in a system have different internal representations; and the *Subjective Perception* that an agent may have, either by its relative position in space-time or by how the agent processes perceived states. Semantics is covered in Section 3.1, scale mediation in Section 3.2, and Section 3.3 covers agent perception. Throughout the thesis, use-case scenarios will be used as examples; these are introduced in Section 3.4. Finally, Section 3.5 discusses deliberate simplifications we have made to ensure usability in terms of complexity and performance.

3.1 Semantics

The problem of scale must be handled in a semantically consistent way if it is to be incorporated into a useful modeling system. The semantics chosen must be flexible and meaningful, but also intelligible and decidable.

The purpose of this work is to provide modelers with tools to build agents whose perception, and as a consequence their reasoning and behavior, is influenced by the different scales adopted by different interacting agents. Mechanisms should be provided to implement behaviors as fluidly as possible, but differences in scale and scale translations should be explicit and controllable. This requires that appropriately expressive semantics be created for representing scales and the operations that can be done on them, and also for the reasoning and behaviors that result from observations made across differing scales.

We do not want differences in scale to be hidden from agents or the users who create simulation models, nor do we want to require users or agents to reason about scale unless it is important to their functioning. Scale should be an explicit, first-class concept which can be reasoned upon but also automated by straightforward logic within the core Thinklab system. This logic must be intuitive to understand for users, because designing agents with the ability to reason about scale implies that these agents are performing reasoning tasks equivalent to the automated reasoning performed by Thinklab. Also, it should be possible for agents to fully replace, or to perform only a subset of, what is normally automated by the system, which implies that these automated functions are both modular and intelligible.

Within the field of agent modeling and simulation, there is a wide variety of agent representation semantics. FLAME [23] uses XML files to define agents as enhanced state machines; procedural reasoning is not present in FLAME agents, and so the execution model is not rich enough for our purposes. SWARM [22, 36, 53] and Repast [42, 43] allow modelers to create code objects in Java and other languages, so procedural reasoning is possible, but in both of these systems, the world of interaction is a single, synchronous entity in the simulation, meaning that the issue of heterogeneous scales is not addressed within these systems (or, more precisely, it is the problem of the modeler to deal with it before introducing external data).

Some projects using these systems have concepts of heterogeneous scale as a lightweight layer of translation on top of a synchronous lattice representation of space/time. Balbi [5] reviews many systems built for socio-ecosystem research, specifically identifying how space and time were treated in each system. About half were aspatial; the spatially explicit systems used combinations of cellular and GIS-based representation. Temporal scales were all synchronous, with simulation time "steps" representing some number of years. One slight exception to this was Berman's study [7], in which year-long time steps were broken down into fourths or fifths for certain event types. This small departure from synchronous time and space does not fulfill our goal of truly heterogeneous scale representation, so we had no previous analysis of the problem to follow.

3.1.1 Cyclical Dependency vs. Circular Reference

In this thesis, we draw a distinction between *cyclical dependency* and *circular reference*. Cyclical dependency is a natural process by which two agents influence each other over time; a circular reference is an un-resolvable dependency in a single instant of time in which two or more elements depend on each other in a way that none can be computed. A circular reference can be an artifact of a limited view of time and a product of flawed model design: most circular references in models disappear if the notion of time implemented is made flexible enough to solve the problem of simultaneous write access to co-dependent states. As an example, consider the implications of the speed of light and event horizons: beings cannot influence each other immediately, and the synchronization of information transfer in cause/effect relationships can only be slower than the speed of light; therefore, simultaneous effective co-dependency is impossible.

A system which models real-world interactions must allow the unlimited definition of cyclical dependencies, because such interactions are common in real life. For usability, accuracy, and the convenience of finding errors quickly, the system should also be able to detect and handle circular references showing up as artifacts. The system should fulfill these requirements explicitly, problem scenarios should be unambiguously classifiable as either cyclical or circular, and the cyclical problems should all be computable in a straightforward way.

3.2 Mediation Between Scales

Agents in a modeling system may have internal representations that are required for their correct functioning. In some cases, agents' internal representations will be matrices of discrete values, and in other cases they will use continuous values (e.g. vectors) to represent their state. Agents may also represent values as qualitative classifications or semi-quantitative ranks. In other cases, an agent may simply apply a modification to underlying data, and can inherit and re-present arbitrary representation schemes.

Simulations using heterogeneous scales will be forced to translate observable phenomena from one scale to another using scale mediation mechanisms. As mentioned in 2.3, the scale mediation problem can be stated as: given the N values of a studied phenomenon $z_j, j = 1, \ldots, N$ measured at discrete points

$$r_j = (x_j^{[1]}, x_j^{[2]}, \ldots, x_j^{[d]}), j = 1, \ldots, N$$

within a certain region of d-dimensional space, find a d-dimensional variate function $F(r)$ which passes through the given points; that is, that fulfills the condition:

$$F(r_j) = z_j, j = 1, \ldots, N$$

In this section we describe the various elements of the scale mediation problem.

3.2.1 Quality and Performance

The effectiveness of a scale mediation solution depends primarily on two features: *segmentation* and *continuous representation*.

Segmentation is the ability to split a data transformation problem into segments, and is essential to processing large data sets, as it allows parallelization of the computation. Data represented using values with only localized context can be segmented. For instance, the coefficients of a cubic spline model representing some z value within a (x, y) surface only affect a small number of neighboring values. Contrast this with polynomial vectorization, which takes all data points into account and generates a high-dimensional formula whose coefficients fit the resulting curve to all points along each dimension. A change in any one of the coefficients of a polynomial vector will affect the value of all points, whereas a change in one coefficient in a cubic spline model will affect only a few points immediately around the center of the change.

Continuous representation uses continuous, real-valued curves to represent the data set, rather than measurements located at discrete locations and having possibly limited-accuracy values, as in a bitmap image with finite color depth. Continuous representation allows data to be represented concisely and accurately, and also allows data expressed in one scale to be re-sampled into multiple arbitrary scales without repeating the full scale mediation process. This can be accomplished by saving the continuous-valued representation of the data after a conversion is done, and re-using this representation as the basis for generating discrete data in another scale.

3.2.2 Interpolating Between Sampled Points

Geostatistical data sets are commonly assembled from some number of discrete measurements. Various data collection techniques may be used, depending on the phenomena, location, instruments used, etc. Each of these instruments or collection techniques introduces a degree of granularity into the data set; the resolution generated by a soil survey may be a grid of categorical soil quality values with a two-meter distance between each sample point, whereas non-remotely sensed sample data such as biodiversity profiles might be taken only where humanly possible, which may be very limited, such as on exposed cliff edges or swampy jungle terrain.

Innumerable other possibilities may preclude samples being taken at regular intervals, such as time or financial constraints, political or legal hurdles, bad weather or visibility, etc. In this way, geostatistics shares our need to accommodate source data which is either incomplete or in multiple incompatible scales, including scales which differ in their fundamental treatment of the measured phenomenon.

All the limitations of real-world sampling (irregularity, inaccuracy, discrete or blunt values) have equivalents in a computing environment. Values in a computer are necessarily finite, although they can be expressed with very fine resolution; inaccuracies are always present in parameterized computations when parameters are measured or assumed; and irregularities occur when accommodating heterogeneous scales in the form of *aliasing*. Aliasing refers to the patterns which are created when two differing sampling schemes are overlaid on each other. It can commonly be seen in small fonts which become pixelated or in digital images of subjects with regular patterns in them (such as a photo of a mesh screen taken with a digital camera). These patterns can manifest as jagged edges, spots, or warping.

Therefore we would like to draw inspiration from multivariate interpolation techniques in other areas and apply them to the simulation environment. This will allow us to leverage thinking which has already been invested by experts, and it will also provide domain-specific solutions which more closely fit the needs of the data in question.

3.3 Subjective Agent Perception

Agents have subjective views of space and time. Real-world agents' experiences can include distortions due to physics, error-prone sensors, limited experiential capacities, distance, interference, or other factors. Agents in the target implementation aim to reflect real-world semantics, so accuracy in the representation of perceptual scale distortion is a goal. Agents may sense distance or take other measurements accurately at close proximity, but less accurately at longer range; they may wake up and become dormant at certain intervals, which causes their awareness of time to vary; or they may be subject to physical effects, such as heat or magnetism, which affect the physical properties of their inputs.

These are all examples of ways that the same world can be *perceived* differently by agents with different sensors, internal structures, or positions in space. These effects of subjective perception can be broken down into three categories: *perspective*, *accuracy*, and *distortion*. These do not need to be treated explicitly as separate concepts by a simulation system, provided they can all be included in simulated agents.

3.3.1 Perspective

Perspective is a subjective distortion due to the laws of physics. This could include parallax distortion, vanishing points on the horizon, objects hidden behind other objects, or other natural distortions that happen because of physics. The reason why we differentiate this from other subjective effects is that it happens *outside* of the agent, in other words, it is just a characteristic of the way that phenomena *arrive at* an agent (the way light approaches the lens of an eye or a camera), and it is not because the agent is incapable of interpreting the inputs accurately.

We would like to include these filters internally in the agents, even though the physical effects happen outside the agent, because we would like to reserve the possibility of creating agents for which these distortions are not present. An agent might be composed of a distributed sensor network which is not subject to interpreting input at one physical location, for instance a satellite system with a processing center in one city. This type of agent has a physical center of computation and is therefore considered to be *located* there, but its inputs are distributed over its environment.

3.3.2 Accuracy

Agents' inputs can vary in accuracy based on the qualities (or quantity) of their sensors. A real-world example is the various ranges of light which are visible by different animal species. Humans see what we call the visible spectrum, but other animals may see frequencies below or above our range. Combinations of frequencies can be identified with more or less accuracy based on the number and type of cones present in the animal's retina. Similarly, the senses of smell can differ between species, and the sense of touch can vary widely even between locations on the same animal body.

3.3.3 Distortion

Distortion effects are those in which an agent experiences one class of phenomena and makes assumptions or develops meaning in another related class, for instance judging spatial scale based on the amount of time that has passed or the visual or emotional cues that have been triggered.

The same external phenomena can be experienced by an agent differently at different times based on the agent's state. An animal that is sleeping, for instance, may experience the time asleep as passing very quickly; many hours of sleep might be perceived to be the same as only a few seconds of waking life.

3.4 Use Case Scenarios

To make discussions more concrete, and to make the concepts more accessible, we have created use case scenarios which illustrate the functioning of the Thinklab system and the requirements we would like it to fulfill.

The scenarios involve multiple agents interacting with each other in real time. All agents have inputs which correspond to observations of other agents' behaviors, and therefore they have a strict dependency on the forward movement of time.

Cyclical dependencies are immediately noticeable in both of the diagrams. These types of influences must be accounted for in a general modeling system because they exist in the real world. However, true circular references do not exist. As introduced above, circular references in modeling/simulation system are usually the products of flawed design; our treatment of this issue is covered in detail in Section 4.4.3.

Events that may take place in these scenarios are changes in government policy, socio-economic activity (working, farming, moving to another village), agents coming into existence or being destroyed (family units either coming of age or dying), or natural phenomena (rainfall and flood patterns). Events may take place as a result of procedural decision making, as would be the case with an autonomous deliberative agent, or as a result of thresholds or other requirements being met, such as a family being forced to move when there is no food available. Observations may be directly made of other agents, or they may be aggregated observations over collections of other agents. These events and observations should be straightforward to model and should be clear to users who observe the simulations being executed.

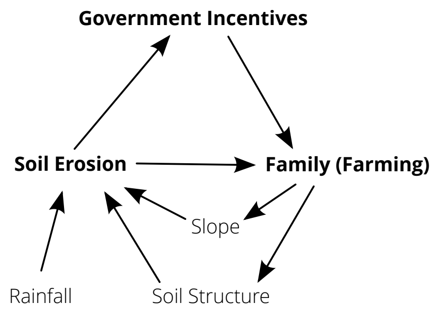


Figure 3.2: Agent interactions in the Soil Erosion scenario

3.4.1 Scenario 1: Soil Erosion

Some of the primary drivers of soil erosion risk in a particular area of land are rainfall, slope of the ground, vegetation root strength, and human activities such as farming, foot or car traffic, and clear-cutting the land (see Figure 3.2). We can develop estimations of potential soil erosion areas and model people's behavior in response to soil erosion, as well as people's contribution to soil erosion.

Also, governments may intervene by providing subsidies or penalties to modify people's behavior. In this example, we assume there is a government program which, when the soil in an area is at risk of erosion, pays farmers 50% of the potential crop value if they decide not to farm.

3.4.2 Scenario 2: Food Security

This scenario is more complex (see Figure 3.3). Administration of economic and food policy is handled at both the regional and village level for a certain community. Family units may make decisions such as whether to farm, work in other employment, collect food aid, or to move to another village depending on various criteria, and one result of decisions in the context of available resources is a given family's level of food security. The criteria which affect families' behaviors may be availability of water, arable land, food for purchasing, economic policy, risk of flood, and risk of events such as social or political upheaval. Also, families' behaviors influence the availability of food and employment in a given village.

These three agent types – family units, leaders, and environmental systems – all respond to each other in patterns currently being explored by field researchers related to the Thinklab project. The researchers develop Bayesian networks for decision making based on their findings, and the natural phenomena (rainfall, flood danger) is modeled in the system through field studies and from published geophysical data.

