An Evaluation of Complex Adaptive Evolvable System Simulation

Kevin E. Anderson

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science and Systems

University of Washington

2016

Committee:

George Mobus

Wayne Wakeland

Chris Marriott

Ka Yee Yeung

Program Authorized to Offer Degree:
Institute of Technology - Tacoma

University of Washington

Abstract

An Evaluation of Complex Adaptive Evolvable System Simulation

Kevin E. Anderson

Chair of the Supervisory Committee:
Associate Professor George Mobus
University of Washington, Tacoma, Institute of Technology

Building models has been a human activity for centuries. Through the ages man has discovered patterns in how objects function together in the exchange of information, materials, and energy. We know these groups of objects and how they behave as *systems*. As we study systems, we have learned that systems can be extraordinarily complex. Additionally, complex systems can be adaptive. Computer software exists to help us design, simulate to ultimately understand these Complex Adaptive Systems. However, there exists a type of system that has not been as thoroughly explored. These systems are known as Complex Adaptive Evolvable Systems (CAES). In this thesis I describe what makes these systems unique, and provide a list of properties a simulation tool should possess to successfully design and simulate such a system. I then evaluate several popular modeling platforms (StarLogo TNG, NetLogo, VensimPLE, Simile, and AnyLogic) against these properties. The results show that none of these provide significant support for CAES simulation. However, Simile and AnyLogic both provide an extensible framework that could support such simulation if additional development is performed.

I then conclude with how such development could be approached and provide an alternative solution if these platforms (Simile, AnyLogic) cannot be extended to support evolvable components. This alternative approach is a new language, built as a hybridization of current System Dynamic and Agent Basted Modeling frameworks, that includes components with

attributes that support evolvability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

SYSTEM: a set of things—people, cells, molecules, or whatever—interconnected in such a way that they produce their own pattern of behavior over time [35].

SYSTEM DYNAMICS: a methodology for studying and managing complex systems that change over time [20].

TIMESTEP: a discrete unit of time used in simulating a model, also sometimes referred to as a 'tick.'

ACKNOWLEDGMENTS

My sincere appreciation for the patience and support of my friends, family and colleagues, especially Kathy—my better half, and Daniel—a close friend and mentor. Special thanks to Dr. George Mobus for his feedback, ideas, and dedication to this work. *Cheers to all.*

DEDICATION

to the sunny Kathryn Rota

Chapter 1
INTRODUCTION

Building mathematical models has been a human activity for centuries. Through the ages, models have been used to understand the world around us. We build mental models [38] to describe how we should act in social situations like at a market, or attending a show. Models were built by Kepler to describe (and predict) how the heavens moved around us [44], or rather, how we moved through the heavens. In recent times, models are used to predict collapses in stock markets [30], or even determine whether or not a customer is pregnant based on their shopping habits [22]! From the laws of planetary motion, to the complexities of social networks and buying habits of consumers, all of these models have the same thing in common. They are are models of systems at varying levels of complexity.

1.1 Background

A significant body of work exists describing methods for designing, simulating and analyzing complex systems. Methods range from Ford's System Dynamics framework [20], to Resnick's Agent Based Modeling [43]—and many hybrids in between [6]. Methods have been introduced for object oriented approaches to help accelerate the construction of population of heterogeneous agents [41]. Complex system design has even found its way into entertainment in the form of electronic games, such as Will Wright's SimCity [43].

1.2 Motivation

The science dedicated to the deep understanding of systems and their behavior, Systems Science, has become increasingly important and has given rise to new disciplines such as Systems Biology, Systems Chemistry, and Earth System Science [38]. The fundamental

theory that nearly everything is a system means that there are aspects of systems in all things. And, as we try to answer more challenging questions—requiring answers and predictions to more complex ideas—we require a robust and rigorous way to solve these problems. Therefore, tools and methods have sprung up over the last 50 years to help us answer these questions. Using tools available today, adaptive systems can be simulated using techniques such as feedback loops [35] for dynamic systems, and reinforcement learning [32] for agent based models. However, there is still work to be done. Current modeling platforms appear to be insufficient for simulating systems with highly complex adaptivity, such as our nation's economy [17].

Less explored in system simulation is the idea of *Evolution as a Universal Principle* [38]. In this thesis I will explore the concept that systems can, and do evolve and there are insufficient tools available for the adequate simulation of that evolution. Though a body of work exists in the domain of genetic and evolutionary programming, some addressing learning and evolution together [33], the current methods do not address the problem at a system level. Much like the fact that the behavior of a system is much different than the individual contributions of the agents therein [29], such is the difference between the evolution of a system and the evolution of the agents. There is need to further explore these differences and assess whether current modeling platforms can support this domain of exploratory modeling.

1.3 Objective

The understanding and predicting the behavior of complex systems is the focus of this work. I seek to evaluate various tools, languages and paradigms that help us understand complex systems today. I will also propose potential gaps in these tools and paradigms that restricts the ability to fully understand specific types of systems—particularly systems that simulate deeply rooted learning, adaptive frameworks for strategic and tactical decision making, and systems that have the ability to evolve. These systems we shall call Complex Adaptive Evolvable Systems (CAES).

1.4 Key Contribution

The key contribution of this thesis is to help clarify key differences between adaptive and evolvable systems with respect to modeling, and evaluate the tools and languages that are used to model and simulate them. I also introduce key attributes for evaluating simulation tools in their support of modeling CAES and demonstrate their use on several popular modeling languages. These attributes can be used as a stepping stone for the evaluation of additional tools and languages or to help a modeler decide what tool to use to simulate a CAES.

Chapter 2

COMPLEX ADAPTIVE EVOLVABLE SYSTEMS (CAES)

In this chapter, I will provide a detailed description of what it means for a system to be Complex, Adaptive and Evolvable. The intention is to describe how a system can contain all of these properties. Furthermore, I will touch on the necessary attributes a simulation tool must contain to support the design and simulation of such a system. It is important to note that there are varying definitions for complexity, adaptation and evolution. I will attempt to use general definitions—as to be as inclusive as possible—while expressing suitable specificity to discuss concrete attributes required for design and simulation of such systems.

2.1 Complex Systems

Many definitions of complex systems exist [37, 38, 27, 12]. Indeed, when some of the most eminent names in the field (Doyne Farmer, Jim Crutchfield, Stephanie Forrest, Eric Smith, John Miller, Alfred Hbler, and Bob Eisenstein) were asked this to define complexity in a panel discussion in 2004 each provided a different response [37]. If a formal definition of complexity cannot be agreed upon by scientists, a formal definition of Complex System would also be illusive. There are, however, attributes common to complex systems that can allow us to identify whether or not a system can be described as complex [37]. The classification and naming of such attributes varies by author, but there are fundamental similarities.

First, is the aspect of non-linear behavior [43, 37]. Rolling a rubber ball across the floor will result in very simple, uninteresting behavior. Such a system would not be considered complex. Placing that same ball in a river, however, would produce less predictable, non-linear patterns, and would thus be considered complex. Another example is population dynamics [35]. A population doesn't continue to grow exponentially. Rather, we discover

that it is influenced greatly by its environment, and even its own growth rate. Often, the higher level, non-linear behavior of a system is called emergence [29, 37].

The second common attribute is information and resource sharing [37]. Information can consist of messages, such as communication between agents, or messages sent across a wire, or the exchange of energy or matter [38]. The exchange of information occurs both within the system, between components, and with the environment in which the system exists.

Third, is the apparent lack of centralized control, or decentralization of the system [43, 37, 29]. Complex systems, such as ant colonies, exist and execute without a commander [29, 43]. Even the human brain operates without a single node dictating how the rest should behave [37], and economies are no different.

Finally, an attribute common to complex systems is that of hierarchical structures [13, 38]. Systems are in many cases made up of components that are themselves a complete system. Consider the human body as a system. The human body consists of the immune system, nervous system, digestive system, etc. Each system is made up of a collection of cells, which are also a complex system. This hierarchical distinction between systems can be difficult to define, however. Meadows describes the idea of systems as a continuum [35], drawing boundaries around systems is for the purpose of discussion and understanding. It is the case, that when we seek to understand how a system works, we must be able to observe these boundaries. Therefore, we must construct our system hierarchies with the intention of deeper understanding and explanation.

There are many tools/software that allow for the design and simulation of complex systems [6]. These tools allow for non-linear behavior to be monitored over time by capturing information at each time step [41, 38]. Information sharing is managed through connecting components with what are often referred to as 'flows.' In agent based models, agents can share information and change the state of their environment or other agents [6]. The thirdly mentioned attribute, decentralization, is at the heart of tools like StarLogo [43] allowing simulation of many components acting on their own, in parallel. Other tools, such as AnyLogic [9] and Simile [41] allow for systems to be constructed and then used as subsystems. The

ability to do so, is important to construct models of complex systems. [38].