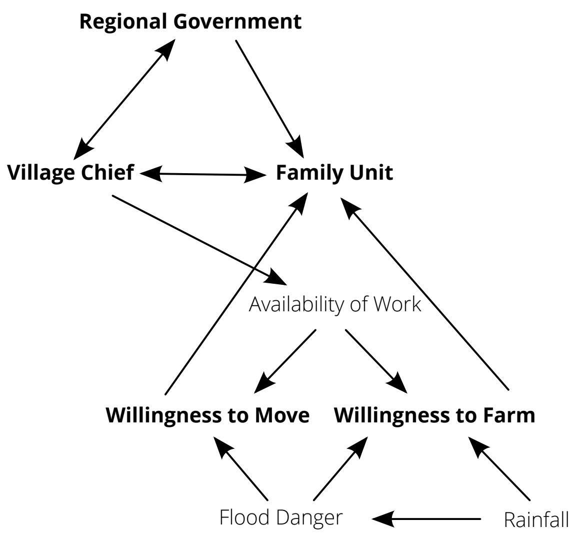


Figure 3.3: Agent interactions in the Food Security scenario

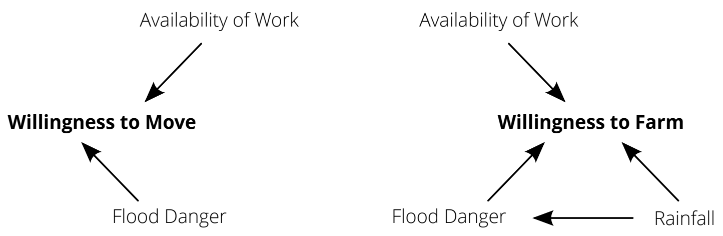


Figure 3.4: Willingness to Farm and Willingness to Move can be seen as independent Bayesian networks

"Willingness to Farm" and "Willingness to Move" are concepts that may be resolved in Thinklab to models that use Bayesian networks. Each represents an abstract decision factor on which a family unit will rely to make decisions about whether to change their location or farming activity (Figure 3.4). We would like to allow Bayesian networks to be used as the strategy to compute concepts representing inputs for agents.

In this scenario, decisions are made by families and by the village chief depending on collective economic activity, and the collective actions of village chiefs influences regional government decisions. This implies that aggregate functions must be present in the simulation system.

3.5 Deliberate Simplifications

As we developed Thinklab, we could only spend a finite amount of time in design and development. Also, it is bad practice to over-develop a system in the attempt to obtain indiscriminate flexibility for every use case. The simplifications in this section constitute the boundaries of our work, but it should be possible to expand into any one of them as the need arises. We leave these expansions as Future Work (see Section 6.1).

3.5.1 Handling of Relativistic Space/Time

An observer's experience of the physical universe will depend on the observer's position in space and time. According to Einstein's theories, time, space, and gravity are co-dependent, and are experienced subjectively by observers. Special relativity states that observed events cannot be ordered in any absolute way; events which appear ordered a certain way from one point of view may be ordered differently when seen from another point of view.

A truly general modeling framework would allow for observations which include time delay and spatial warping based on an agent's position and velocity. It would also include the limitations of the speed of light itself (e.g. event horizons). Our system has not been used for a simulation task requiring these, so instead of trying to adopt a space/time view compatible with relativistic representations, we impose a synchronous integration of time and space, allowing the agents to internally diverge from or re-interpret this global synchronization as they wish.

A simulation is driven by an agent we call the *root subject*, which implicitly has its own subjective view of space and time. The global synchronization mechanisms we use could be said to represent the dimensions seen by this agent. However, this ordering is also enforced upon the various agents within the simulation as they interact with each other. In other words, each agent within the simulation is forced to inherit the total ordering of events as experienced by the root subject, even if they are not aware of its existence. This assumption should not cause any limitation representing events and phenomena whose scales are compatible with human history and experiences.

3.5.2 Arrow of Time

A large catalog of work exists on the "arrow of time" in terms of thermodynamic and cosmological laws [49]. Physicists do not agree on the nature of the forward progress of time; it is at best negotiable what implications exist about causality, reversibility, the direction of increasing entropy, expansion or contraction of the universe, etc. We leave our mention of it at this rudimentary level, because to include this type of question in our research or to leave open the configuration and modeling possibilities would require an untenable amount of time and would probably not result in a working system at this stage.

3.5.3 Arbitrary Dimensions

In a general modeling system, the number of dimensions seen by an agent must be flexible. Typical agents may measure a data point as a function of time (e.g. a thermometer), as a function of two-dimensional space (e.g. a topological elevation map) or three-dimensional space (e.g. an air quality model), or four-dimensional space and time, (i.e. a three-dimensional model which changes over time).

Other dimensions could include the string- and superstring-theoretic addition of eleven to over twenty dimensions, including *small* dimensions in some cases [48], or could include concepts of *parallel planes of existence* or *modes of agency*, such as a single agent acting under many different prototypical roles simultaneously, and having various perceptions and abilities but a single cohesive awareness of the world.

Throughout this thesis, time and space will be considered four-dimensional, and agents may be aware of any number of these dimensions. Arbitrary numbers of dimensions outside these views of space-time are beyond the scope of this work, but the design proposed is compatible with arbitrary dimensions. We also include this in Future Work (Section 6.1).

Chapter 4

Design

In this chapter, we describe how we have incorporated ideas from other research areas into our system design. These ideas and the designs they informed collectively allow multi-agent, multi-scale simulations to take place in Thinklab.

Figure 4.1 shows a general overview of the work done for this project. Scale mediation is composed of a few steps, each of which has specific design elements. An agent observes its subject by requesting specific properties from a *scale mediator* which is dedicated to observing one subject agent over its lifetime. Each such request is made for a specific observation time and for a specific observable property, and the result is filtered through the agent's *perception filters*. The scale mediator may return a result by a number of different routes. It may contain a cached result in the desired scale which can be returned immediately, or it may also contain an intermediate vector representation of the subject agent during the observation time which can be quickly generated into the desired scale. If neither of these are present, it will have a reference to the subject's agent-states over time from which it can generate the two representations mentioned above. All subjects' agent-states are kept in a single authoritative *observation controller* during a simulation. Agents proceed through time by generating new agent-states and reporting them to the observation controller, which in turn updates all affected scale mediators through a standard publish/subscribe mechanism.

Incorporating multiple scales into Thinklab is not as simple as introducing a single data conversion technique. The multiple scale problem creates a different set of challenges for different types of agents, different types of observations, and different system functions (observation, collision detection, messaging); all of these areas must be compatible with a unitary semantics for scale and scale mediation.

Time is a dimension with peculiar characteristics. Entire fields of physics are

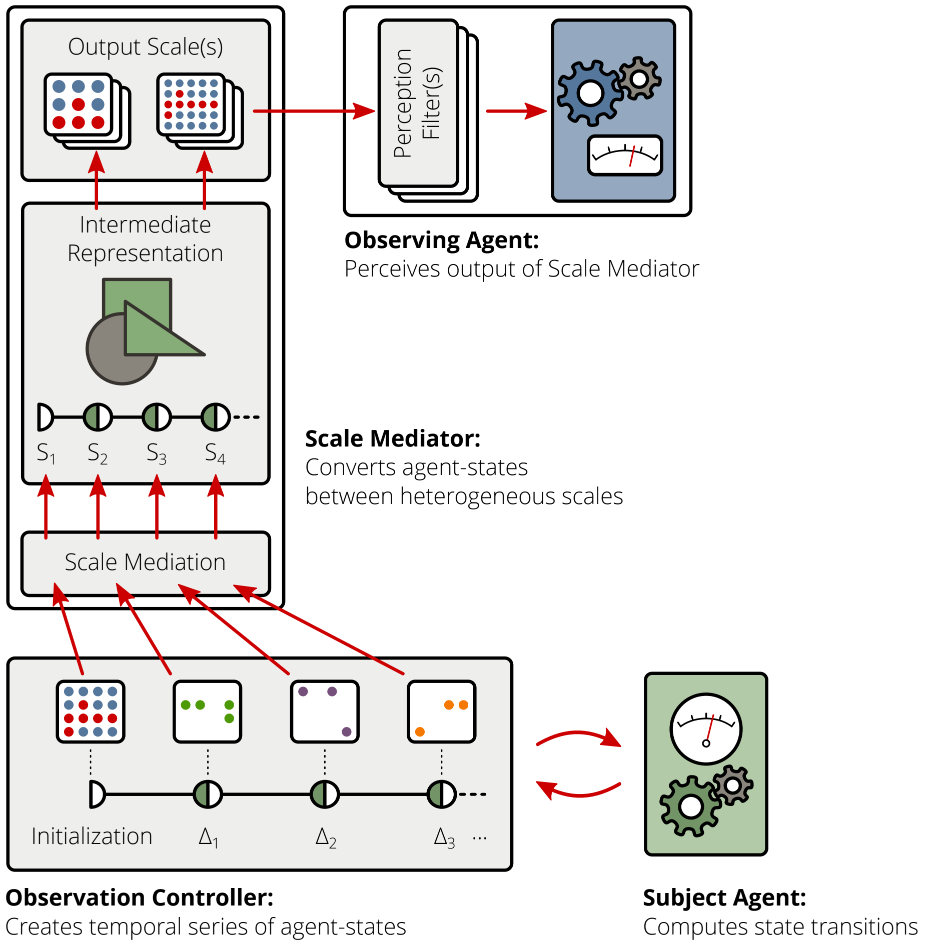

Output Scale(s)
Perception Filter(s)
Intermediate Representation
S_1 S_2 S_3 S_4
Scale Mediation
Observing Agent:
Perceives output of Scale Mediator
Scale Mediator:
Converts agent-states between heterogeneous scales
Initialization Δ_1 Δ_2 Δ_3 ...
Observation Controller:
Creates temporal series of agent-states
Subject Agent:
Computes state transitions

Figure 4.1: Overview of observation components in Thinklab.

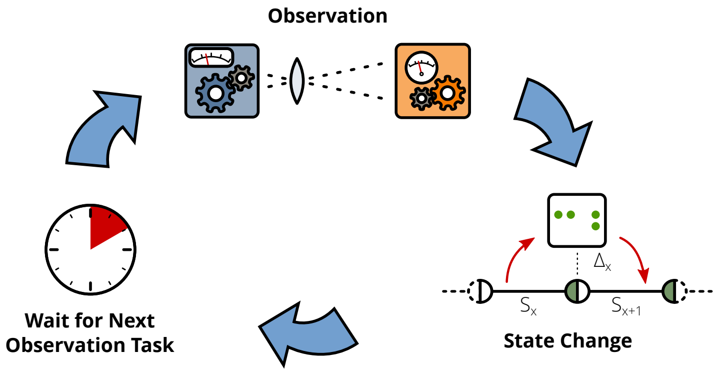


Figure 4.2: The observation and agent-state cycle in Thinklab

devoted to its inconsistency and directionality, and to addressing limitations it places onto phenomena and events (causality, speed of light, event horizons, etc). In our domain, time is irreversible; it cannot be reasoned with other than in the forward causal direction. As a simulation platform, Thinklab must handle taking steps that generate unpredictable results and emergent behavior that cannot be predicted in advance. This corresponds to the main goal of simulation in the first place: to show results of what cannot be proved or modeled mathematically. Therefore, the forward direction of time is an essential feature which is unique to time and does not exist in other dimensions such as space. Also, there are abstract and theoretical dimensions which we did not implement but we must allow for (see Section 3.5.3). These include the theoretical dimensions of string theory, the abstract dimensions of viewpoints and interpretation, etc.

The rest of this chapter is organized as follows: Section 4.1 discusses the semantic model we created to consistently and explicitly handle multiple scales and to allow agents to reason about scale and scale mediation; Section 4.2 covers how scale mediation strategies were incorporated into the system; Section 4.3 describes how agents' perception can differ subjectively, allowing agents with different characteristics to interpret the same environment differently; and finally, Section 4.4 explains the execution model which converts the semantic specifications into concrete computational steps during a simulation.

4.1 Semantics

Here we give the semantic model for observations in Thinklab. Some of the design here was already in place prior to the start of this project, but adaptations had to be made to accommodate the forward progression of time and agents' ability to reason about scale and scale translation.

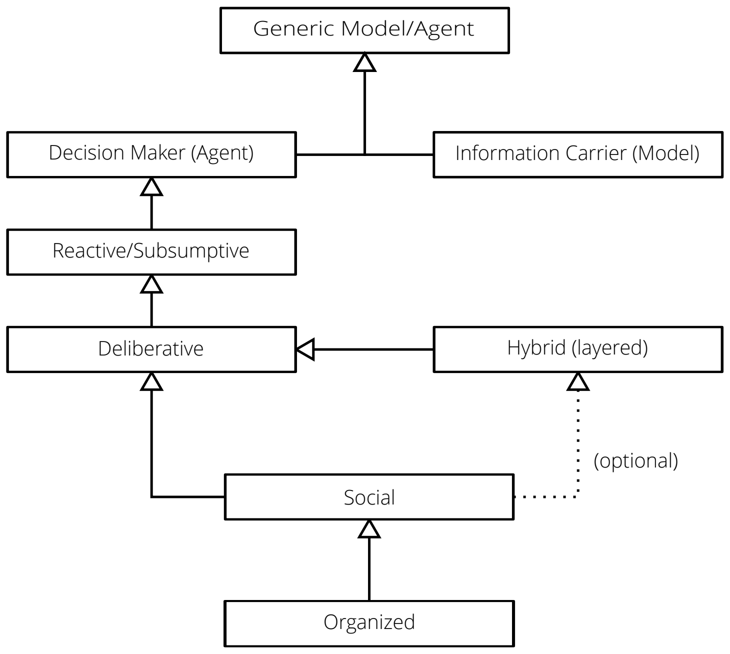


Figure 4.3: Agent Type Hierarchy for Thinklab

Because Thinklab is a simulation system based on agent observations, its execution flow is primarily centered around the cycle of agents' observations and subsequent state changes (see Figure 4.2).

4.1.1 Agent Types

This section is a summary of the Agent object in Thinklab, which is implemented as a hierarchy of Java classes and corresponds to the basic categories of agent types in literature (see Figure 4.3).

Agent types are described in Section 2.1.3, and the Java classes we implemented are described in Section 5.1.1.

In Thinklab, agents fall roughly into four categories that differ according to perceptual and reasoning abilities. They might (i) have no reasoning ability at all (information carriers), (ii) apply layered rules (reactive agents), (iii) have internal state and logic (deliberative agents), or (iv) have more complicated interaction/organization (social/organized agents).

Athanasiadis' *information carrier* agents [4] correspond to the messaging system planned for Thinklab and to models generated by field-collected data, neither of which can perform reasoning of their own. These types of agents act only as subjects (rather than observers) during a simulation.

Implementing specific classes of agents is done by extending one of these classes, or an instance of a class can be created as is. Agents in Thinklab are created by issuing an `observe` statement in the ThinkQL language that specifies the concept they should incarnate and any pre-defined initial state (such as their initial scales and locations in time and space). Models (defined with a `model` statement) can be defined independently and are paired to agents intelligently to provide them with behavior that reflects context and external conditions, and can change during the course of the simulation.

4.1.2 Agent Interaction

For communication, agent systems typically employ either a directory service to allow agents to find each other, or a blackboard system to allow agents to send messages in a more anonymous fashion. In Thinklab, the discovery system is merged in the abstraction of the *resolver*, a component that dynamically resolves dependencies stated in terms of ontological concepts to specific models (agent behaviors and state computation algorithms) which are applied to observe information for the requested concepts. The resolver will also provide agents with appropriate scale translation mechanisms, because it is the software component that will be aware of the native scales of the subject and observer agents.

Although many explicit agent interaction types exist in literature (see Section 2.1.4), we have not specifically designed any interaction protocols. We provide an API by which modelers can create interaction mechanisms, accessible both through Java and through the Thinklab modeling language, and we also provide a messaging system which fits into the ontological semantics of Thinklab. The messaging system treats messages as lightweight agents, because they fit our agent definition quite well: they are produced by other agents (a characteristic of agents which come into existence during a simulation), their properties are expressed in a native scale, they are temporally bound and possibly temporally dynamic (if they expire or degrade, for instance), and they can be observed subjectively by other agents by invoking the scale mediation mechanism in Thinklab.

4.1.3 Temporal Scale

Scale is an arbitrary-dimensional concept. Depending on the interpretation and the purposes, time, space, and other dimensions may or may not be treated differently. We made the decision that time is a unique dimension with different characteristics than spatial or abstract dimensions. Two characteristics are essential to time as we have incorporated it: 1) causality is one-directional; and

2) time is a single continuous dimension which can be evaluated in isolation from other dimensions. Neither of these two are essentially true at the core of physics, but for the purposes of a machine-based agent simulation, and according to the simplifications we have chosen for our model of physical reality (see Section 3.5), they hold consistently and have been essential for allowing us to implement Thinklab in a straightforward way.

Thinklab allows both irregular and subjective scales in all dimensions including time. Our temporal semantics align with Functional Reactive Programming (FRP) (see Section 4.4.5); specifically, we tried to adopt a model compatible with push-pull dynamics by distinguishing between the *discrete* and *continuous* components of real-world phenomena, and by distinguishing between times when meaningful events happen and times in which they do not.

Agents are only able to make decisions at specific points in time which we call *observation times*. These points are the equivalent of FRP's discrete *event* components of behaviors, and are the times when agents' decisions cause meaningful changes to *agent-states*. In agent perception, observation times do not represent discrete points in time: simply, time *does not exist* between them, as the perception of time make the time domain a continuous one in each agent. Therefore, between observation times, agents' observable state functions do not change, but the functions can be evaluated continuously along the intervals between points. These functions are the equivalent of the *continuous* components of behaviors in FRP, and represent periods of time when no meaningful changes are happening beyond what is expressed by the state functions.

This structure leads to an overall dynamic where each agent is allowed to proceed along a totally subjective temporal scale, making decisions at intervals appropriate for the agent, and changing state over time in both discrete increments and continuous intervals. These subjective time scales are synchronized by using globally-synchronous values for time to enable agent observation as described below.

4.1.4 Temporal Synchronization

We have decided to use a globally-synchronous timer to allow Thinklab to coordinate agents which may be unaware of each other's temporal scales. Each observation is made by an agent at a specific time, which is subjective to the agent and does not correspond to any other agent's time scale. This observation time is converted into a globally synchronous time value by Thinklab so that consistency between agents can be enforced (Figure 4.4). As agents make decisions and take action, the state changes that occur as a result of those actions are recorded with globally-synchronous time values.

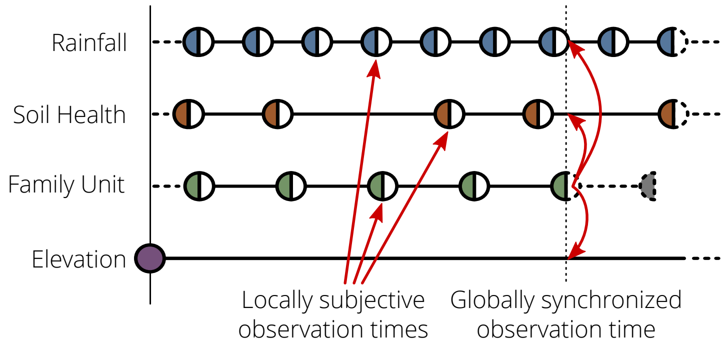

Rainfall
Soil Health
Family Unit
Elevation
Locally subjective observation times
Globally synchronized observation time

Figure 4.4: Agent observations using globally synchronous, locally subjective time scales

An agent may or may not know its position in space and time. The modeling of this is left to the mechanisms internal to each agent: all observable properties in the simulation are available at all times to all agents, but the implementation of the agent decides what is *perceived* of these properties (see Section 4.3). However, one restriction is enforced: only properties that have become valid *before* an observation time are observable *during* the observation (see Section 4.1.6).

Agents' schedules of temporal awareness are allowed to proceed by whatever time steps are deemed appropriate, fully disconnected from the globally synchronous simulation clock. Thinklab contains a mechanism as part of the agent API which transparently makes these conversions on the behalf of the agent, so that agents are not aware (through this mechanism) of the global time but live fully within their subjective experience of time.

4.1.5 Observation Times

An *observation* is made *of* a subject agent *by* an observing agent at a specific instantaneous *observation time*. All agents operate according to discrete, subjective time scales made up of contiguous intervals of arbitrary and possibly heterogeneous duration. Agents perceive their scales to be a continuous series of observation times, between which time does not exist, but in the system a time scale is represented as a series of intervals bounded by these discrete instants. Agents may make observations during each observation time that their scale puts into existence between the intervals. An observing agent can make any number of observations (including zero) of other agents during an observation.

After all observations implied by the model are made by the observing agent, the latter, when its semantics allows, has the ability to make a decision and act upon the world. As required by the use case scenarios in Section 3.4, decisions can be

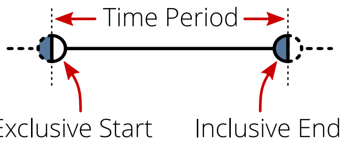


Figure 4.5: Exclusive-start, inclusive-end semantics for time periods

made in reference to simple observations (e.g. `move away if color == blue`), thresholds (`move away if income < 1000`), or aggregate functions (`move away if sum(population) > 100000`). Actions may change the states and scales of other agents besides the observing agent through what we have termed *collisions*, as a reference to the field of *collision detection*. For instance, a rock might hit and wake up a hibernating bear, who otherwise would have not experienced the next four weeks of time at all. State changes can only disrupt the states or scales of any other agents through collisions; by default, they only modify the observable properties of the agent making the state change.

4.1.6 Exclusive-Inclusive Time Periods

From the point of view of the system, agents operate by making observations and decisions at specific *observation times* according to their subjective temporal scale. Any observations made at an observation time will be based on agent-states which are valid *up to and including* the observation time, and the observable result(s) of any decisions made become active immediately *after* as they are made. Using normal interval notation, we would write these exclusive-inclusive periods of time as $(t_x, t_{x+1}]$.

This assumption reverses common practice for interval semantics, and for a good reason. It most closely resembles the real world, in which the effect of an action becomes "real" the (perceivable) moment after the action takes place. A direct effect of this interpretation is the avoidance of circular references, again exposing their fictitious nature due to the oversimplification of event occurrence in simulated systems.

The end time of the period of validity is not required; state(s) can remain valid indefinitely by setting an `INFINITE` end time. This kind of time representation is useful in simulating, for example, systems driven by real-time data acquisition, or an agent's expiration which will never be re-evaluated.

In our diagrams, we show each observation time as a circle with open and closed halves to illustrate the exclusive start (open half) of the state coming into existence and the inclusive end (closed half) of the state which is expiring (see Figure 4.5).

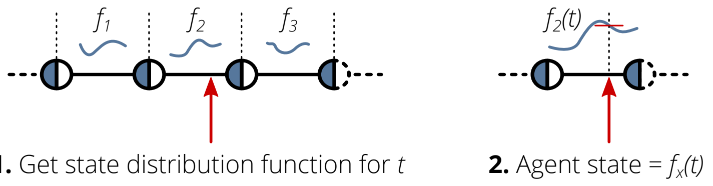


Figure 4.6: Evaluation of a subject's state may use deterministic or probability-distribution functions, rather than constant values

4.1.7 Agent-States

As agents proceed through time, their observable states change at specific points in time (observation times) and remain effective during the time periods bounded by the observation times. We refer to each time period as an agent-state time period, or simply *agent-state* for brevity.

An *observing agent* observes a *subject agent* by first identifying the *state distribution function* for the subject's desired observable property that is valid at the observation time, and then solving the function for the observation time t (Figure 4.6). The state distribution function expresses the value of an observable property of the subject agent in its native scale. After the state distribution function is evaluated, the result is converted into the observing agent's preferred scale and made subjective by applying the observing agent's distortion filters.

By mathematical definition, an agent A makes observations at times $t_x \in T_A$, each of which produces an agent-state function f_x in its agent-state series F_A. Agent-state functions express a probability distribution of values for each observable property of the agent; they are functions of both Time and $\mathrm{Property}$:

$$\begin{aligned} T_A &= \{t_1, t_2, \ldots, t_n\} \\ F_A &= \{f_1, f_2, \ldots, f_n\} \\ f_x &: (\mathrm{Time}, \mathrm{Property}) \to \mathrm{State} \\ \mathrm{state}(t, p) &= f_x(t, p) \text{ for } \max(x) : t_x < t \end{aligned}$$

Strictly speaking, $T_A \in \mathbb{R}$, but software implementations of time will generally introduce some level of discretization.

A state distribution function can be any distribution over a set of state functions of the independent time and space dimensions. The range of a state function is the domain of the property being observed. A state distribution function is capable of expressing a single constant value, a single path function (e.g. a ball moving through the air), a probabilistic distribution over constants, or a distribution over functions. Throughout this thesis, we refer to "observing an agent's state"; it is implied that the state distribution function is being observed.

This design fits with the concepts of *endurants* and *qualities* of formal ontologies. Endurants hold identity throughout the life span of an entity, independent of changes in state. Qualities are not entities of their own, but they describe entities and can be bounded in time. In Thinklab, agents are endurant entities and their agent-states are qualities that change over time. Agent-states are composed of any number of properties, which are attached to ontological concepts just like agents themselves are. It is possible that agents express different properties during different time periods; even if an agent expresses no properties for a time period, Thinklab will store an empty property-value mapping for that agent-state.

4.2 Mediation Between Scales

In this section we describe the design of the scale mediation system, which allows agents in Thinklab to observe one another despite their differing views of space and time.

The scale mediation system includes a strategy for selecting scale mediators, and the scale mediators are components which are inserted between an observer and a subject at the time the observer's dependencies are resolved. The scale mediator is composed of interpolation, intermediate representation, and output caches at potentially many scales. We have attempted to make the scale mediation system as modular as possible to facilitate module reuse, and we have also tried to make the scale mediation process as modular as possible to facilitate reuse of interpolated results.

4.2.1 Optimal Agent Scale

If a given agent operates at a given scale, then that agent will optimally make observations of dependencies which are at the same scale. The existing resolution mechanism in Thinklab does not take scale into account, although that is an intended upgrade being implemented outside the scope of this work. Currently, the resolution mechanism finds observable models that are capable of fulfilling requirements stated ontologically, and selects the model with the best coverage given the context of observation. The resolution mechanism in Thinklab will be able to prioritize models that are best suited to delivering the most appropriate scale and granularity, those with more efficient conversion computations over less efficient ones, and also it will be able to filter its selections based on both objective information and user-provided quality assessments. (See Section 4.2.3 for more information on mediator selection.)

This will allow the system to minimize computation associated with scale translations by using agents that share a common scale whenever possible, and using the best available mediation strategies when agent scales differ. Of course, this prioritization must be parameterized so that circumstances can dictate preference for speed, accuracy, coverage, availability of efficient mediators for the given domain, etc.

4.2.2 Intermediate Vector Representation

Many scale mediation algorithms inherently have an intermediate step that can be represented in vectorial form. For instance, linear interpolation is a technique that draws a straight line between each pair of neighboring points (in two dimensions; the equivalent three-dimensional technique is called triangulation, and connects neighboring coordinates by means of triangular surfaces). This type of scale mediation generates discretized points by first generating the linear functions, and then solving those functions at each discrete point.

Linear interpolation uses purely straight lines, but other types of vectors can be used as well. Some strategies generate spline functions, which are general-purpose, configurable curve definitions commonly used in vector graphics. Other strategies use domain-specific vector representations which can be mathematically more complicated but still provide desirable functionality for the scale mediator design we have chosen.