It is often the case that complex systems possess a further attribute, *adaptability*. Of the examples used in this section, most would be considered adaptive, such as an ant colony or the human body. This is not the case for *all* complex systems, however, and thus the separation in this chapter. For example of a ball flowing down a river, is not considered adaptive as it does not contain the properties that are explained in the following section.

2.2 Adaptive Systems

Like complex systems, a tight definition is difficult to wrap around adaptivity. Holland [27] describes adaptive systems as those that "change and reorganize their component parts to adapt themselves to problems posed by their surroundings." Indeed, a system that performs an act such as reorganization of its components is adaptable. It may seem to some that this would imply a consciousness to the system, making direct decisions about how to solve a problem. It turns out, sometimes these adaptations are the result of emergence. Consider, for instance, ant colonies—specifically the behavior of worker ants. Research has shown that worker ants adjust their activities based on the relative frequency of pheromones produced by other ants [29]. If an ant is on trash duty and starts to sense, via pheromones, that significantly more ants are collecting trash than are harvesting food, the worker ant will change roles. The primitive brain of the ant couldn't possibly be aware that this is better for the colony as a whole, but its behavior did change on the basis of its environment. Thus, an ant colony is considered an adaptive system. A design and simulation of such a system using an agent based approach is relatively simple, based on a very small set of rules. In fact, researchers have done constructed such models [43]. This, however, is a very basic form of adaptivity, based on very simple rules. A second example of such an adaptive system based on simple rules is our own immune system [27]. As new potential threats are introduced into our bodies, our immune system adapts to protect us from future harm.

I've explained that adaptive systems needn't be based on deep learning and strategic planning, but as a system becomes more sophisticated, or more highly evolved, adaptivity

grows from a much greater level of complexity. Consider the action of a single human being. A human is a complex adaptive system, just like many other organisms (or perhaps better stated, humans are a complex adaptive system made up of many complex adaptive systems). Exhibiting even greater complexity is the simulation of many humans, working collaboratively or competitively in social circles, economies, governing bodies, and more. Modeling such complex systems requires a much deeper understanding of the system, and how adaptations are accomplished at operational, tactical and strategic levels.

A tool that enables the design and simulation of the entire spectrum of adaptive systems would undoubtedly support basic feedback loops [20] and management of the state of simple agents [43]. In addition, a hierarchical approach, enabled for both bottom up (construction [43]) and top down (decomposition [38]). To support adaptation at varying levels of planning, learning at varying timescales would also be required. A component with similar properties to an *Adaptrode* [40] would prove useful, though Volterra functions could also prove useful for purely mathematical models [24]. However, as I will explore later in this thesis, a primary feature of a system design and simulation tool is for it to allow for rapid creation of models and to be accessible to those without a system science background.

2.3 Evolvable Systems

Holland describes complex adaptive systems as having an "evolving structure" [27]. It is often the case that Holland used the term *adaptive* where *evolutionary* or *genetic* might have taken its place. Consider [26, pg. 121] where an "adaptive plan" is described as employing crossover, inversion and mutation. The then follows in the same explanation a reference to the plan as a "genetic plan." In [38, pg. 527], Mobus summarizes different types of change. One of which, adaptability, is described as the change "wherein a bounded system adjusts its internal workings in response to environmental changes." In the sense of simulation, a component can continue to operate within a dynamic range by adjusting its internal processes or behavior when changes to its environment occur, e.g., temperature variations can be tolerated by within a specific range in the short term. Humans for example

tolerate temperatures between 50 and 100 degrees, with an ideal of 70. Their system adapts within the degree of tolerance. Over time, this dynamic range and ideal condition may change through evolutionary processes through slight genetic variation, known as microevolution.

The last type of change described is that of evolvability which "involves the construction of new structures and functions." [38, pg. 527]. Dramatic changes in a system's structure, such as the replication or the complete removal of a component is known as macroevolution. Placed into the context of simulation, this requires the generation or destruction of fundamental building blocks of the system, in addition to the modification of existing components.

Genetic algorithms or evolutionary methods for solving problems are not new to system simulation, and have been used specifically in agent based models [33, 46]. What has not been thoroughly explored in the literature is how complex adaptive systems evolve, in this specific way—through accidental, stochastic, or entropic forces. It is within the scope of this thesis to discover if exploration of evolvable systems can be conducted using existing simulation tools, and if not, provide a basis for how we might construct such capabilities.

To fully realize how a complex adaptive system would behave in at large timescales, an introduction of such a property in the model is required. At such large time scales, changes to the environmental conditions can lead to a new selective forces. In terms of a genetic algorithm, this would be equivalent to modifying the fitness function over time, changing what it means to be fit. For example, to model the evolution of temperature tolerance in a human population, each agent would be given an evolvable attribute for temperature tolerance. This attribute would be given a function to change the ideal and range states, and a probability of undergoing mutation. An upward trend to the environmental temperature would then be given an evolvable attribute designed to trend upward overtime. This would allow us to see how the agent population might respond to global warming.

Because a complex system is inherently hierarchical any subsystem within the model may be tagged with such a property. That property could result in changes within any component to model components, their behavior, relationships with other components (flows), the limits and ideal weights of stocks—or result in entirely new components, flows, stocks and more.

Thus, a property may need only be set and configured on a system component, and be inherited by its subsystem.

When dealing with evolutionary simulation, there is a need to evaluate survival based on a fitness test or function [36]. The fitness test for system survival is slightly more complex based on one simple observation... "the environment always changes [38]." Many changes could have evolved within a system without an increase (and perhaps even a decrease) in fitness for a period of time, only to become advantageous in a particular environmental shift. What makes a system more or less fit in what kinds of environments? At what level of a system are evolvable components more or less effective? Far more work in evolutionary systems will be required to be able to answer these questions.

2.4 CAES

Complex Adaptive Evolvable Systems (CAES) are systems that contain properties of the systems I previously described. Simulating such systems requires a hybrid of evolutionary programming, adaptive agency and complex system dynamics. Examples of these systems are (in order of size and complexity): Planet Earth, the biosphere, human civilization, the world economy, corporations and governing bodies. Constructing models of these extraordinarily complex systems that reasonably reflect their natural counterparts requires consideration of all of the attributes and structures I have described in this thesis. Tools used to construct these systems must support key methodologies for building complex systems, such as "Internal Models" and reusable "Building Blocks [12]." The extraordinary size of these systems requires a method of decomposition, so that understanding can be gained at varying levels of abstraction. This task is no small order, and the computational power required could be enormous, so parallel processing could prove paramount [46]. However, understanding these complex systems can help us understand how our behavior is shaping the world around us, how policy changes impact our economies and our environment [20].

Chapter 3

EVALUATED ATTRIBUTES

Building a model of a Complex Adaptive Evolvable System (CAES) is not an easy task. It requires that the model creator has sufficient knowledge, not only of each individual component, but how those components interact, behave and change at different timescales. Even if the design of such a system could be performed without the aid of computer software, the simulation of that system most assuredly requires digital computation. Even a small system consisting of 10 components, simulated with a timestep of a single day, for single year period would require thousands of calculations. If software is required to simulate the model, it is only natural for software to support building the model as well. It would be ideal for the design and simulation of the model be included in the same software package, and indeed that is quite often the case.

In the following sections, I will describe the aspects and features that would assist in the design, construction, simulation and analysis of a CAES. Existing literature or standards for such an evaluation do not yet exist. Allan [6] completed an evaluation of many Agent Based Modeling tools in which he described what tools are currently being used in which domain. However, a systematic review of these tools and their features was not included in the survey. Based on the characteristics of the systems described in the previous chapter, I have identified six aspects of a system simulation tool needed to effectively simulate a CAES. These are not intended to be *de facto* properties, but could indeed lay some foundation for future work. The properties as as follows: Adaptability, Fertility, Ease of Use, Support for Hierarchies Inheritance and Modularity, Ease of Output Consumption, and Evolvability.

3.1 Adaptability

To design and simulate an adaptive system, a software must contain a series of features. Most trivially, the modeler must be able to create components that have the ability to change their behavior based on a current state. This could be as simple as basic branching in a behavior function.

To enable a more sophisticated approach to adaptation, some components should be able to retain memory, and modify their behavior differently based on events that have happened in the past. Furthermore, the way that a component is changed may not be known a priori. A component should be able to learn ways to respond based on reinforcing or correcting feedback. Moreover, a component should be able to be programmed in such a way that forecasting by the component is possible based on learned behavior.

If a simulation software fully supports responsive behavior based on a component's current state, the ability for a component to behave differently based on previous actions, and enable predictive behavior of its components, without requiring extensive coding, the software would be considered in this evaluation to be extremely supportive of Adaptivity.

3.2 Fertility

Fertility is sometimes used to describe how useful a model is in several contexts. A basic predator-prey model, for example, would be considered highly fertile as it can be applied as an answer to many different questions, not just simulations of one species of animal preying on another, such as a competing business in a marketplace. In this context, a simulation tool's fertility is that of how usable are its outputs external to the software itself. To support reuse and collaboration, systems (or any subsystem therein) should be able to easily serve as components or subsystems in other projects, or elsewhere in the same system.