Mediation algorithms which inherently use some continuous intermediate representation have the advantage that these representations can be cached inside the mediator object and re-used for generating new observations of the same underlying phenomenon at different scales. Some algorithms produce interpolated data points without ever generating an intermediate representation; for instance, IDW (Inverse Distance Weighted) interpolation is a technique for “querying” a certain point for its observation value. As mentioned in Section 2.3.1, IDW generates data points $f(x, y)$ based on some number of input values $v(x_i, y_i)$ in the neighborhood of (x, y):

$$\begin{aligned} f(x,y) &\approx \frac{\sum_i w_i v(x_i, y_i)}{\sum_i w_i} \text{ where } w_i \propto (distance_i)^p \\ &\approx \frac{\sum_i \left(\sqrt{(x-x_i)^2 + (y-y_i)^2}\right)^p v(x_i, y_i)}{\sum_i \left(\sqrt{(x-x_i)^2 + (y-y_i)^2}\right)^p} \end{aligned}$$

All parts of the formula depend on the value chosen for (x, y), including which values $v(x_i, y_i)$ are chosen to represent the neighborhood of (x, y). To generate a concise representation, the boundaries where the neighborhood composition changes would have to be generated; this is at best non-trivial, and still the developer is faced with representational problems. Simply, when using IDW there is no concise intermediate expression of the set of all generated values, as there is in other techniques.

4.2.3 Scale Mediator Selection Strategy

Scale mediators can be categorized by many different features: those which have a continuous intermediate representation vs. those that do not; differentiable vs. non-differentiable; number of dimensions which can be handled by the scale mediator, $O(n)$ performance, etc. These can be required by the relationships between ontological concepts or by a user running or testing the system.

Requirements are attributed by using reasoning on the ontological concepts because it is the concepts that determine the mathematical characteristics of the feature being modeled. Requirements can be stated for specific mediators, classes of mediators, or arbitrary features for which mediators can be tagged.

A scale mediation strategy is selected for each relationship between two agents based on a layered preference model:

System Default Global default system-wide preferred scale mediator. This will most likely be the "nearest neighbor" algorithm because of its simplicity and performance.

Ontology-Relation-Specific Default Thinklab can be configured with specific mediation requirements for each ontological dependency relationship. This will be specified at the "relationship" level to accommodate domain-specific criteria. For instance, computing slope based on an elevation map may require that the mediated data set is differentiable; i.e. that dy/dx is continuously computable.

Runtime Parameters Thinklab could accept parameters on the command line to use specific scale mediators as overrides for the above selections. For instance, a simulation parameter could dictate that mediator x is the new system default, or that x is always used for translating from model M to N, or that mediator x is used in place of mediator y whenever it is requested.

Available scale mediators are filtered in the order Runtime $\rightarrow$ System Default. If at any stage in the selection process the list of qualifying mediators is empty, then Thinklab will select the *most appropriate* mediator from the previous list. We have not yet determined a good algorithm for determining the most appropriate mediator; one idea would be to throw out the selection criteria that made the list empty and continue without it. The algorithm should also require that the scale mediator is compatible with the number of dimensions and with the subject and observer scales.

4.3 Subjective Agent Perception

Real-world agents and their simulated counterparts will generally have finite and possibly inaccurate sensory input, and possibly systematic bias such as humans' inability to place certain sound frequencies in three-dimensional space. To allow different agent types to process observations differently, we chose to incorporate into the agent semantics a distortion mechanism, which is a collection of filters that can be applied to incoming information so as to distort, enhance, degrade, skew, stretch, or otherwise change it as it is processed by the agent.

Essentially, we have separated the act of *observation* into the external process of *information gathering* and the internal process of *perception*. In Thinklab, the external information gathering is performed by observing other agents' properties and converting them to the observer's scale by *scale mediation*; perception is internal to the agent. By doing so, we have emulated nature by differentiating between the outward natural phenomena and the inward interpretation of and response to the phenomena by interactive beings.

The perception filters can be as simple or complex as desired. For instance, it is possible to create an "ultimate" agent in Thinklab which observes every agent's properties in the simulation, over all time and space at the finest level of detail in the system. The only constraint is that observations occur at a specific instant in time and are not allowed to observe the future. This is not a semantic restriction to limit the power of ultimate agents, but rather a necessity of our implementation. Collision detection and dependency resolution would be impossible if we allowed future observations, and future observations are of no practical use in Thinklab anyway.

As an agent makes an observation, the necessary properties are collected from the appropriate subject and delivered to any filters which may be present for that agent. After the filters have been applied, the agent receives the observed property in its modified state and may then perform its reasoning.

4.4 Execution Model

The execution model of a Thinklab simulation is summarized in Algorithms 1 and 2. A Thinklab simulation begins with agent initialization. Agents are created (explicitly for the root agent, usually based on observed evidence from datasets for subordinate agents) and initialized with states that are valid during the instant of time that starts the simulation. States expire at varying times as appropriate for each agent's temporal scale. After all agents have been initialized, observation tasks corresponding to the expiration time of each agent-state are placed in an observation queue. Each agent-state expiration time is an *observation time* when the agent will make new observations and generate new agent-state distribution functions for the following time interval (see Section 4.1.5).

```
Data: $rootAgent$ with initial state; empty $temporalGraph$, $queue$, and
      $processing$ objects
Result: fully computed simulation
begin
    $InitializeAgent(rootAgent, queue, temporalGraph)$
    while $|queue| > 0$ do
        $task \leftarrow$ pop the next task from $queue$
        add $task$ to $processing$
        $result \leftarrow$ execute $task$
        remove $task$ from $processing$
        $agentState \leftarrow$ agent-state from $result$
        add $agentState$ to $temporalGraph$
        $relationships \leftarrow$ causal/influential dependencies from $result$
        for $relationship \in relationships$ do
            add $relationship$ to $temporalGraph$ according to type
        end
        $subsequentTasks \leftarrow$ additional tasks which result from $task$
        for $subsequentTask \in subsequentTasks$ do
            add $subsequentTask$ to $queue$
        end
    end
end
```

Algorithm 1: RunSimulation

Once the observation tasks are created and placed on the observation queue, the system starts a loop of computing agent observations and adding subsequent observation tasks to the queue. When all observations have completed and the observation queue is empty, the simulation is finished. Agents may follow a regular or irregular time scale; Thinklab is completely indifferent and is not actually aware of agents' time scales. Agents manage their time scales internally, and it is up to each agent whether it generates an entire schedule when it is initialized, or generates each successive observation time as it proceeds. Even if the scale is generated upon initialization and is completely deterministic (such as a soil quality agent with a daily or seasonal re-evaluation cycle), it reports each

```
Data: $agent$ with initial state; $queue$ and $temporalGraph$ objects
Result: fully initialized $queue$ and $temporalGraph$
begin
    $agentState \leftarrow$ initial agent-state from $agent$
    add $agentState$ to $temporalGraph$
    $task \leftarrow$ new observation task for $agentState$ expiration time
    add $task$ to $queue$
    $subjects \leftarrow$ observable subjects of $agent$
    for $subject \in subjects$ do
        $InitializeAgent(subject, queue, temporalGraph)$
    end
end
```

Algorithm 2: InitializeAgent

observation time to Thinklab in real time during the preceding observation.

Agent-states are not required to change at each observation time, but agents must explicitly re-declare their states and state expiration times, or Thinklab will consider the agent to be dead (see Section 4.4.1). Any agent-state which is requested by an observer agent but has not yet been computed will cause the observer agent thread to go to sleep. (Technically, the observer is given a *future value* to delay the sleep if possible, but if the observer calls `Future.get()` before the value is ready, the sleep is invoked.) Thinklab remains extensible so that agents can be optimized further if developers would like to implement more sophisticated sleep/wait semantics, team dynamics, etc.

Thinklab initializes agents with agent-states that have some temporal duration. The initial agent-state created for each agent applies to the agent's first temporal extent, according to its subjective temporal scale. Two other designs would have been possible: either we could have used non-temporal initial states and designed a completely separate path of execution that happens at the start of a simulation (surely an exercise in code duplication), or we could have initialized agents with temporal states of zero duration and added special-purpose conditions in the code to consider zero-duration states "valid" if they occur at the start of a simulation. Either technique would have led to special-purpose code which breaks the strict semantics we have chosen to simplify observations, and would have probably led to code duplication.

4.4.1 Agent Life Cycle

This section discusses the agent life cycle that happens during a Thinklab simulation. All agents will experience birth; they may or may not experience life

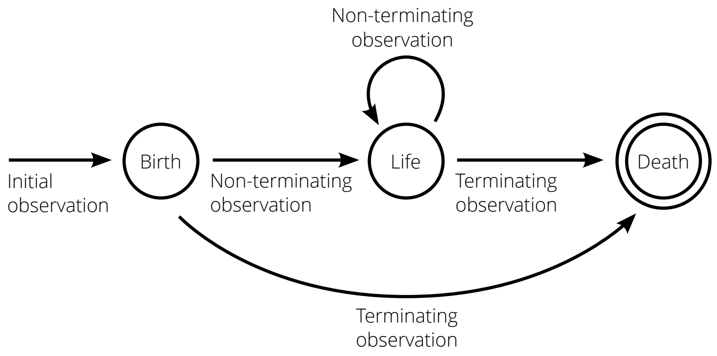


Figure 4.7: State-transition diagram of the agent life cycle

or death. (Agents in atemporal simulations will experience neither.) A state-transition diagram showing the agent life cycle is given in Figure 4.7.

Birth

Thinklab creates one agent as a result of an `observe` command. This agent is called the *root subject*. When the agent is created, Thinklab associates the best model it can find with it (or create a base model if none are found). The model associated by Thinklab to the root subject may introduce dependencies on other agents, and these may have other dependencies, and so on. If so, observations are triggered to resolve all dependencies to states and agents. All agents that are necessary to provide an initial observation of the root subject are created at the start of the simulation according to their start-up semantics. If Thinklab is unable to produce enough observations to maintain the root subject with semantic consistency, this constitutes a failure condition.

As the simulation proceeds, agents are allowed to create other agents. The most common use of this feature is for agents to send messages to each other (see Section 4.4.2), but other scenarios are possible as well.

As in other parts of Thinklab, agent creation can be a cascading effect: an agent that is created at runtime may have dependencies of its own that are not yet present, and therefore new agents are created as a result of observing those dependencies, and so on. Dependencies may be stated in a conditional manner that depends on states observed before, which may lead to the selection of a different model to complete the observations, so the results of the initial observation may constitute all the modeler is interested in. Hence even atemporal models can be used to accomplish significant modeling undertakings.

Life

Once agents have come into existence, they maintain contiguous agent-state values for all periods of time during their life span. The momentary boundaries between agent-state time periods are the points where Thinklab allows the agent to make observations and decisions, and take action accordingly. After this, Thinklab enforces that another agent-state time period is created which starts at the time that the previous one ends (i.e. at the observation time). No gaps will ever exist between agent-state time periods for an agent.

Death

One possible outcome of an observation/decision is that an agent dies. In this case, the agent would not create a new agent-state interval and no further observation tasks would be added to the observation queue for the agent. Because no valid agent-state would exist for that agent after its final observation time, other agents could not make any observations of it.

4.4.2 Messaging System

The messaging system in Thinklab is not an independent component but rather a protocol that leverages the resolution, observation, and collision subsystems. Sending a message consists of three steps:

Discover the recipient of the message using the resolution mechanism. Usually the message recipient will already be known by the sender at the start of a simulation via a resolved dependency, but the resolution mechanism can also be invoked at run time as a directory service for this purpose.

Create the message by instantiating a message object. Messages are lightweight agents which follow the same observational and temporal semantics as other agents.

Send the message by creating a specialized *collision* event. The collision system in Thinklab is a way of interrupting agents at times other than their self-imposed observation times, but also functions as a message delivery service. Agents are free to respond to messages (possibly changing their time scales and agent-state expiration times), or ignore them.

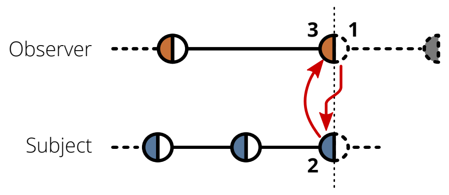


Figure 4.8: Cyclic dependency resolution using exclusive-start, inclusive-end time periods

4.4.3 Cyclical Dependencies

As noted in Section 3.1.1, *circular references* are different from *cyclical dependencies*. Circular references do not exist in the real world, but are an artifact of the ways models are represented. Thinklab provides safeguards against circular references in two ways: during agent initialization via static (atemporal) circular reference detection, and at runtime via exclusive-inclusive time period semantics.

The initialization-stage dependency resolver avoids circular references during initialization of the agents by throwing a `ThinklabCircularDependencyException`. This will occur when a cycle exists in the dependency graph for the agents in a simulation, where no agent-state can be initialized because of the cycle. (It is possible that a cycle exists which can be resolved by observing supplemental data sources. Thinklab will automatically attempt to resolve cycles in this way, and if successful, no exception will be thrown.) If a user runs a simulation which throws this exception, then the simulation stops and the parameters or models must be modified.

At runtime, the exclusive-inclusive time period semantics (Section 4.1.6) and temporal observation rules prevent any circular references from occurring (Figure 4.8). Temporal observation rules stipulate that as the simulation proceeds forward in time, all observations are made at a specific instant in time, and agent properties which are observed must be valid *during* that time. agent-states which are created or modified during the same instant are not valid until just *after* that instant.

Thinklab operates with a synchronous global event clock, which opens up the possibility of contradictions of causality in simulations. Without explicitly identifying the distance between agents and the *amount* of time between an action of agent A and the observation or impact on agent B, we use this technique as a way of enforcing that the impact on agent B must be at least *some minimal amount* of time after the action of agent A.

Because all computing environments operate with discrete values, all implementations of time will necessarily have some minimal granularity. Coupling this constraint with our exclusive-inclusive semantics means that any effect which A's action has on $\mathtt{B}$ must necessarily occur at least one granular unit of time after the action is made by A. (In the reference implementation of Thinklab, we use the `joda-time` library which uses a granularity of one millisecond.) If we did not impose these temporal semantics, then it would be as if temporal delays between cause and effect did not exist and agents were capable of influencing each other in simultaneous instants.

4.4.4 Collisions

Collisions may change agent-states after they have been computed. By the observation semantics we have chosen, no collision can ever change a state prior to the collision time, but it may cause an agent-state time period to end sooner than it had previously, and a new agent-state time period to begin at the collision time.

It is also possible that an entire agent-state becomes invalid if cascading invalidations occur. In this case, Thinklab would invalidate the entire affected agent-state, rather than partitioning it into the portion that remains valid before the collision and the portion which must be re-computed after the collision.

The collision/invalidation mechanism in Thinklab will cause any dependent observations to be re-computed in a cascading manner. To accomplish this, causality links are recorded for all observations. Agents must re-compute their agent-states by responding to the collision event, taking the new collision information into account. They are free to ignore the collision, generate a new agent-state for the partition using the original agent-state end time, or generate a new agent-state with an arbitrary end time. The last option requires that any observation task which had been queued for the agent is updated to reflect the new observation time.

For agent-states to *collide* they must *coexist*. No further restrictions can be made *a priori* by Thinklab, because we have not stated any limitations for causality other than exclusive-inclusive semantics. Also, there is no limit to our definition of "collision" beyond the requirement that agent-states overlap in time. Thinklab does not have any knowledge of communication or movement mechanisms which the agents may be employing, so agents which overlap in space or communicate over other kinds of information transfer may collide with each other in specialized ways about which Thinklab would know nothing.

Static (*a posteriori*) collision detection algorithms will probably not be a good class of solutions for our semantic model, because they are built on the assumption that object positions are re-computed at some frequency of time steps. Our model makes no assumption about the frequency of time steps, or their relation to agents' motion in space (or any other agent behaviors which may cause collisions). Rather, the task of collision detection in Thinklab can be described as discovering the collisions which happen *between* time steps. Because of this, it seems as if we are bound to using only the closed-form collision detection algorithms. We do not preclude the use of static techniques, but our semantic model is a much better fit for dynamic ones.

Collision detection algorithms will not be handled in this project, as their proper handling constitute an academic field of its own. Instead, we only provide the semantic model described here, within which collision detection must happen, and defer to this field for appropriate solutions. See Section 6.1 for more information.

4.4.5 Functional-Reactive Programming

We have taken inspiration from Functional Reactive Programming (FRP)[13, 15] in two ways. The first is by representing objects in the most natural way possible by using *continuous representation* and by not discretizing except as an output step when needed. The second is by separating the *discrete* and *continuous* components of temporal phenomena. (We also use a third concept in FRP, *future values*, but this idea originated elsewhere.)

One early implementation of FRP used Interval Analysis as a technique for optimizing computation; event detection is only performed during time intervals where events might happen, and time intervals where no events can happen are not analyzed.

Push-Pull FRP extends these optimizations by propagating data differently according to whether it changes discretely or continuously. Discretely-changing values are *pushed* to any component of the system which is affected (much like an Observer design pattern [18], or like event-driven programming), and continuously-changing values are *pulled* by the observer on demand.

Continuous Representation

There are two main benefits to using continuous representation semantics to model real-world phenomena: 1) continuous representation can more accurately describe the real world; and 2) when discretization is necessary, continuous representation provides flexibility in dealing with multiple scales of granularity. Accordingly, we have incorporated continuous representation into the execution model in two ways.

The first is by representing agents and agent transitions using continuous values when possible. In many cases, agents are data sets collected from the field and are inherently discrete; these agents are represented in Thinklab in their natural, discrete scale. But other agents generate their states and can use continuous values to represent themselves. These agents can also express their state transitions fluidly by reporting them directly, instead of reporting a new agent-state at each transition.

The second is by creating scale mediators (Section 4.2) which store a continuous intermediate representation of the agent they are connected to. The primary benefit for this design is when the scale mediators must present agent-states at multiple discrete scales. Instead of re-generating each scale from the original each time, the intermediate representation can be discretized into whatever scale is necessary. As mentioned in Section 4.2.2, many scale mediation algorithms inherently have an intermediate step which can be represented in a vector form. It is this vector form which is used as the continuous intermediate representation of the agent being observed.

Interval Analysis

Interval Analysis (IA)[52] is a technique used by one implementation of FRP to determine when events *will not* happen so that computation effort can be focused on analyzing when events *might* happen. In Thinklab, we have enforced well-defined times when events happen through our temporal event semantics, which removes the need for IA.

The classes of events in Thinklab are *observation times* and *collisions*, each of which can only happen at well-defined times. Observation times are always known *a priori* because agents are required to state their next observation time as a result of each observation (see Section 4.1.5); collision times can be known through collision detection (see Section 4.4.4) or through explicit collision creation, as is done in the messaging system (see Section 4.4.2). No other event times exist in Thinklab, so there is no ambiguity for which we need to optimize through analysis techniques such as IA.

Push-Pull FRP

Push-Pull FRP [15] is an implementation of FRP which takes more optimization details into consideration. It distinguishes between discrete *reactive values* and non-reactive, continuous *time functions*, two semantic methods of expressing time-varying values. Push-Pull FRP models real-world activity using *reactive behaviors* which are composed of these two time-varying components. A summary of the

In General	**Discrete**	**Continuous**
Event Discovery	"pushed" by actor	"pulled" by observer
Programming Style	imperative	functional
In FRP		
Concept Name	event	behavior
Value Name	reactive value	time function
Knowable *A Priori*	depends on context	fully knowable once generated
In Thinklab		
Concept Name	observation time	agent-state time period
Value Name	observation/decision	agent-state function
Knowable *A Priori*	event time only	fully knowable once generated
Consistency Requirement	required for valid agent-state	executed only on demand (inherently consistent)

Table 4.1: Push-Pull dynamics in Functional Reactive Programming and in Thinklab

differences between discrete and continuous is in Figure 4.1.

Our temporal observation and behavior semantics compose these discrete and continuous components into one cohesive model for agent-state changes, similar to FRP's reactive behaviors. We stop short of analyzing the similarities between agent-states and reactive behaviors, though, because of the differences between our implementations.

As in FRP, we must accept that some processes in our system such as run-time decision making are stochastic. What this means in Thinklab is that, at every observation time, an agent may potentially make a decision which affects its state in an unpredictable way. We cannot take shortcuts by skipping observation times where agents decide to be inactive, because we cannot know whether an agent will be inactive until that agent's decision computation has been performed. (We would like to relax this assumption; we leave possible implementation techniques to Future Work in Section 6.1.)

Without doing additional work on detecting when agent decisions can be skipped, we have designed our observation system with the assumption that all observations will necessarily be made. Our agent-state functions are flexible state representations, blending discrete and continuous time-varying values; in addition, we allow decision distribution functions. A full description of time-varying observable values is given in Section 4.1.7.

Push-Pull FRP and Thinklab both make use of *future values*, a way of providing a temporary representation for a value that is not yet knowable so that processing can continue until the value is absolutely needed. Because this is a standard technique in software engineering and did not originate with FRP, we leave its treatment to Implementation (Section 5.4).

Discrete Events

Discrete events generate what are called *reactive values* in FRP. Reactive values express things like user input, such as key press events, or other instantaneous events that modify a property to take on a new constant-valued state. These events are separated by arbitrary lengths of uneventful time. Discrete events can be expressed as a step function: they are an ordered series of occurrences, each of which is a constant value that becomes valid for a specific time interval. When the next value in the series becomes valid, the previous one becomes invalid.

Push-Pull FRP's treatment of discrete events focuses on detecting *when* they may or may not happen, and on the types of states they bring into existence. We model discrete events differently than FRP because we have the luxury of knowing the exact time of every agent observation and decision *a priori*. Besides this difference, the same properties of discrete events hold for Thinklab as for FRP. We do not need to perform interval analysis to optimize event discovery, but we do use discrete event semantics as a way of efficiently modeling agent behavior.

Continuous Behaviors

Continuous *time functions* in FRP express constantly-changing states which cannot be computed on an ongoing basis without introducing artificial discretization, and in general cannot be efficiently computed in a "forward" direction. Instead, it is better to evaluate time functions as needed.

Time functions allow for modeling continuous phenomena like motion through space which cannot be discretized without losing information. In Thinklab, continuously-changing states are modeled as *agent-state distribution functions*, or *agent-states* for brevity (see Section 4.1.7). Agent-states come into existence as a result of decisions made by agents during discrete *observation times*, which are the equivalent of Push-Pull FRP's *discrete events*.

4.4.6 Distributed Processing Considerations

In Thinklab, causality follows the forward movement of time; events can cause state changes and collisions, and observations can only depend on states which have become valid prior to the observation time. Agents can be created and destroyed according to events or thresholds.

Thinklab operates with a globally synchronized clock and observation graph, but it would be helpful from a system design point of view if agents' internal processes and localized interactions could happen without any global synchronization. This would allow for parallel implementation and would surely lead to better performance than with the overhead of total agent (i.e. process) synchronization.

An optimal implementation taking multiple scales into account would have agent scales which are completely decoupled from each other, such that there is no globally consistent scale at all. The final observation of the root subject which is requested by the user would inherently operate with its own scale which is presented to the user, but this should not impose any restriction on other agents.

A fully distributed approach would have two advantages over a serial one in two primary ways:

Flexible implementation options would allow distributed processing and the performance gains that come as a result

Simpler agents in terms of their semantic definitions and logistical overhead needed to run them.

With fully decoupled scales, spatio-temporal boundaries must be synchronized between agents, so that agents can interact with each other, and so that observable values can be integrated. This task depends only on a master agent with enough knowledge to position the various agent-states in time and space. Thinklab performs this task by allowing agents to proceed without requiring knowledge of a global schedule, and synchronizing their observations and behaviors as necessary.

One potential pitfall exists: the number of threads in a multi-threaded design must be greater than the number of potential sleeping threads, to avoid an infinite-wait scenario. The current design specifies that an observer's thread should sleep until all of its observable dependencies have been fulfilled; if all threads are occupied by waiting observers, then no dependencies will be met. (This is also a shortcoming of our single-threaded proof of concept, but we have avoided it by the simplicity of our tests; multi-threading capabilities are a strict requirement to avoid this scenario under the current design.)

We believe we have found a good balance between global consistency and amenability to parallelization in the Thinklab platform. More information on our global synchronization semantics can be found in Section 4.1.4.

Chapter 5

Implementation

Here we discuss the implementation of our design, and the engineering issues therein. We have implemented our design in the form of a complete API which integrates into the Thinklab modeling system, a substantially completed reference implementation of this API, and a proof of concept using Scenario 1 from Section 3.4.1. The API contains all components necessary to facilitate the designs described in Chapter 4, with three exceptions. First, we have not included convenience functions for creating and sending messages (although they can be manually created by agents using the technique described in Section 4.4.2); second, we have not incorporated scale mediator selection into the TQL language or by automated reasoning (although these changes will modify the API minimally if at all); and third, we have not implemented any agent perception filters because their treatment is outside the scope of this work.

UML diagrams of the most important Java classes implemented for this project are shown in Figures 4.3, 5.1, 5.2, and 5.3.

5.1 Semantics

In this section we describe the ways we implemented the semantic constructs described in Section 4.1.

5.1.1 Agent and Interaction Types

Agent types available in Thinklab at the time of this writing are *information carriers* (e.g. messages and non-dynamic model agents), *reactive agents* (rule-based agents), and *deliberative agents* (imperative-logic agents). While semantics is