Along with the ability to reuse components, a modeling tool's fertility is measured by its its portability. The software should be able to be ran on a wide range of computing systems, with different operating systems and architectures. It is sometimes the case that

the simulation of the model (e.g. via web-browser or Java applet) can be more fertile than the design of the model. The simulation portability can help the fertility of a tool in this evaluation, but the overall evaluation is a combination of design and simulation.

Additionally, a tool's extensibility contributes to its fertility rating. Allowing a population of programmers the ability to extend the feature set of a simulation software can prove extremely useful in making the system more robust, and perhaps enable even more complicated structures to be designed with greater ease.

Lastly, it is extremely valuable to fertility to allow for executable or compilable code to be generated by the simulation software. This enables models to executed on many platforms, or even optimized by software developers. It also enables portability by allowing a user that does not possess the design/simulation software to still execute the model and see the results.

3.3 Ease of Use

Software, regardless of its purpose, is only as effective as it is usable. A tool that allows for simulating complex systems such as the economy may not be simplistic, and would take time to learn. It is however important that the interface be accessible for a wide range of users. To begin, the tool should not require knowledge of computer programming principles such as data structures and algorithms. Even the need for coding would ideally be reduced. Furthermore, addressing attributes of a system such as adaptability shouldn't require extensive knowledge of machine learning or artificial intelligence.

Any well designed software should be responsive, stable and relatively intuitive. Software that can be used to simulate large models is no exception, and responsiveness is even more important. Specifically, if a system is in the middle of simulation, interactivity with the software would be quite useful. This can be done with proper use of threading.

Finally, as with any software, the tool should be well documented. If detailed documentation and instructions are not available from the creator of the software, a public forum dedicated to collaboration would be a strong replacement. If both exist that is even more ideal.

3.4 Hierarchies, Inheritance and Modularity

As I described in chapter 2, many systems are recursive in nature, in that systems contain components that can be and often are systems themselves. The ability to focus on an individual component in a system and construct within it a subsystem would greatly aide the modeler in designing and understanding systems with a high number of components.

Systems also often exude a property of inheritance, where components of a system are very similar in behaviors, but have describable differences. To aide the modeler in design of these systems an object oriented technique of inheritance should be supported. This can provide more rapid creation of new similar components. Moreover, allowing the overriding of inherited features could prove beneficial as well. This could assist in simulating mutation or allow for many inheriting classes to exhibit common behavior with a few single outliers.

Lastly, a hierarchy could exist to manage varying timescales of the model. Lower level components may need to be simulated at a minute scale, while higher level components could updated at an annual scale. This is simply a performance enhancing feature, in that cycles are not spent calculating values of components when they are not needed. The option of varying timescales also offers the modeler to think differently about each component at the level at which that component part of the system is existing.

3.5 Ease of Output Consumption

The simulation is of little use if the output or results of that simulation are not consumable. Therefore, the output of a simulation is as important as its performance. The output of a system can be split into two categories: real-time—information displayed to the human or system performing the simulation, or persistent—information that can be accessed either after the simulation is completed, or for past timesteps of a simulation while it is still running.

For real-time information, the display should ideally be human readable, or alternatively machine readable to be streamed into another system for visualization if the simulation software doesn't support real-time display of the information it calculates. If human read-

able, the ability to focus on specific components of the model will be critical for human understanding.

Persistent information needn't be directly written to a persistent media, but should at least be stored in memory or a buffer to allow it to be consumed by a human or other system. The output of such a system should be exportable, preferably in a non-proprietary format to allow other tools to read, visualize and interpret the results. In addition, the ability to export partial or aggregated results would prove beneficial as data collected from simulations can be quite large.

Lastly, some of the data exported from a simulation should be able to be optionally modified and used as an input to the same model for subsequent runs of the simulation. This could allow the system to be continued with a slightly perturbed state, or started from a specific timestep in the middle of the previous simulation.

3.6 Evolvability

To support evolvability a simulation software would need to enable a few key behaviors. First, the system should be able to modify itself by constructing brand new components, flows, behaviors and functions or by removing existing components. In order to design such behavior, evolutionary functions would need to be supported, and applied to the components to which the functions are applied.

System evolution can depend on the evolution of several components at varying time scales. The ability for software to simulate this relies on the software's ability to execute the evolutionary functions designed for each evolvable component at a different time scale.

It is important to note that if evolution can be programmed to take place within a simulation, does not necessarily mean that the tool used in that simulation supports evolvable systems—at least not in this context. This evaluation of evolvability support focuses on the ability for the building blocks to replicate and modify their structure without the need for explicit programming.

Evolutionary systems have not been deeply studied, and it is difficult to predict exactly

what features could exist to aid in the design and simulation of these systems. More work is required in this area to understand the fundamental system design required to support the domain. It is my intention to explore possible ways current tools may be used to enable this behavior and determine in what ways modifications can be made to advance our understand of evolvable systems.

Chapter 4

EVALUATION

In the following sections, I will describe my experience with five system simulation tools, StarLogo TNG, NetLogo, VensimPLE, Simile, and AnyLogic. These tools were selected based on their popularity and frequent reference in the literature [43, 20, 36, 41, 11]. Other software was considered, such as STELLA [4], but not evaluated due to financial and time constraints. For an extensive list of Agent Based Modeling tools, see [6]. A similar extensive list of System Dynamic software was unable to be found in the literature—there is, however, a list is available on Wikipedia [2].

Comparing the tools that were chosen, StarLogo TNG and NetLogo specialize in Agent Based Modeling, while VensimPLE supports primarily System Dynamics. Lastly, Simile and AnyLogic provide a user with a hybrid approach to system modeling. The purpose of this evaluation is to measure specifically how each of these tools may be used to easily design and simulate a Complex Adaptive Evolvable System (CAES) and is not meant to be an exhaustive evaluation of the software.

4.1 StarLogo TNG

StarLogo is a agent based modeling tool created by Mitchel Resnick and Eric Klopfer at MIT Media Lab. StarLogo TNG (The Next Generation) is one of the latest iterations in the "Logo" family, grandfathered by Resnick [43]. This series of tools was developed to help make modeling more accessible to students. StarLogo TNG adds to its predecessors by including a 3-D World and block programming structure [8] that allows further enrichment to the simulation. The interface is colorful and inviting, and interesting models can be produced in a matter of minutes. The block programming interface opens the programming of agents to

a population that does not possess knowledge of typical programming syntax, though such knowledge does help. Figure 4.1 shows an example of the block programming interface. You can see the common if/then/else construct found in other programming languages, but it is implemented using a series of colorful blocks that snap together only if their shapes fit, much like a jigsaw puzzle. I have found, however in some circumstances, such as designing a graph to track output, blocks with matching shapes may not fit in all slots of matching dimensions.

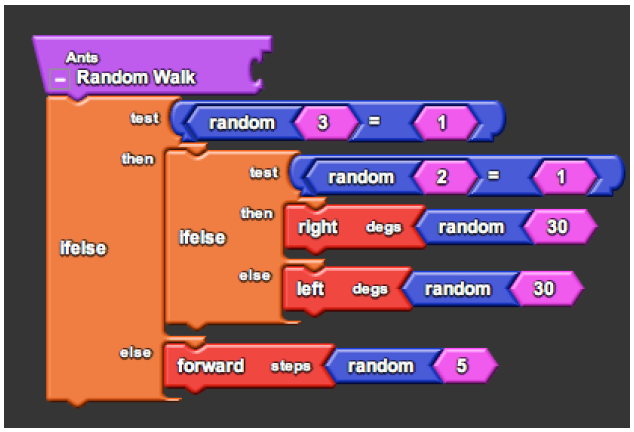

Figure 4.1: An example of a random walk created using the StarLogo TNG Block Programming interface.

4.1.1 Adaptability

StarLogo TNG supports the creation of agents that are segregated into different "breeds." Each breed behaves as all other agents in the same breed. This means they will have the same procedures and decision making patterns as others in their cohort. Each agent maintains its own state and behaves according to instructions provided before the model is designed. Clever use of the block programming interface could produce learning behavior, though it could prove difficult. StarLogo TNG provides some built in functions to occur when collisions occur with other agents, or if they 'smell' something near by. Very little exists within the

program itself to promote deep learning within the agents. I have concluded that producing memory at varying timescales for agents in StarLogo would be a difficult task left to the modeler, therefore the support for adaptability for StarLogo TNG should be considered *low*.

4.1.2 Fertility

StarLogo TNG provides the modeler the ability to create *public* procedures that are used by agents of different breeds. In this way, procedures can be accessed by any breed in the system. Procedures, however, cannot be exported for use in OTHER models. Breeds with predefined behavior also cannot be exported. As such, a StarLogo TNG model must be constructed from scratch. It could be the case that a base model is saved and used as a template, however the heterogeneity of breeds and procedures in modeling is vast, constructing all inclusive templates would be more burdensome than effective. StarLogo TNG does allow for collaboration and publishing of models to a central repository. An expansion of this system may help increase the fertility of StarLogo TNG models. However, until objects can be used between simulations, the fertility of StarLogo TNG should be considered *low*.

4.1.3 Ease of Use

Because StarLogo TNG was created as a teaching tool, it is not surprising that it is incredibly easy to use. The interface is intuitive and the models are responsive as long as they remain sizable enough to fit into available memory. The system is also highly stable. I experienced no crashes or unexpected error messages in my experience evaluating this tool.

A modeler can learn the fundamentals of this tool within a few hours of experimentation and reading. The block programming interface allows for a user to explore what functions are available and how they interact with each other. Additionally, many sample projects appear online and MIT provides extensive tutorials for building many different types of simulations. The amount of online documentation is satisfactory, though the in-tool help provided little value.

From the perspective of a modeler attempting to begin constructing very simple agent based models, StarLogo TNG is extremely simple to learn and use. Once a model becomes more complex, the simplicity of the tool could become a hindrance. Because of this limitation, I have rated the Ease of Use for StarLogo TNG as *medium*.

4.1.4 Hierarchies, Inheritance and Modularity

StarLogo TNG possesses only limited support for object oriented approaches to modeling. Breeds are able to behave uniquely to themselves, but they inherit no behavior from a parent—nor can they share processes with other breeds, with the only exception public procedures that are available to all breeds.

There is also no concept of a subsystem. All agents exist as the same level with everything else in the system. Furthermore, the timescale for simulation is static and cannot vary between objects. The modularity of this tool is visible through the block programming structure, and is extremely limited. Separation exists only between different breeds and public procedures.

Because of this tool's complete lack of support, I consider the rating for Hierarchies, Inheritance and Modularity to be *none* or *no support*.

4.1.5 Output Consumption

During a simulation, a modeler can watch the simulation unfold on a 3D rendered world known as *SpaceLand*. The camera can dynamically follow an individual agent, or can remain static. The angle and zoom of the camera is also customizable. Additionally, traces can be made on specific variables as the model is simulated. From these monitors, graphs can be constructed and exported as an image, or their underlying data saved to a text file in comma separated format. These graphs must be constructed prior to simulation in order to track the desired variables. Each graph must be exported using the graphical interface. If a complex simulation was to be ran, and its output analyzed by an external tool, a user could possibly have to manually export dozens of charts, which could be quite cumbersome.

Because images and raw data can be exported by this tool the output can be consumed with external tools. The exportation is cumbersome but possible, thus the rating for Output Consumption for StarLogo TNG is *medium*.

4.1.6 Evolvability

These is no support for constructing new breeds, procedures or components within this tool. New instances of agents can be created during run time, and procedures can be created in such a way that random behavior can be simulated. Through sophisticated use of procedures and state, significant changes to the way a procedure executes is possible, but the absolute manipulation of the model's components is not. It is quite clear that StarLogo TNG does not support the design or simulation of an evolvable system and thus receives a rating of *none*.

4.2 NetLogo

NetLogo is an agent based modeling tool created by Uri Wilensky and is described in detail in [43]. As a successor to StarLogo (predecessor to StarLogo TNG), NetLogo's design is extremely similar. Agents exist in an environment consisting of patches. The simulation is controlled by a series of buttons and variable manipulation tools such as sliders and chooser. A key difference between StarLogo and NetLogo is the method in which agents receive instructions. Instead of the jigsaw-like block programming interface, NetLogo requires the user to program in a simple code window. An example of the syntax is displayed in Figure 4.2. NetLogo versions do exist for 3D environments. However the graphic libraries were not supported by my computer and are therefore not part of this evaluation.

4.2.1 Adaptability

The programming interface of NetLogo supports differential behavior based on the current state of an agent. Each agent contains their own state and behavior set, inherited by their

```
to walk
  ask turtles [
    ifelse random 3 = 1  ;; 1/3 OF THE TIME CHANGE DIRECTION
    [      ;; true
      ifelse random 2 = 1    ;; 1/2 OF THE TIME TURN LEFT, OTHERWISE, RIGHT
      [
        lt random angle ;; TURN RIGHT RANDOMLY (MAX = ANGLE)
      ]
      [
         rt random angle ;; TURN LEFT RANDOMLY (MAX = ANGLE
      ]
    ]
    [      ;;
      fd random dist   ;; WALK FORWARD RANDOM DISTANCE
    ]
  ]
end
```

Figure 4.2: A sample of a random walk routine created in the NetLogo programming language

breed. It would be possible to design procedures for varying behaviors based on a particular state of an agent, and the environment surrounding them. This includes nearby agents and patches. Further more, a process can easily be invoked at predefined intervals. This would allow for memory procedures to be ran at increasingly distant intervals, copying more recent memories to more distant registers. Doing so, could produce learned behavior over different timescales producing similar behavior to an Adaptrode [40].

Though possible, constructing such a learning algorithm could prove difficult. NetLogo includes only a few native data structures, though extensions for more advanced structures do exist. Additionally, the coding interface is quite primitive, lacking features of a modern IDE such as auto-complete, project explorer, or script outlines. This could make managing a system with a large code base and complicated algorithms quite cumbersome. Even so, I would consider the level of support for adaptability to be *medium*.

4.2.2 Fertility

NetLogo models can be uploaded to a shared community site known as *Modeling Commons* and can be exported in whole or in part to a binary file. Likewise, models can be imported in whole or in parts. These parts are not by component, or breed, unfortunately. One can export their interface or patch color, the initial landscape of the model.

In addition to exporting a model, NetLogo supports a framework of *extensions* that allow programmers to extend the language to include additional features such as hash tables. This community appears to be very active and has provided many extensions [3] that were not included in this evaluation.

Lastly, NetLogo allows a user to export a model to HTML which generates enough HTML markup and JavaScript to execute the model in a web browser. This includes the ability to simulate the predefined model, but to also manipulate the source code, all from the browser window. With so many ways to extend, import and export models using NetLogo, I consider the fertility rating to be *high*.

4.2.3 Ease of Use

NetLogo is quite easy to use, and the system is responsive. A user can begin to create basic models in a matter of minutes, after learning a few basic commands. Easy access to sample models can also help a user quickly learn how to build various types of models.

However, though the syntax of NetLogo's programming language uses simple nouns and verbs for referencing agents, patches, and procedures, a strong knowledge of procedural programming is required in order to construct a model with any level of complexity. There is no graphical interface for designing how the model behaves, only interfaces for building buttons, plots, and global variable manipulation objects. The goal of such a system is to provide non-programmers the ability to construct complex systems without the need to understand programming. I consider the ease of use to be *low* in this regard.

4.2.4 Hierarchies, Inheritance and Modularity

NetLogo supports agents of varying breeds, but contains no support for object oriented design or inheritance. Code for all procedures for all breeds exists within a single file which even for smaller models can grow quite lengthy. Modularity is not well supported within the tool. Additionally, breeds cannot inherit procedures from their parents, though they can be grouped into an *AgentSet* so procedures can be called on multiple breeds simultaneously. Even so, the support for hierarchies, inheritance and modularity within NetLogo is substantially absent enough to be rated as *none* in this thesis.

4.2.5 Output Consumption

NetLogo allows for traced values to be plotted and exported, even during simulation. The data can be exported as an image, or to a text file in comma separated format. The exporting of data can be handled via the user interface, or throughout simulation by including output commands in the source code. This programmatic method of data export could potentially allow a user to stream output of an NetLogo model to another system for consumption. Additionally, the plotting interface accepts programmatic expressions to be used for data capture. This permits significant flexibility to the modeler. Lastly, an entire simulation can be exported in the form of a *world* which is a consumable comma separated value file. The export includes the state of all agents, patches, and tracked plots. Because of the flexibility of the data extracts, I consider the rating for output consumption for NetLogo to be *high*.

4.2.6 Evolvability

NetLogo provides several features that may support Evolvability. First, the ability to implement procedures in other programming languages and then implement them as extensions could allow a developer to construct an agent that contains Evolvable properties. However, further study of the limitations of NetLogo's extensibility would be required to prove it possible. Also, the exportation of a simulation's source code into an interpretive language such as

JavaScript, along with the ability to export and import an entire world into the model could allow creative methods for implementing an evolvable system. A system could be simulated, exported, the source code manipulated programmatically, and the previously exported state imported, and the simulation continued. If this method could be implemented, however, it would require code to be executed outside of NetLogo, thus NetLogo itself wouldn't be said to support evolvability on its own, and that is what is being evaluated. My conclusion is that though potential exists for supporting evolvable systems, the resulting rating from this evaluation is *low*.

4.3 VensimPLE

VensimPLE, a product developed by Ventana Systems, is a model simulation and design tool with extensive capability to support dynamic system modeling and is used quite heavily in [48], a popular system dynamics text. VensimPLE supports a number of analytical tools to help the modeler understand more deeply the behavior of the dynamic system being modeled. These tools include loop detection, causation trees and dependency trees. Tables, graphs, comparisons and reality checks are other tools made available to the user to analyze their models. It is quite clear that VensimPLE is a powerful tool for discovering how a dynamic system behaves.