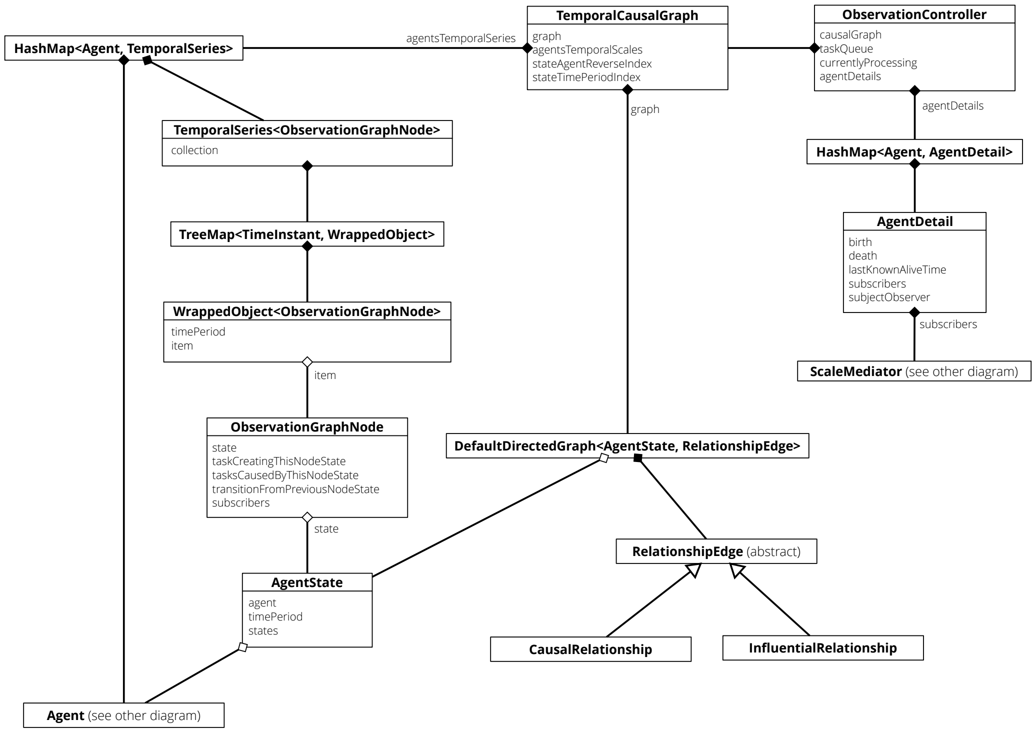


Figure 5.1: UML Diagram of the main Java classes in our implementation

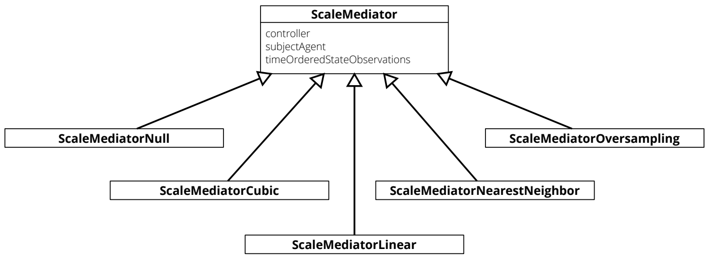


Figure 5.2: Inheritance structure of the Scale Mediators

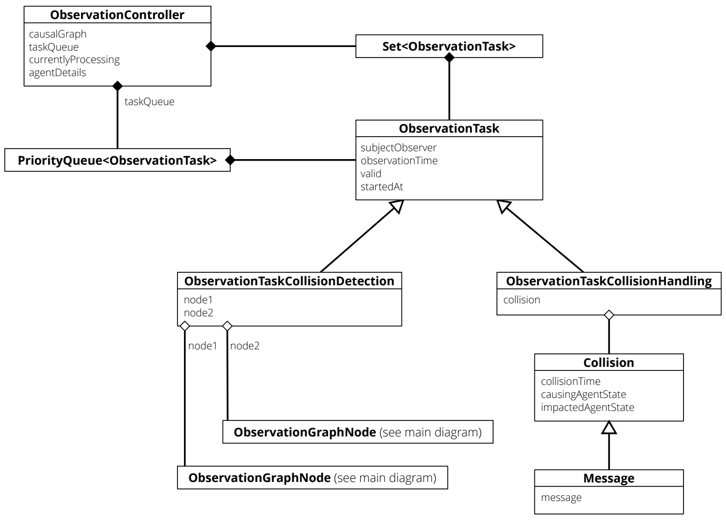


Figure 5.3: Tasks, Collisions, and Messaging components

available in the TQL language to specify these classes of agent, the definition of agent interaction patterns and protocols is left to the developer for the time being. For the purposes of this work, we capture the agent taxonomy as a standard Java hierarchy of Agent types, from which developers can derive agents aware of specific communication protocols, team-building strategies, etc.

An essential mechanism for any kind of agent interaction design is communication. We have developed a messaging component which is a simple re-use of the *collision* and *observation* paradigms designed earlier in this project (see Section 4.4.2). Because we have not incorporated agent coordination into our proof of concept, we have left messaging code and tests for later implementation. However, it is worth mentioning that this implementation only requires a wrapper function call consisting of a single line of code. A message is nothing more than a special collision object which is handled by both the receiver and by the message itself. (Specific message types with appropriate handling functions must be developed as well.)

5.1.2 Temporal Scale

The `ITemporalExtent` and `ITemporalSeries` interfaces were created during this project when it was decided that the temporal dimension had enough unique properties to warrant special treatment compared to a generic topology type. `ITemporalSeries` extends `ITemporalExtent`, which extends `IExtent`, which had already been a component of `IScale` before the work presented here was started.

The directionality of time is represented by a strict ordering of its sub-extents; `ITemporalExtent.getExtent(1)` should always return the contiguous extent that occurs immediately after the one returned by `ITemporalExtent.getExtent(0)`. One-dimensionality is expressed by the fact that the methods `getStart()` and `getEnd()` return scalar `ITimeInstant` values.

Temporal scales composed of contiguous time periods can be represented through instances of `ITemporalSeries`. This interface expresses the operations one would expect to perform on any contiguous linear series with specialized treatment for time. For instance, `ITemporalSeries.shorten(spliceTime)` takes only a single instant as a parameter, and it is assumed that the caller would like to shorten an overlapping time period and keep only the portion which falls *before* the instant. This is due to the forward-causality of time, and exists to support collision computations: only the events *after* a collision are affected and should therefore be discarded, not the ones before it.

Similarly, `ITemporalSeries.bisect(spliceTime, newObject)` splits a time period into two, and attaches `newObject` to the newly created period (the other one is only shortened, and keeps its originally attached object). As in `shorten()`, the newly created period in `bisect()` is always the one which occurs *after* the splice time.

5.1.3 Temporal Synchronization

To allow agents to operate at different temporal scales, we have implemented a `TemporalCausalGraph` class, containing a series of agent-states of arbitrary duration for each agent. At the end of an observation, an agent reports at what time its next observation will be. These observations are ordered and placed into a queue of `IObservationTask` objects for which the controller serves requests.

The temporal series (instances of type `ITemporalSeries`) stored by the `TemporalCausalGraph` for each agent are not required to synchronize with each other, other than being specified using absolute, globally synchronous time values. Agents themselves have perception filters which may imply that these globally synchronous time values have different meaning and implications for different agents, but the `IObservationController` is only made aware of the time values and does not consider any subjective meaning they may have for a given agent.

The `TemporalCausalGraph` is the single authoritative knowledge store for simulation results. Because it is maintained by the single-threaded `IObservationController`, synchronization is maintained in the system via the `IObservationController` API.

5.1.4 Observation Times

At run time, agents report the results of their observations by generating instances of the `ITransition` interface. Currently, agent-state information is retrieved from the `ITransition` objects using `ITransition.getAgentState()`.

One of the most important strategies for the temporal synchronization of observations is using globally synchronous observation times. As a result, the time values reported by agents at which their observations occur are in absolute-valued, globally-meaningful time values. The time value that one agent reports can be understood by another agent unambiguously to mean a single instant in time. In Thinklab, we use the UNIX time system, which expresses the number of seconds (and milliseconds) elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday 1st January, 1970.

Observations are synchronized as their results are reported to the Observation Controller. The Observation Controller maintains a single authoritative graph of observation times and the resulting agent-states; all associated time values share the same temporal measurement system.

5.1.5 Exclusive-Inclusive Time Periods

The agent-states generated by agents at each observation are a set of property-value pairs valid during some time period. These time periods follow exclusive-inclusive semantics, which we enforce by implementing the `ITimePeriod` (for single time periods) and `ITemporalSeries` (contiguous series of time periods) interfaces such that instances of them are aware of exclusive-inclusive rules. These interfaces extend `ITemporalExtent`.

The creation of a new agent-state happens at the instant of observation, and takes effect the instant after observation. Reading from existing agent-states (in other words, performing the sub-observations that are necessary for generating observation data and new agent-states) will only return states which are valid at that instant. To determine whether states are valid at a given time, the `ITimePeriod` methods `overlaps`, `contains`, `endsBefore`, etc. were implemented using exclusive-inclusive semantics. In most cases, calling code does not have to know what temporal semantics are being enforced. The methods are called using a time instant (`ITimeInstant`) or time period (`ITimePeriod`) parameter, and return meaningful values without needing to state their underlying semantics.

5.1.6 State Distribution Functions

Agent-states are composed of property-value pairs valid during a given time interval. The values are not required to be constant over the duration of the agent-state, or even to be scalar or categorical in nature. They can be functions of time, or they can even be distributions over some number of functions over time. The values contained in these property-value pairings are instances of the `IState` interface.

Probabilistic or statistical models can be created with agent-state functions that in place of single values express a probability density function or an explicit discrete distribution. The statistical or probabilistic values have not yet been implemented for any agents in our proof of concept, but could be done easily by implementing the `IState` interface, or extending an existing implementation. This interface has methods to store any type of Java object as an agent property and so is un-constrained in its ability to store complex values or functions.

Some implementations of `IState` (specifically, the subclasses of `IndexedState`) already accommodate arrays of values which match the cardinality of the scale to which the values apply. A similar approach could be used to store not just arrays of *dimensionally* distributed values, but *probabilistically* or *statistically* distributed values. Such values could be represented using special-purpose Java objects (for instance by using a generic class like `StatisticalDistribution<T>`), and stored the same as the scalar or categorical values already being used in Thinklab.

Evaluating a state-distribution function would be a simple matter of calling `IState.demote()` to return the bare state-distribution object, and then evaluating that object for the evaluation time using a call like `demotedValue .evaluateAtTime(t)`. Depending on how the usage evolves, this operation could be rolled into a helper method on the subclass of `IState`.

5.2 Scale Mediation

Scale mediators were implemented as modular components which can be dynamically selected at runtime based on the ontologically-defined semantics of the agents involved.

A simple example is an agent which computes rainfall runoff characteristics over some area of land by reading the elevation data for the area and converting it into slope values. In this scenario, a scale mediation strategy that preserves (or creates) differentiability in the surface data will be beneficial to the slope calculation. This benefit relates to the *relationship between concepts* of elevation and slope; it has nothing to do with the agent itself. Because of this, preferences and selection strategies are used which take ontological concepts into account.

The decision to use a decoupled mediator takes its inspiration from Dependency Injection [17] and the Strategy design pattern [18]: by decoupling an object from its strategy, and by leaving instantiation, initialization, and coupling to higher levels of the system, configuration can be made extremely flexible, allowing both modelers and developers to finely tune behavior, test and debug issues, and explore the dynamics of a simulation by focusing on broad or narrow behavior characteristics as necessary.

5.2.1 Caching Mechanisms

Push-Pull FRP takes advantage of the fact that *push* events are represented by pure data, and are therefore automatically cached in Haskell. Although this strategy takes advantage of mechanisms unique to Haskell, it applies equally well to our implementation. Haskell is able to automatically cache pure-data results not just because it is a well-designed language, but because pure data can *inherently* be cached to the extent that it is immutable.

In our environment, push events are triggered by agents making observations and generating agent-states for the subsequent time periods. We will store all agent-state results in a graph data structure (see Section 5.4.2), which will eventually be persisted to a graph database during the course of the simulation. (The persistence feature is not included in our proof of concept.) This graph, whether or not it is persisted physically, stores all agent decision and collision results. It will serve as the core of synchronization and will provide a full causal history of the simulation.

By intelligently building the components of this graph and the scale mediator objects which translate the states stored in the graph nodes, we have created an observation-task, agent-state, and cache invalidation mechanism which can reliably follow the trail of cached information throughout the system in the case that an agent-state must be altered after it is placed into the graph. The cache invalidation event occurs only when a *collision* is generated between agents (see Section 4.4.4).

Cached information in the system only represents *duplicated* information inside of the mediator objects. Mediators are allowed to keep two types of caches: *intermediate vector representations* and *output caches*. The latter follows standard software engineering practice as a method for reducing computational load, and has no features not present in the former. The former has unique characteristics and is the first cached element within a scale mediator, so we focus our attention on it.

Intermediate Vector Representation

Many scale mediation mechanisms use an intermediate representation of the source data which is continuous. As a result, they are capable of generating arbitrary discrete scales without the need to replicate the mediation process. Using this property, we can design mediation strategies which are inherently re-usable and more scalable than if the whole process was repeated from source data.

To facilitate building re-usable scale mediators, the `ScaleMediator` abstract parent class contains a `SubjectObservation` type which is used to store each node of agent-state information. This is a base type intended to be extended in subclasses of `ScaleMediator` to contain a member field `intermediateRepresentation`, used to store any intermediate vector representation necessary to support the specific scale mediation strategy. Re-computing a new scale after a cache has already been primed will incur only the cost of re-rendering the vector representation into the desired scale. Put simply, this cached intermediate result is part of what other scale mediation strategies actually *do*: at their heart they are just mechanisms to generate a continuous representation of the underlying properties. The final step of rendering this representation into a discrete scale is trivial and is essentially the same across all mediation strategies. We take advantage of this fact and of the fact that the continuous representation is cacheable.

Also included in `ScaleMediator` is a spatial index which uses the R-Tree data structure to facilitate spatial searching and overlap detection [20]. Its use is demonstrated in the `ScaleMediatorOversampling` subclass (still incomplete at the time of writing). To mediate from scale A to scale B, random points are generated within an extent of scale B, and those points are then superimposed upon scale A using the spatial index. This superimposition accomplished by searching the R-Tree of scale A for the coordinates of each point. Other scale mediation strategies will make use of spatial and other dimensional indexing, and R-Trees can adapt to arbitrary dimensions, so this spatial index was placed in the abstract parent class `ScaleMediator`.

Output Caches

When multiple observers observe a single subject, it is a common occurrence that they operate at the same scale; it is therefore best to cache results generated by mediators in a way that is accessible by other observers. This cache can take the standard form of $f : Signature \rightarrow Result$ where $Signature$ is a unique description of the function call by the requesting agent, and $Result$ is the result returned by that function call.

It is anticipated that these caches will become too large to store in RAM for the duration of a complex simulation; a flushing mechanism based on storage size, staleness, or the time values of the current observation tasks will be designed to address this problem.

5.2.2 Multivariate Scale Mediation Strategies (d-Dimensional)

Many interpolation techniques in the image processing canon are applicable to arbitrary dimensions. This allows these techniques to be applied to the problem of integrating multiple scales for arbitrary modeling tasks where then number of dimensions is not known *a priori*. Some techniques borrowed from the image processing domain are limited to two dimensions. A general modeling platform should ideally feature more flexibility than this, so they are not included here. They could be added to Thinklab using this same API under special conditions if deemed useful by a developer.

There are also many existing geospatial interpolation techniques (e.g. Kriging [34]), but even so, these algorithms might not be the best fit for our problem space, because the problem as we experience it is not that our data set lacks *samples* (as for example in the sparsity of high-altitude weather stations in the French Alps). Rather,the measured data is of a granularity which is incompatible with an observer. Our problem lies much closer to that of effective image re-sampling than to geospatial interpolation, despite the latter having been the primary domain in which Thinklab has been used so far.

Still, the rich field of geostatistical techniques can be mined for input into the domain-aware areas that we are developing. Specifically, the Application-Specific Approaches from Section 2.3.4 allow for domain knowledge to be applied as a final step so that vectors are selected from among many possibilities, and smoothed appropriately.