4.3.1 Adaptability

VensimPLE offers typical components for constructing a dynamic system, including stocks, flows, valves, feedback loops, sources and sinks. Valves that are applied to a flow can be customized using a formula editor and can be influenced by any external variable. Those variables, in turn can be influenced by any number of variables. Using built in functions such as delays, a structure could be produced that mimics learning at multiple time scales, even though the timestep is a constant and applies equality to all components.

The variable editor, as seen in Figure 4.3.1, allows the modeler to provide a significant level of detail to the model, beyond a simple equation. Because of the flexibility of variable

behavior, variables could be designed in such a way to create adaptive behavior, including forecasting.

Programming how all of the variables and objects requires only writing the equation for how each variable behaves, which is an advantage for non-computer programmers. It is the flexibility of the tool and apparent power of the interface that brings me to consider the support for adaptivity for VensimPLE as *medium*.

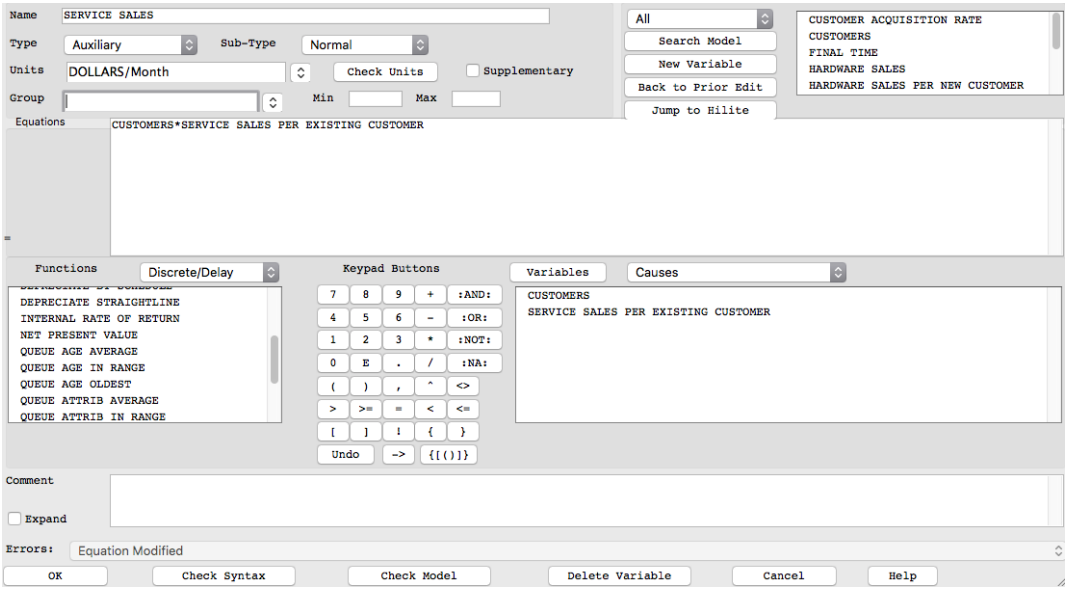

Figure 4.3: A screenshot of the variable editor in the VensimPLE application

4.3.2 Fertility

VensimPLE models are designed and simulated in the same tool, but can be published for use in other Vensim products, though not for standalone simulation. This is unlike other simulation platforms, such as NetLogo or Simile. Likewise constructed components from other VensimPLE models cannot be imported for use into another model. As such, the usefulness of a model constructed in VensimPLE extends only to the model for which it was designed.

VensimPLE also offers a plug-in environment for using VensimPLE capabilities in other tools. It does not appear VensimPLE offers other programs to enhance the capabilities of VensimPLE. Moreover, according to the Workbench website [5] there are no user contributions to this effort. The inability to extend the VensimPLE environment, the lack of activity from the user community, and no option to export the source code for a VensimPLE model, cause the fertility of VensimPLE receive the lowest possible rating, *none*.

4.3.3 Ease of Use

VensimPLE is a robust and powerful tool. Because of the extensive customization available to each component, the tool is not as intuitive as others I have evaluated. The number of various pop-up windows and nondescript drop-downs and check boxes requires thorough reading of the included tutorials, which are just as nondescript. There are however, online references and sample models that can help a new user become acquainted with the tool. The system was responsive and only crashed one time while building a sample model. The built in analysis tools evaluate the model quickly and provide read-outs that are easily interpreted.

Furthermore, the VensimPLE software is exclusively designed to simulate Dynamic Systems, and would require a significant amount of planning to execute a spacial agent based model in the terms of System Dynamics.

Despite the powerful and responsive analysis tools, VensimPLE is a challenging tool to learn and use. Its features support the creation of complex systems, without having to explicitly program how all variables interact. I still consider this system to have a *low* rating for ease of use.

4.3.4 Hierarchies, Inheritance and Modularity

As a System Dynamics language, hierarchies of variables and variables can be constructed as described in [38, pg. 435]. However, the reuse of such a design may not be replicated in a way other than a copy/paste. The VensimPLE system itself does not support the control over hierarchical structures, and contains no references for inheritance or heredity of components.

A complex model can be compartmentalized into different screens known as *views*. Variables from one view can be accessed from another view by use of the *Shadow Variable* feature. The combination of these features can help a modeler focus what is effectively a subsystem of the overall system being designed. However, there is no forced relationship between the views, and careful attention must be made if a modeler is using these features to perform a system decomposition analysis [38].

From a rating perspective, having the ability to construct a hierarchy manually, and the ability to separate parts of the model into various views, balance the complete absence of inheritance or object oriented design. Therefore, the rating of VensimPLE for this attribute considered for this section is *low*.

4.3.5 Output Consumption

According to VensimPLE documentation, the tool supports data exports into spreadsheet (xls) files. However, in the educational version used in my evaluation, no such feature existed. In addition to exportation, VensimPLE also allows a user to use a properly formatted spreadsheet as input to the model. This feature can be very useful when continuing a model from a previous simulation having made a manual perturbation of the model's state. The data created from each simulation is saved to disk in a proprietary (.vdf) format for use in Vensim tools only.

As mentioned in previous sections, VensimPLE offers extensive analysis tools. The results of these tools are exportable as images for use in publications, however they are of little use for analysis in other tools. The output consumption for VensimPLE is thus rated as *low*.

4.3.6 Evolvability

VensimPLE provides no support for programmatically manipulating the defined structure of the model being simulated. New variables are unable to be added, nor can new connections be made during simulation. Additionally, the model itself cannot be exported in a format to be consumed by any other tool outside of the Vensim family. It is the lack of declarative

language and nearly nonexistent fertility that results in a rating of *none* for the evolvability attribute.

4.4 Simile

Simile is a visual modeling language first created by Muetzelfeldt and Massheder [41] and currently maintained by the company Simulistics. Simile adopts the familiar language of system dynamics, much like Stella, Vensim and Powersim. What separates Simile from other languages is its object oriented approach to modeling. Furthermore, Simile allows disaggregation of components that is akin to agent based modeling tools such as NetLogo. Simile provides a modeler with the ability to create classes of objects and blend system dynamics and agent based modeling in a single declarative language.

4.4.1 Adaptability

Simile contains an equation editor similar to VensimPLE, and thus can modify its behavior based upon current state. Variables can influence other components based in custom formulas much like any other system dynamics simulation tool. However, because of its disaggregation feature, each instance of a specified class has its own unique state. This allows it to behave and adapt differently from others in its cohort, increasing the adaptive nature of the model. Furthermore, this behavior can be constructed using the graphical interface, with minimal coding.

Simulations in Simile function on a single timescale. However, Simile has built in delay functions that could be used to simulate values on varying timescales. Using a series of sub-models each with a varying setting for delay, one could simulate memory in which more recent events are weighted more heavily, similar to the Adaptrode [40] mentioned in Section 4.2.1.

Though lacking packaged components for learning and adaptivity, simile has the building blocks needed for creating complex learning inside of the system, in both the aggregated and disaggregated systems. Simile thus receives a rating of *medium* for adaptivity.

4.4.2 Fertility

Simile has a robust system for exporting and importing Simile models. First, the Simile language is declarative and the model's design can be exported to a text file in either Simile's declarative syntax, or a well-formed XML document. An example of the declarative syntax can be found in Figure 4.1. These files can then be used to as a source for importing into another model to be simulated using the Simile software, or for other developers to write software that can simulate Simile models.

Alternatively, C++ source code can be generated for execution on any platform for which a C++ compiler can be written. This allows Simile models to execute on any machine, even if the Simile simulation software does not exist for the machine's architecture. Furthermore, a C++ model can run much more quickly than a model running inside of the Simile application [41].

Furthermore, Simile offers a open framework to allow developers to create new ways to input and output data from Simile, offering extensibility to the product line. In one example, Mazzoleni et al [34] created an integration of Simile with a Geographic Information System.

With the ability to import and export models into the simulation system using an open structured text file, a method for exporting compilable C++ code, and an extensible framework for input and output, Simile receives a rating of *high* in fertility.

4.4.3 Ease of Use

Simile's interface is not as intuitive as NetLogo, but more so than VensimPLE. After reading several tutorials online I found myself being able to create models, both aggregated and dissagregated comfortably. It also came packaged with several samples for new users to review to become comfortable with the tool. The plotting interface is unlike any I have used, and took a few tutorials to understand how it functioned. Overall, learning the system was not difficult.

My installation of the Simile software was quite unstable. Resizing windows and ma-

```
node(node00002,compartment,[],[complete=true,name='Temperature'],[centre=[45,-14]]).
node(node00003,function,[],[complete=true,name=fn1,units=1,value=25],[]).
node(node00004,cloud,[],[complete=true,name=cd1],[centre=[211,-14]]).
node(node00010,cloud,[],[complete=true,name=cd3],[centre=[-122,-12]]).
node(node00012,variable,[],[complete=true,name='Heat loss'],[centre=[-83,-62]]).
node(node00013,function,[],[complete=true,name=fn6,units=int,value=10],[]).
node(node00014,variable,[],[complete=true,max_val=100,min_val=0,name='Set point',units=1,value=65],[centre=[-88,43]
    ]).
node(node00016,variable,[],[complete=true,name='Heat content'],[centre=[281,79]]).
node(node00017,function,[],[complete=true,name=fn8,spec=' 13.2',units=1,value= 13.2],[]).
node(node00018,variable,[],[complete=true,name='Gas flow'],[centre=[175,119]]).
node(node00019,function,[],[complete=true,name=fn9,units=1,value='Valve_position'*'Valve_linearity'],[]).
node(node00020,variable,[],[complete=true,name='Heat input'],[centre=[157,55]]).
node(node00021,function,[],[complete=true,name=fn10,units=1,value='Gas_flow'*'Heat_content'],[]).
node(node00022,variable,[],[complete=true,name='Heat capacity'],[centre=[42,-82]]).
node(node00023,function,[],[complete=true,name=fn11,units=1,value= 4.184],[]).
node(node00024,function,[],[complete=true,max_val=1,min_val=0,name=fn2_0,units=1,value=0],[]).
node(node00027,variable,[],[complete=true,name='Valve linearity'],[centre=[263,154]]).
node(node00028,function,[],[complete=true,name=fn1_1,units=int,value=1],[]).
node(node00031,variable,[],[complete=true,name='Valve position'],[centre=[179,204]]).
node(node00032,function,[],[complete=true,name=fn3_1,units=1,value='Output'],[]).
node(node00048,submodel,[node00008,node00029,node00030,node00033,node00034,node00035,node00037,node00039,node00041,
    node00043,node00044,node00045,node00046,node00049,node00050,node00051],[complete=true,fill_colour='#ffff00',
    multiplication_spec=[count=[]],name='PID controller',separate=0],[bounding_box=[-209,92,93,270],caption_offset=
    [0,0],internal_extent=[0,0,302,178]]).
links(node00048,[arc00024-arc00010,arc00024-arc00026,arc00027-arc00025,arc00028-arc00023]).
node(node00008,border,[],[name=var1],[along=814]).
node(node00029,variable,[],[complete=true,description='Error',name=e],[centre=[155,39]]).
node(node00030,function,[],[complete=true,name=fn2_1,units=1,value='Temperature'-'Set_point'],[]).
node(node00033,compartment,[],[complete=true,description='Integral error',name='Integral Error'],[centre=[52,123]])
    .
node(node00034,function,[],[complete=true,name=fn4_0,units=1,value=0],[]).
```

Figure 4.4: A sample of a model expressed using Simile's model declaration syntax.

nipulating plots would cause the application to crash. Research into the issues revealed no known issues with the program. Fortunately, an installation on another computer was far more stable. Additional research to the system's stability is needed. Because of it's mediocre interface and significant stability issues, I have rated Simile as *low* for ease of use.

4.4.4 Hierarchies, Inheritance and Modularity

Simile was built on object oriented principles, and as such supports modularity, inheritance and hierarchies. Every group of components can be grouped as a subsystem. Components with a subsystem can also be grouped further into subsystems, which inherit properties from their parents. In the example given in [41], a subsystem of a field is created with two subsystems within it of crop or grass, to which each instance of a field is assigned. attributes of fields are assigned, such as owner (farmer) and an (x,y) coordinate. Internal to each subsystem is different system that accepts input from the field system. This example shows an explicit support for the three main attributes of this section. Therefore, the rating for support for hierarchies, inheritance and modularity is *high*.

4.4.5 Output Consumption

I have already discussed that the output framework for Simile is a open framework to allow developers to produce software to consume the output of a model. For a non-programmer however, Simile provides the ability to trace variables throughout a simulation and export the saved table as a file in comma separated format. The variables tracked must be identified prior to simulating the model. For complex models with long running times, this could prove problematic. The options for export customization within the tool are also limited.

Simile supports the export of a dataset to be used as input for a subsequent run of the simulation in the form of saved and loaded configurations.

The open framework offers great flexibility for handling output, but the built in functionality for simulation data export is somewhat limited. The balance between these features results in a *medium* rating for output consumption.

4.4.6 Evolvability

Simile does not support the manipulation, creation or removal of model components during simulation. However, the ability to export an declarative language does offer an opportunity for an extension to be programmed to perform such a manipulation. With a customizable input and output system, a model could in theory be exported to a program that manipulates the model using the open declarative syntax and provides the model back into the system in an altered state. Further more, token attributes of models and submodels may be used as flags to the external program to indicate what models should be considered for manipulation.

While Simile doesn't natively support manipulation of components, it does contain enough fertility and modularity to support an extension that could possibly enable evolvability. I consider this promising set of features strong enough to rate Simile with a *medium* rating for evolvability.

4.5 AnyLogic

AnyLogic is a commercial software produced by the eponymous company, *The AnyLogic Company*. AnyLogic is a powerful simulation tool that incorporates System Dynamics, Agent Based Modeling and Discrete Event processing. AnyLogic runs on the IBM Eclipse [16] framework [10]. The environment is thus modular in nature allowing palettes to be moved and resized to the modeler's preferences, and saved as different perspectives for switching between views. The system is quite impressive and robust when compared to other tools in this evaluation.

4.5.1 Adaptability

Though not much has been written on the use of AnyLogic for use in creating highly intelligent agents, it is clear that the modeling framework would support such development. Based in Java, a modeler can implement nearly any learning algorithm in defining the state or decisions made by an agent class. Simulations have been constructed [11] at multiple time scales; such a technique could be used for enhanced learning of an agent. In addition, multi-method approaches can allow learning to occur using standard feedback looks and not require deep understanding programming.

AnyLogic is extensive, powerful and allows modeling at multiple timescales and multiple methods of modeling to be combined into one single model. However, because not much has been written about implementation of adaptivity within the tool, more research is needed to claim with certainty that adaptive agents are possible within AnyLogic. As such, the rating for AnyLogic's support for adaptivity is *medium*.

4.5.2 Fertility

AnyLogic models can be exported in use in a web browser as a JavaApplet, as a standalone Java Application, or published online in the AnyLogic community portal, RunTheModel.com. The source code of these exports are not available for consumers, but the model

can run without the need of AnyLogic software, allowing others to make use of the simulation.

Also built into the tool is support for SVN source control. This allows multiple individuals to work on designing a model simultaneously by checking out and checking in objects on which they are currently working.

Finally, the object oriented design of AnyLogic allows for objects to be reused within a model, or exported for use in other models. I will discuss the ability to reuse objects further in Section 4.5.4. While strong in the sense that objects in the model can be reused, the language of AnyLogic is not descriptive and cannot be exported for consumption in another tool, nor can the source code required to execute a model be exported, as is the case with other languages, such as Simile. Because of this, and the proprietary aspects of the tool's simulation platform, the Fertility rating for AnyLogic is *medium*.

4.5.3 Ease of Use

The numerous options available to a modeler in AnyLogic is difficult for a new user. To guide a user through their first models, The AnyLogic company provides a book [23] to introduce a user to the platform. Once familiar with the interface, models can be constructed without the use of any computer code. A simple to use properties interface is available to a user for most object types. See Figure 4.5.3 for an example.

However, because AnyLogic provides a user with such a rich environment for simulation, an experienced programmer can greatly customize the behavior of objects in the environment using advanced features that rely heavily on Java code.

Because the interface is so robust, the ease of use is initially very low. However, AnyLogic provides for free an introductory book on modeling, publishes many online tutorials, and has published an inexpensive complete text [9]. This indeed helps a user understand the environment in as much or as little detail as they require. The interface is clean, and crashed only a single time during my evaluation. All of these attributes combined result in a rating of *medium* for ease of use.