Java's Advanced Imaging (JAI) Library is the foremost candidate for general image manipulation and re-sampling [45]. It was considered as a reference implementation for some interpolation functions, given its stability, efficiency, and Open Source code base which could be re-used within Thinklab. But although the algorithms it implements are able to scale to d-dimensions its API is tightly locked into a two-dimensional world. For this reason, it cannot be used in its native state, although the option exists to closely mimic its data structures so as to re-use some of the code or optimizations contained in it.

This problem is true of most Java implementations of image manipulation algorithms. Java AWT, for instance, also contains many out-of-the-box resizing tools, but it is even more embedded semantically into its niche (creating user interfaces and displaying images) [60].

5.3 Subjective Agent Perception

We do not anticipate the development of perception filters to pose any serious challenge, but our proof of concept did not require their use. Their implementation will be a simple task of creating a callback structure within the observation process, in which agents can define filter objects. The filter objects will do no scale mediation and will have relatively loose constraints placed on their behavior. They will simply take `IState` values as parameters and produce filtered `IState` values. We have left this for future work.

5.4 Execution Model

In this section we describe the implementation of the execution model design outlined in Section 4.4.

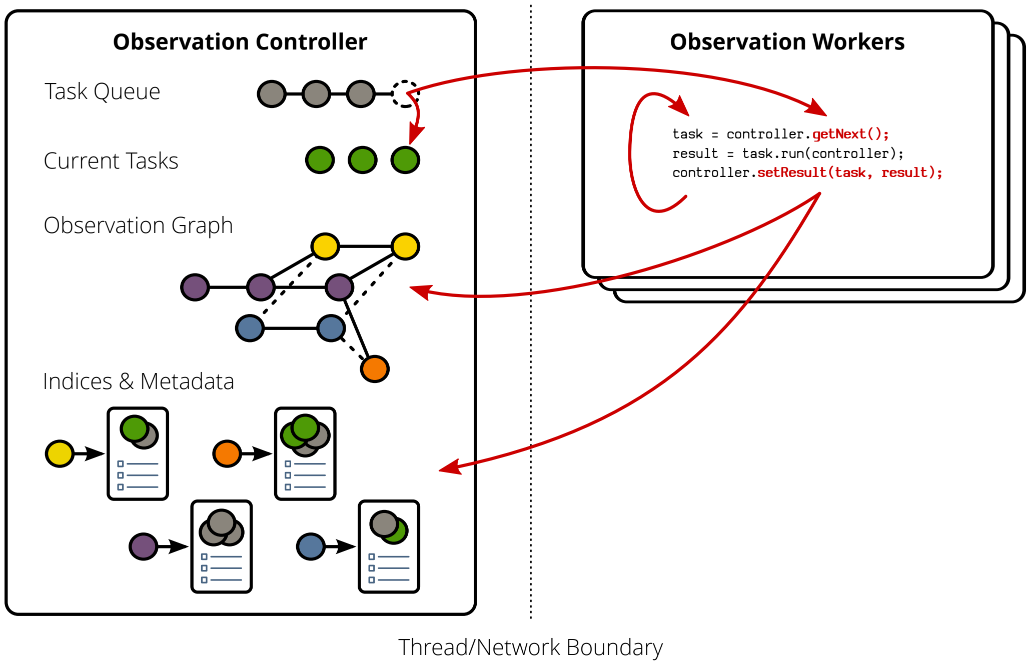
Because the proof of concept was written in a single-threaded style, there was no need for the use of *future values*, as mentioned in Section 4.4.5. The code already includes multi-threaded capabilities which were not tested, and incorporating future values into the multi-threaded code will be a simple matter of replacing the current command `Thread.sleep()` with a future of the value requested, which internally will call `Thread.sleep()` if it enters its blocking code path and no value is available.

5.4.1 Circular References

No special treatment was given for circular references beyond the use of exclusive-inclusive temporal semantics, whose implementation is described in 5.1.5. Atemporal simulations and the instantaneous initialization procedure which had already been a part of Thinklab had already had circular reference detection and avoidance mechanisms built-in, so the only challenge in this work was to avoid circular references at run-time for temporally dynamic simulations. Because exclusive-inclusive semantics provided this solution, no further work was needed.

5.4.2 Observation

We built a single-threaded proof of concept for this project, but the observation system is designed to allow multi-threaded execution (Figure 5.4). For each observation cycle, a worker thread pulls a task from the observation queue by



```
task = controller.getNext();
result = task.run(controller);
controller.setResult(task, result);
```

Figure 5.4: The observation system is composed of a single-threaded controller plus worker threads which perform the observation tasks

requesting it from the observation controller, performs the task, and returns the result by calling `ObservationController.setResult(task, result)`.

The controller processes the result by storing a new agent-state in a causal *observation graph* (Figure 5.5), a graph structure indicating causality relationships. The observation graph records both *causal* relationships which represent how agent-states were created, and *influential* relationships which indicate how agent-states influence each other during observation.

The controller also maintains a simple *task queue* organizing the observation tasks by the order in which they must be processed. As tasks are pulled from the queue and given to worker threads for processing, they are added to a collection of *currently processing* tasks. This will allow a multi-threaded version of the system to detect and mitigate a task execution failure, and also allows tasks to be invalidated while they are being processed (see Section 5.4.3). Finally, the controller stores metadata about each agent and indices for navigating between agents, tasks, and agent-states.

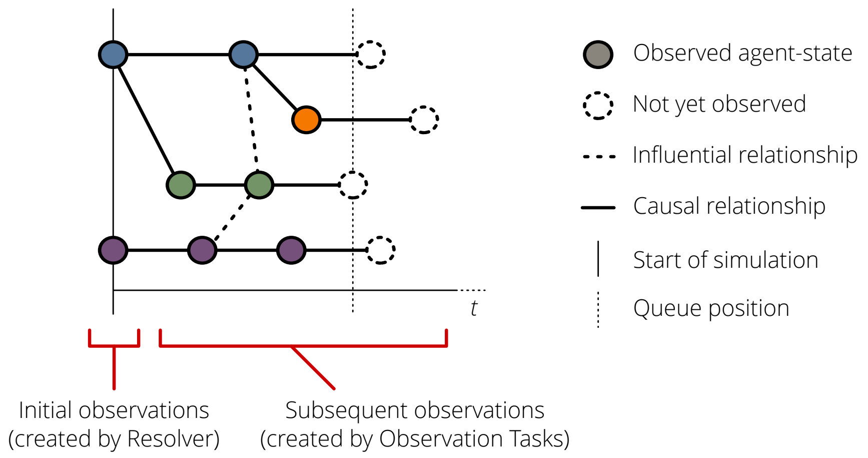


Figure 5.5: Observation graph used for storing agent-states and their relationships

5.4.3 Collision Detection and Invalidation

Collision detection is a loose term we use to describe computing when agents interact with each other outside their normal observations, decisions, and agent-state changes. Collision detection is done using `ObservationTaskCollisionDetection` tasks, which are generated by `ObservationController.setResult()` after a normal observation task has been completed.

Because collisions may affect previously created agent-states, the `IObservationController` and `IScaleMediator` interfaces have an invalidation mechanism to maintain accurate, temporally synchronous states and caches.

Collisions are not restricted to be physical in nature (as is the case in most literature, generally in the context of video games, interactive animations, or physics-based simulations). Collisions can happen as a result of active decisions by agents, for example in message passing; collision *detection* refers to the detection of inadvertent interruptions which are not an explicit act of an agent.

Collision Detection

Collision detection is done by specialized task objects (instances of `ObservationTaskCollisionDetection`) which are created at the conclusion of every agent-state observation task. The controller iterates through all overlapping agent-states and creates one collision detection task for each pair of agent-states (Figure 5.6). These tasks are executed by worker threads in the same manner as general observation tasks. If any collision detection task results in a collision, the `ObservationTaskCollisionDetection` object calls `ObservationController`

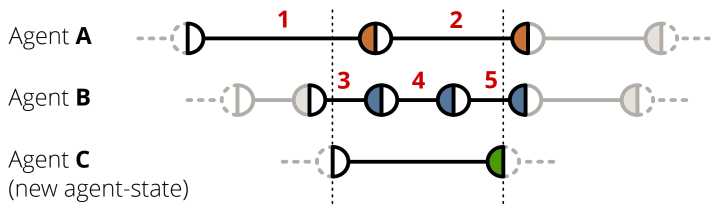


Figure 5.6: Collision detection is done for all overlapping time periods

`.collide(node1, node2, collision)`, where `node1` and `node2` refer to nodes in the temporal causal graph maintained by the observation controller.

In `ObservationController.collide()`, the controller asks each agent to determine whether or not the collision affects it. Both agents for every overlapping pair are interrogated; this is because they belong to different types and may use different logic to determine whether collisions are important. One agent may be inanimate with respect to collisions (such as a water table model) and therefore always return `false`, whereas an agent interacting with it may return `true`, for instance if an agent falls into a well and hits the water as a result of the water level reported by the water table agent.

The proof of concept we created does not implement the various collision detection algorithms or criteria. We have created the detection, invalidation, and handling procedure outlined here, but the actual work of detecting collisions is beyond the scope of this project. We believe that the API is a good interface for collision detection logic, as it provides a simple callback mechanism where detection algorithms can be introduced.

The field of collision detection is broad, even the problem is stated differently in different contexts. For each form of the problem, many strategic approaches exist [30]. But even these physics-based definitions are more restricted than our collision detection system, because we leave the possibility of abstractly-defined collisions open. Concrete collisions may represent a loud noise that disturbs an agent, a flash in the background of an agent's visual field, unexpected distortion from a static-electric field, etc. For now, we leave our API intact with the possibility to expand via standard collision detection strategies as well as any domain-specific collision detection that may be appropriate.

Agent-State and Task Invalidation

The observation graph supports invalidating previously made observations through a time-based invalidation procedure (Figure 5.7). If an agent reports that a collision results from two agent-states, then each agent involved in the

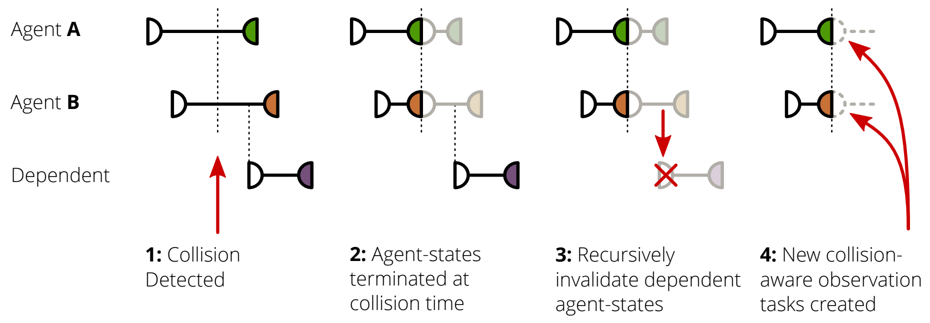

Agent **A**
Agent **B**
Dependent
1: Collision Detected
2: Agent-states terminated at collision time
3: Recursively invalidate dependent agent-states
4: New collision-aware observation tasks created

Figure 5.7: Invalidation of observation results during collision detection

collision is asked whether or not the collision affects its agent-state. If an agent responds with `true`, then the invalidation procedure is triggered.

Observation tasks maintain a causal relationship with the agent-states that initiated them. For instance, an agent a which remains alive beyond agent-state a_t creates an observation task to observe agent-state a_{t+1}. In this way, when a collision invalidates an agent-state in the observation graph, any subsequent observation tasks which resulted from the invalidated portions of agent-states are also invalidated.

The observation graph treats *causal* and *influential* relationships differently. An example is a child who goes to a certain school for some period of time: if by processing a collision, the simulation determines that the school never existed, then the child would be updated to go to a different school. But if it is determined that the child's parents never existed, then the child would be deleted from the simulation. The differences in the handling procedures are explained below.

The collision is reported to the observation graph, along with the affected agent-state. The observation graph shortens the agent-state so that it ends at the collision time, and also determines which other agent-states had a causal (as opposed to influential) dependency on the portion of the original agent-state which has now become invalid. This is done by examining all causal links in the graph which originate at the agent-state being invalidated; any of the dependent agent-states which begin after the collision time are removed from the graph. This is done in a recursive manner, so that any states which are in turn causally dependent on the ones being deleted are also deleted.

Influential relationships are treated differently. Any agent-states in the graph which have an influential dependence on an agent-state which is invalidated must be re-evaluated, rather than deleted. To accomplish this, a mapping between each agent-state and the observation task which led to its observation is kept by the observation controller. A task can be re-executed at any time by querying this map for the agent-state which must be re-evaluated. Executing the task is as simple as re-adding it to the observation queue.

Once the agent-states have been shortened and the invalid dependents removed, a new *collision handling* observation task is created so that the agent can decide how to handle the collision. This task contains a reference to the collision object.

Invalidating observations which have been queued but not yet observed is done by setting a flag on the observation task object. If the next task from the observation queue retrieved by the observation controller has been invalidated by a collision, the task is discarded and the next task is retrieved from the queue. If a result is returned to the observation controller whose original task object has been invalidated, the result is discarded.

Chapter 6

Conclusion/Future Work

In this thesis we have presented a novel way of decoupling the particular scale at which agents operate from the simulation system and from the scales of other agents with which they interact. During the course of our work, the unique characteristics of time have compelled us to give it special treatment, and even to hard-code some of its unique characteristics directly into our reference implementation, which has not been necessary for space or for other dimensions.

The library that we have generated could also be a generalized platform for adding raw data processing (e.g. Kriging) techniques into a system, such that field data can be used directly without any pre-processing. The pre-processing and data cleansing algorithms operate under the same assumptions and share the same task as our own Scale Mediators: generating data sets in a specific scale based on some data source values that potentially adopt a different scale.