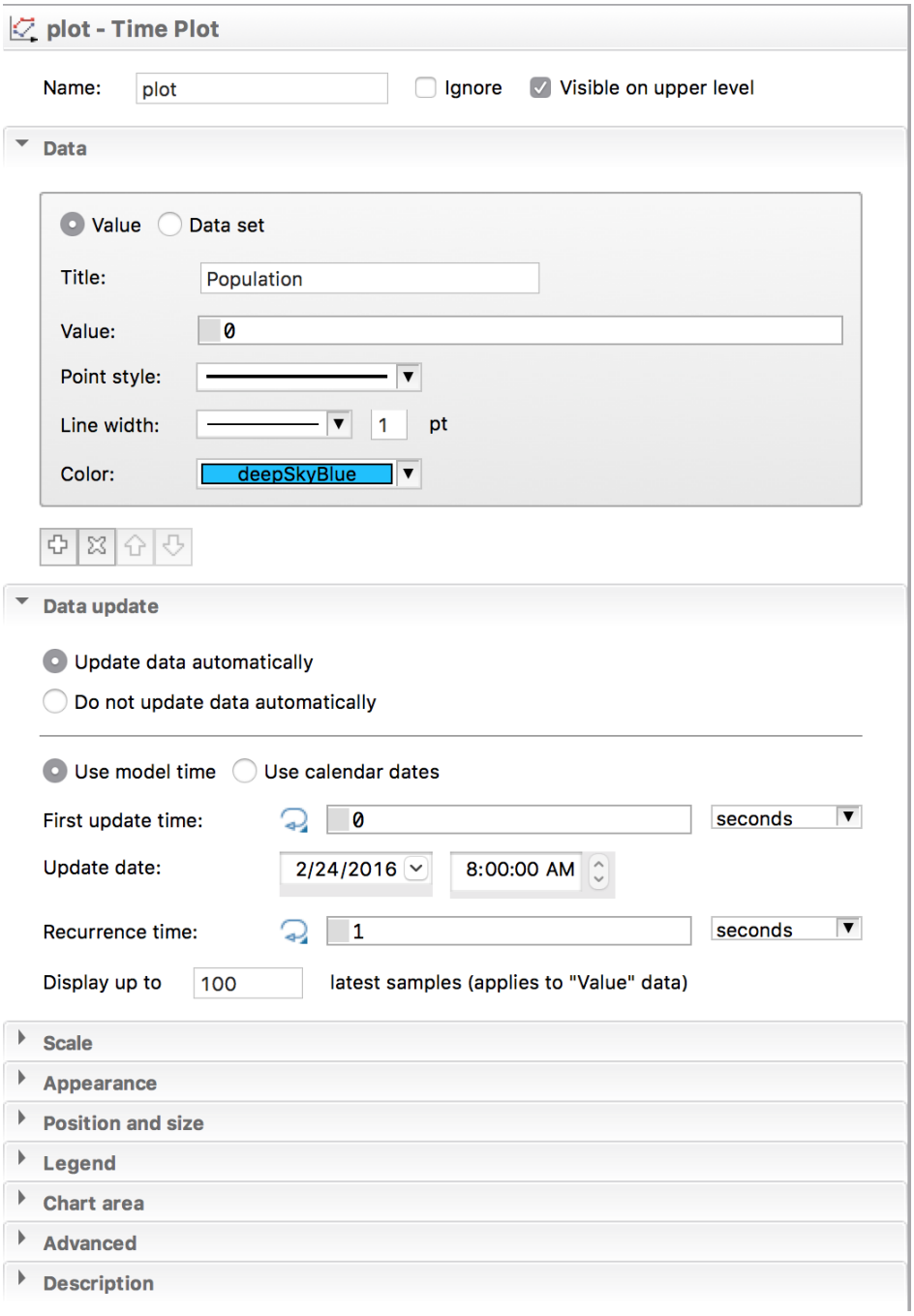

Figure 4.5: The properties window for a Time Plot object in AnyLogic 7.

4.5.4 Hierarchies, Inheritance and Modularity

AnyLogic is built on object oriented principles. Every object in the environment is a member of a class hierarchy. Agents can extend other agents, and objects can implement interfaces, ensuring they can be controlled or interact with their environment using common, though uniquely defined, methods.

In addition to inheritance, and class hierarchies, AnyLogic implements what are known as *Libraries* for certain classes of objects such as Pedestrians, Railways, Connectivity, or Presentation. Libraries can be imported from externally developed JAR files. These libraries contain objects that can be selected and added to your modeling canvas to be customized and connected to proper components. A customized component can be saved or copied and pasted elsewhere in the model.

In my evaluations I have not found a system that implements hierarchies, inheritance and modularity more completely than AnyLogic. It is because of this that I have rated it as *high*.

4.5.5 Output Consumption

As of AnyLogic 7.2 [1] each model contains an integrated database for reading data from and writing data to a model simulation. This allows tremendous flexibility to a modeler in storing any necessary information for later export, though is less intuitive than exporting data from other software I have evaluated.

Part of output consumption is the display of the model while it is simulating. Like StarLogo, a viewer of a model in AnyLogic can watch agents move about their world, or watch charts display levels of stocks and rates of flows. What is unique to AnyLogic however, is the ability to interact with the simulation to update what one is seeing during the simulation.

For example, a map can be displayed, and during simulation, a region of the map can be clicked and the charts updated to be filtered on events occurring within that space. This level of interactivity is not easy to program, it requires knowledge of Java user interfaces to

do so, but the AnyLogic system permits such output.

The flexibility of output consumption for AnyLogic surpasses other systems I have evaluated. I rate the output consumption as *high*.

4.5.6 Evolvability

AnyLogic does not export a declarative language of a system or model, nor is the source code for simulating such a system made available. It does, however, provide an experienced programmer a platform for constructing new instances of objects using native Java code. Classes and interfaces could then be written and exported for others to implement. It could be the case that an *Evolvable* interface be created and implemented by a particular object. The modeler, intending to model an *Evolvable* object would then be *required* to implement specific methods associated with the evolution of the object. For example, fields specifying the expected frequency of mutation can be included in the interface along with a *mutate* function. Interfaces by nature provide only a common structure, however. The modeler would have to understand *how* to properly implement these fields and functions to achieve a truly evolvable component. Furthermore, an *Evolvable Component* or *Evolvable Agent* class might be constructed and exported for use in other models. An extendable class would enable evolvability implemented within an object to be inherited, in whatever way the Component or Agent was designed. Considering several possible types of evolution described in [39] a evolvable component could replicate itself and form new forms, the component may destroy itself, or systematically alter it's functionality. In a compiled language such as Java, making fundamental changes to an object's functionality could prove complicated, however the cloning and destruction of an object is routine functionality of the language.

Possibilities exist and should be explored more deeply in future research. If these types of interfaces or classes existed in AnyLogic today, the rating would be high. However, because the system could possibly support such behavior, but doesn't directly include it, the rating is *medium.*

4.6 Summary

A wide range of languages and tools exist for modeling systems. Each of which have individual purpose and work well for executing within their specialized space. In the set of tools I have evaluated, no language fully supports simulating Complex Adaptable Evolvable Systems (CAES) specifically. To score each language against one another, I have calculated a simple overall score for each language. This score is a simple percentage of total points available. Five of the six properties are worth 3 points. The exception, *Evolvability* is weighted to 6 points, thus 21 points are possible. A rating of High grants 3 points (6 for *Evolvability*), Medium grants 2 (4) points, Low results in 1 (2) point(s), and No Support grants 0 points. Of these, only 2 scored above 60%. To see the summarized results, please reference Table 4.6.

Of the systems evaluated, AnyLogic and Simile possess the highest promise of being extended in such a way that they could support CAES simulation. Of those tools, each have unique advantages. Simile's declarative language is a stepping stone for expansion to include evolvable components, and the exportation of the source code opens doors for performance optimization and reuse. AnyLogic's purely object oriented design and support for external libraries of classes and interfaces offers an nearly endless world of possibilities to enable non-programmers to construct CAES models. Further exploration of these two languages is indeed required to know for certain.

Attribute	StarLogo TNG	NetLogo	VensimPLE	Simile	AnyLogic
Adaptability (3pts)	L	M	M	M	M
Fertility (3pts)	L	H	X	H	M
Ease of Use (3pts)	M	L	L	L	M
Hierarchy Support (3pts)	X	X	L	H	H
Output (3pts)	M	H	L	M	H
Evolvability (6pts)	X	L	X	M	M
Overall Score	29%	52%	24%	71%	76%

Table 4.1: A summary of evaluation results of five tools and six attributes. The letters represent levels of support: High (H), Medium (M), Low (L), No Support (X). The overall score is a percentage of the 21 possible points. High = 3(6), Medium = 2(4), Low = 1(2), No Support = 0.

Chapter 5
A NEW LANGUAGE

I have shown in the previous chapters that there exists unique systems known as Complex Adaptive Evolvable Systems (CAES), and that the current set of languages and tools are not sufficiently equipped to support the effective design and simulation of such systems. In this chapter I will explore how future development may enhance current languages to involve the language of evolvable systems to be included in current simulation tools, or a new tool designed specifically for building a CAES.