Extending a system to include this level of data processing would allow e.g. ecosystem modelers to work more efficiently, speeding up the job of ecosystem simulation. It is possible that this type of update would require metadata, including for example estimates of any loss of accuracy due to scaling, to be generated in addition to the raw data output – a simple enough modification to make in our Scale Mediator API.

6.1 Future Work

Throughout this thesis we have mentioned areas that deserve further exploration. Here we revisit them as a guide for other researchers as well as to hint at our own upcoming development efforts.

6.1.1 Collision Detection

We have not yet incorporated any collision detection algorithms into our code. The collision detection system exists, and collisions can be explicitly created if desired, but no agents have yet been designed to detect collisions in real time. Using the `RTree` class which is already used for spatio-temporal indexing in the `ScaleMediator` class would allow agents to provide collision detection relatively easily; a global `RTree` instance would probably suffice. Still, this functionality was not needed for our proof of concept. Many collision detection techniques and algorithms already exist, and we have designed our API in a way that should conveniently interface with pre-existing code.

6.1.2 Other Interpolators

Our proof of concept included only a single scale mediation mechanism, which was adapted from the "oversampling" image interpolation strategy. Many other strategies exist, some of which were mentioned in this thesis. Their implementations should be straightforward, and in many cases pre-existing libraries will exist that can be adapted to Thinklab. However, many pre-existing libraries are highly focused on specific domains of application, and may require rewriting if they are to work with another platform or with arbitrary scales or dimensions. Java's Advanced Imaging library, for instance, is well established but would require significant work for the algorithms to be adapted (see Section 5.2.2).

6.1.3 Intermediate Representation and Differential Transitions

Because the proof of concept includes an interpolator based on the oversampling technique, and because oversampling does not require an intermediate representation, this technique was not implemented during this project. The mechanisms do exist, as outlined in Section 4.2.2, but no scale mediators have yet been developed which take advantage of this feature.

When implementing this feature, it will also be advantageous to implement the differential transition mechanism in the `ITransition` interface. Eventually, `ITransition` will support methods to dynamically update pre-existing cached data rather than replacing it with newly generated data. This will improve the scalability and accuracy of the results delivered to agents at different scales. This approach is described in Section 5.1.4. Applying transition differentials to vector representations of objects is much more accurate than doing so through discretized representations of those objects, and applying differentials to either type of representation is generally more efficient than re-computing results from scratch.

6.1.4 Optimization by Skipping Observations

In section 4.4.5 we mentioned that we cannot take shortcuts by skipping observation times where agents decide to be inactive. However, it is not difficult to imagine that we could detect cases where observations could be skipped, and postpone the appropriate observation task in the Observation Controller until the next observation time. This undertaking would include two tasks: identifying the scenarios under which observation tasks could be delayed, and implementing the mechanism by which they are delayed. The first will most likely result from using the system and paying attention to the patterns of high resource usage by agent type, and the second will be trivial once progress has been made on the first.

6.1.5 TQL Language Changes

For the proof of concept, we created test cases using JUnit 4.0 and a `ModelProxy` object to drive the resolution and observation processes. By doing so, we were able to avoid the need to update the TQL language with all necessary features. Most were updated (the models we used in our tests were defined in TQL) but we did not implement a way of defining ontology-driven scale mediator selection processes or unbounded temporal scales. A rough outline of the scale mediator selection syntax has been started at the model level, but this syntax needs to be fully defined, implemented, and updated so that it relates to ontological concepts rather than directly to models. Unbounded temporal scales do not present any technical challenge; our proof of concept did not strictly require them so they were not prioritized for.

6.1.6 Deliberate Simplifications

Many deliberate simplifications were mentioned in Section 3.5. In cases where a more highly tunable or domain-specific application is needed, some of these limitations may need to be addressed. The most likely candidate is our current treatment of four-dimensional space and time as the limiting bounds of reality; although this is not a hard limitation, we have not yet tested Thinklab beyond this usage model.

Chapter 6: Conclusion/Future Work

Bibliography

[1] Robert John Allan. Survey of agent based modelling and simulation tools. Technical report, Science & Technology Facilities Council, 2010.

[2] Ilkay Altintas, Adam Birnbaum, Kim K Baldridge, Wibke Sudholt, Mark Miller, Celine Amoreira, Yohann Potier, and Bertram Ludaescher. A framework for the design and reuse of grid workflows. In *Scientific Applications of Grid Computing*, pages 120–133. Springer, 2005.

[3] Ioannis N Athanasiadis. A review of agent-based systems applied in environmental informatics. In *MODSIM*, pages 1574–1580, 2005.

[4] Ioannis N Athanasiadis and Pericles A Mitkas. A methodology for developing environmental information systems with software agents. In *Advanced Agent-Based Environmental Management Systems*, pages 119–137. Springer, 2009.

[5] Stefano Balbi and Carlo Giupponi. Agent-based modelling of socio-ecosystems: A methodology for the analysis of adaptation to climate change. *International Journal of Agent Technologies and Systems (IJATS)*, 2(4):17–38, 2010.

[6] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Software-Practice and Experience*, 31(2):103–128, 2001.

[7] Matthew Berman, Craig Nicolson, Gary Kofinas, Joe Tetlichi, and Stephanie Martin. Adaptation and sustainability in a small arctic community: Results of an agent-based simulation model. *Arctic*, 57(4):401–414, 2004.

[8] Michael Bratman. Intention, plans, and practical reason, 1987.

[9] Cosmin Carabelea. Agent architectures. 2004.

[10] Kathleen M Carley and Les Gasser. Computational organization theory. *Multiagent systems: A modern approach to distributed artificial intelligence*, pages 299–330, 1999.

[11] Carl de Boor. *A practical guide to splines*, volume 27. Springer-Verlag, 1978.

[12] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[14] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 421–434. ACM, 1994.

[15] Conal M Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[16] Christer Ericson. *Real-time collision detection.* Taylor & Francis US, 2005.

[17] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`, January 2004.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening ontologies with dolce. In *Knowledge engineering and knowledge management: Ontologies and the semantic Web*, pages 166–181. Springer, 2002.

[20] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[21] Aaron Helsinger, Michael Thome, and Todd Wright. Cougaar: a scalable, distributed multi-agent architecture. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 2, pages 1910–1917. IEEE, 2004.

[22] David Hiebeler. The swarm simulation system and individual-based modeling. 1994.

[23] Mike Holcombe, Simon Coakley, and Rod Smallwood. A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European Conference on Complex Systems*, 2006.

[24] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2004.

[25] Nicholas R Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[26] Abdul J Jerri. The shannon sampling theorem—its various extensions and applications: A tutorial review. *Proceedings of the IEEE*, 65(11):1565–1596, 1977.

[27] Pablo Jiménez, Federico Thomas, and Carme Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[28] KE Kerry and Kenneth A Hawick. Kriging interpolation on high-performance computers. In *High-Performance Computing and Networking*, pages 429–438. Springer, 1998.

[29] Johannes Kopf and Dani Lischinski. Depixelizing pixel art. In *ACM Transactions on Graphics (TOG)*, volume 30, page 99. ACM, 2011.

[30] M Lin and S Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA Conf. on Mathematics of Surfaces*, 1998.

[31] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. Mason: A new multi-agent simulation toolkit. 2004.

[32] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation*, pages 2–15. Winter Simulation Conference, 2005.

[33] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation part 2: how to model with agents. Winter Simulation Conference, 2006.

[34] G Matheron. Kriging, or polynomial interpolation procedures. 1967.

[35] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. 2004.

[36] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. 1996.

[37] Lubos Mitas and Helena Mitasova. General variational approach to the interpolation problem. *Computers & Mathematics with Applications*, 16(12):983–992, 1988.

[38] Lubos Mitas and Helena Mitasova. Spatial interpolation. In Paul A. Longley, Michael F. Goodchild, David J. Maguire, and David W.Rhind, editors, *Geographical Information Systems: Principles, Techniques, Management and Applications*, chapter 34, pages 481–492. Wiley, 1999.

[39] Helena Mitasova and Jaroslav Hofierka. Interpolation by regularized spline with tension: Ii. application to terrain modeling and surface geometry analysis. *Mathematical geology*, 25(6):657–669, 1993.

[40] Helena Mitasova and Lubos Mitas. Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical geology*, 25(6):641–655, 1993.

[41] NASA. Semantic web for earth and environmental terminology (sweet). `http://sweet.jpl.nasa.gov/ontology/`, February 2013.

[42] Michael J North, Nicholson T Collier, and Jerry R Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 16(1):1–25, 2006.

[43] MJ North, E Tatara, NT Collier, J Ozik, et al. Visual agent-based model development with repast simphony. Technical report, Argonne National Laboratory (ANL), 2007.

[44] Hyacinth S Nwana. Software agents: An overview. *Knowledge engineering review*, 11(3):205–244, 1996.

[45] Oracle Corporation. Java advanced imaging (jai) api. `http://www.oracle.com/technetwork/java/javase/tech/jai-142803.html`, May 2013.

[46] Dawn C Parker, Steven M Manson, Marco A Janssen, Matthew J Hoffmann, and Peter Deadman. Multi-agent systems for the simulation of land-use and land-cover change: a review. *Annals of the Association of American Geographers*, 93(2):314–337, 2003.

[47] Lael Parrott and Robert Kok. Incorporating complexity in ecosystem modelling.

[48] Joseph Gerard Polchinski. *String theory*. Cambridge university press, 2003.

[49] I Prigogine. *Introduction to thermodynamics of irreversible processes*. Interscience Publishers (New York), 1968.

[50] Stuart J Russell and Devika Subramanian. Provably bounded-optimal agents. *arXiv preprint cs/9505103*, 1995.

[51] SICS AB. Trading Agent Competition. `http://tac.sics.se`, August 2013.

[52] John M Snyder. Interval analysis for computer graphics. *ACM SIGGRAPH Computer Graphics*, 26(2):121–130, 1992.

[53] Swarm Development Group. Swarm simulation system. *Natural Resources and Environmental Issues*, 8(2), January 2001.

[54] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *in International Conference on Complex Systems*, 2004.

[55] Ferdinando Villa. Semantically driven meta-modelling: automating model construction in an environmental decision support system for the assessment of ecosystem services flows. In *Information Technologies in Environmental Engineering*, pages 23–36. Springer, 2009.

[56] Ferdinando Villa, Ioannis N Athanasiadis, and Andrea Emilio Rizzoli. Modelling with knowledge: A review of emerging semantic approaches to environmental modelling. *Environmental Modelling & Software*, 24(5):577–587, 2009.

[57] Ferdinando Villa, J. Luke Scott, and Ioannis N Athanasiadis. Thinklab API. `https://bitbucket.org/ariesteam/thinklab-api`, September 2013.

[58] Ferdinando Villa, J. Luke Scott, and Ioannis N Athanasiadis. Thinklab Reference Implementation. `https://bitbucket.org/ariesteam/thinklab`, September 2013.

[59] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[60] John Zukowski. *Java AWT reference*, volume 3. O'Reilly, 1997.

Chapter 6: BIBLIOGRAPHY

Glossary

agent Possibly the most important term in this thesis, *agent* refers to a being in the real world which is able to interact with the world. Some agents are simply observed phenomena, such as a message being passed from one agent to another. This message *agent* cannot reason or make decisions. It may be passed electronically and wirelessly and have no physical manifestation at all, and its temporal duration may be so small that no agent in the simulation can measure it. Still, because a message can interact with the world, it is considered an *agent*. Other agents may include weather patterns, vegetation growth, soil quality, family units, larger population groups, governments, etc..

agent-state One segment of time in the life of an agent. Agents' lifetimes are segmented by decision times; between decision times, their state functions do not change. These time periods are called *agent-states*..

collision A general term we use to refer to changes *forced upon* agents in the system which are not a result of their own internal decisions. Collisions are not necessarily physical impacts (or even physical at all); the term is a reference to the field of *collision detection*, which is the closest related field from which we can draw inspiration. Collisions include deliberate interruptions such as messages being sent from one agent to another..

observation The most important act that happens in the system is that of observation. Thinklab is a semantic meta-modeling system composed of agents which observe each other over the course of a simulation. In the context of observation, the observing agent is known as an *observer*; the subject of the observation is known as the *subject*..

observation time At each time step of an agent's *Subjective Time Scale*, any agent which is a *Decision Maker Agent* (see Section 2.1.3) will have the choice to make a decision. It may be the case that the decision leads to no action, but that is up to the agent itself; the system will trigger the appropriate decision making procedure on an agent at every one of the agent's internal time steps.

perception In Thinklab, an agent is considered to exist in a complete, synchronous, real world, and agents are given full access to all observable properties. An agent is free to internally filter the properties so as to create a limited *Perception* of the world around it. *Example:* Say a mountaineer agent and a full three-dimensional mountain model exist in a Thinklab simulation. At runtime, the mountaineer agent will be given the three-dimensional mountain model, but after applying internal filters, the agent experiences only a two-dimensional visual image of the mountain, which is appropriate for a human which sees in the visible spectrum. This visual image is its internal *perception* of the mountain.

root subject The primary observable agent in a simulation, for which all dependent observations are generated.

scale mediator Scale mediators are the objects in Thinklab which allow observations to occur between agents whose views of space and time differ. Agents may observe space as a series of grid cells, polygons, points, etc. and they will generate observable phenomena according to their subjective views. Other agents are able to observe these phenomena by employing scale mediators during observation..

state distribution function a probabilistic distribution over all state functions which could be expressed by a subject agent during one time period according to the subject agent's subjective internal time scale.

subject An agent which is being observed in the context of the discussion. *Example:* Soil quality which is measured by a farmer on a periodic basis is a *Subject*.

subjective scale A view of space and/or time according to an agent's subjective experience of the world. *Example:* An agent representing an animal which hibernates may have time steps totally different from one who does not; these would have differing *subjective scales*..