5.1 Shoulders of Giants

In Chapter 4, we found that the hybrid, and object oriented approaches to modeling used in Simile and AnyLogic could prove to be effective platforms on which to develop additional tools to support systems. The evaluation performed in this thesis is not exhaustive of all available tools and languages. Additional evaluation and study should be performed prior to significant investment of resources constructing something from scratch. New languages such as Systems Modeling Language (SysML) [21] and Universal System Language (USL) [25] have been created. The former, an extension of UML, is a practical way of describing human created systems, such as computing systems. The focus of SysML is to describe/design a system by showing the relationships between components. It does not provide the level of detail, such as functions and variables, required to fully simulate a system. USL, also created to design human created systems, focuses on three primitive control structures. The use of these primitive structures may prove cumbersome when constructing complex structures such as an adaptive agent.

It is my conclusion that the language of System Dynamics and Agent Based modeling, as

described and implemented in Simile and AnyLogic, be reviewed in depth and expanded upon to implement newly defined components and actions attributed specifically to evolutionary systems. An approach to such a review would be a focused effort to construct a CAES such as a single corporation, and its interaction with the market around it. Inside such a system would be governing committees that define policies, and inside of those committees would exist adaptive agents. The agents within those committees would be added and removed as they are hired into and leave the company. The experience of those agents would vary, based on their tenure with the company. For example, if a committee made a decision with poor results, existing agents may be aware of these results, and not repeat the same decision. However, newer members would behave differently. In addition to manipulating variables within the company, committees within the company may split, cease to exist, or apply influence to alternative components within the company. While designing such a model, an effort should be made to construct generic versions of the agents and components therein. The creation of a second model, using these generic versions would then be created for an entirely different company. Doing this successfully would show the ways for construction of more complicated, more adaptive and more highly evolvable systems to be created using components previously constructed.

If an extension of these languages proves unsuccessful upon deeper review, a new language, building upon the hybridization of System Dynamics and Agent Based methods should be created to enable modelers to design, simulate and understand CAES. This language would certainly incorporate concepts from existing languages, however additional functionality would be included to support systems such as the company model described above.

5.2 New Properties

This section is meant to be hypothetical and further work is needed to specify new attributes and components of a new language, or extensions to an existing language. The details of implementation and integration with systems will also be left for future work. The concepts, however, should be helpful should the envisioned future work be performed.

5.2.1 Component Manipulation

Beyond the manipulation of each component's state, a necessary property of a new or expanded language would be the ability to create, destroy or fundamentally change existing components within a system. The manipulations must be constructed within the constraints of the language, so the simulation engine could interpret the modifications and simulate accordingly. These manipulations cannot be random reorganization of bits, as is a common practice of the mutation and crossover methods in genetic algorithms, but rather randomly generated mutations to systems and components to create entirely new or modified structure. These mutations, however, should be constrained based on additional attributes of the component. As explained in [39] these constraints are based on the importance of a component in a system and would include the replication of highly important components, the removal of lesser important components, or a significant modification of the process inside of component.

5.2.2 System Decomposition

The ability to view and describe a system at varying abstraction levels, and apply Component Manipulation at any of those levels will allow the modeler better understanding of how evolution can impact the system. The necessity to write a manipulation function for every component at every level would prove cumbersome. In an ideal modeling language evolvability should be a configured attribute of a component or subsystem, and then easily applied, through inheritance, to other components or subsystems in the model.

5.2.3 Evolution at Aggregation

Finally, to separate evolvable system simulation from standard genetic programming or evolution simulation the language must support evolution at an aggregated level. Thus, the evolvable framework must also be applicable to the System Dynamic components of a system, not just to individual agent behavior and design. Evolvability could then be applied to

models that are not agent based. Additionally, the modeler can further understand how a complex system, far larger than the system of interconnected agents therein, behaves as it evolves at a very high level of abstraction.

Chapter 6

CONCLUSION

Our world is full of systems. Some systems can be described and simulated rather easily, while others require a deeper experience and knowledge to understand them. These are systems that behave in ways that are difficult to predict, and can produce unplanned emergent behavior at higher levels of abstraction. These are known as Complex Systems. Within the set of Complex Systems is a subset known as Complex Adaptive Evolvable Systems. Attributes of these systems are known to scientists, but are very difficult to define, simulate and understand.

With the tools currently available to us, we can model a wide range of systems, yet these Complex Adaptive Evolvable Systems remain a challenge. Through additional work in defining, designing and creating a language that supports the description and simulation of an evolvable system, we will be able to learn much more about the world around us.

BIBLIOGRAPHY

[1] Anylogic release notes. "http://www.anylogic.com/new-features". [Online; accessed 23-February-2016].

[2] Comparison of System Dymamics Sofwtare. "https://en.wikipedia.org/wiki/Comparison_of_system_dynamics_software".

[3] NetLogo Extensions. "https://github.com/NetLogo/NetLogo/wiki/Extensions". [Online; accessed 22-February-2016].

[4] STELLA Modeling & Simulation Software. "http://www.iseesystems.com/softwares/Education/StellaSoftware.aspx". [Online; accessed 22-February-2016].

[5] The Workbench — Vensim. "http://vensim.com/workbench/". [Online; accessed 22-February-2016].

[6] Robert John Allan. Survey of agent based modelling and simulation tools. Technical report, 2009.

[7] Clay Beckner, Richard Blythe, Joan Bybee, Morten H Christiansen, William Croft, Nick C Ellis, John Holland, Jinyun Ke, Diane Larsen-Freeman, and Tom Schoenemann. Language is a complex adaptive system: Position paper. *Language learning*, 59(s1):1–26, 2009.

[8] Andrew Begel and Eric Klopfer. Starlogo tng: An introduction to game development. *Journal of E-Learning*, 2007.

[9] A. Borshchev. *The Big Book of Simulation Modeling: Multimethod Modeling with AnyLogic 6*. AnyLogic North America, 2013.

[10] Andrei Borshchev. Xj technologies: Anylogic 6. In *Proceedings of the 37th conference on Winter simulation*, page 82. Winter Simulation Conference, 2005.

[11] Roelof Boumans and Robert Costanza. The multiscale integrated earth systems model (mimes): the dynamics, modeling and valuation of ecosystem services. *Issues in Global Water System Research*, 2:10–11, 2007.

[12] Jason Brownlee et al. Complex adaptive systems. *Complex Intelligent Systems Laboratory, Centre for Information Technology Research, Faculty of Information Communication Technology, Swinburne University of Technology: Melbourne, Australia*, 2007.

[13] Paul Cilliers. Boundaries, hierarchies and networks in complex systems. *International Journal of Innovation Management*, 5(02):135–147, 2001.

[14] Robert Costanza and Matthias Ruth. Using dynamic modeling to scope environmental problems and build consensus. *Environmental management*, 22(2):183–195, 1998.

[15] Lawrence Davis. Handbook of genetic algorithms. 1991.

[16] IDE Eclipse. The eclipse foundation, 2007.

[17] J Doyne Farmer. The challenge of building agent-based models of the economy. European Central Bank, Frankfurt, 2011.

[18] J Doyne Farmer and Duncan Foley. The economy needs agent-based modelling. *Nature*, 460(7256):685–686, 2009.

[19] J Doyne Farmer, Norman H Packard, and Alan S Perelson. The immune system, adaptation, and machine learning. *Physica D: Nonlinear Phenomena*, 22(1):187–204, 1986.

[20] F.A. Ford. *Modeling the environment*. Island Press, 2009.

[21] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[22] Samuel Greengard. Advertising gets personal. *Communications of the ACM*, 55(8):18–20, 2012.

[23] Ilya Grigoryev. *AnyLogic 6 in three days: a quick course in simulation modeling.* AnyLogic North America, 2012.

[24] R Haber and Heinz Unbehauen. Structure identification of nonlinear dynamic systemsa survey on input/output approaches. *Automatica*, 26(4):651–677, 1990.

[25] Margaret H Hamilton and William R Hackler. Universal systems language: lessons learned from apollo. *Computer*, (12):34–43, 2008.

[26] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[27] John H Holland. Complex adaptive systems. *Daedalus*, pages 17–30, 1992.

[28] PL Houtekamer, Louis Lefaivre, Jacques Derome, Harold Ritchie, and Herschel L Mitchell. A system simulation approach to ensemble prediction. *Monthly Weather Review*, 124(6):1225–1242, 1996.

[29] S. Johnson. *Emergence: The Connected Lives of Ants, Brains, Cities, and Software.* Touchstone Book. Scribner, 2002.

[30] Taisei Kaizoji. Speculative bubbles and crashes in stock markets: an interacting-agent model of speculative activity. *Physica A: Statistical Mechanics and its Applications*, 287(3):493–506, 2000.

[31] Enrique Kremers, Norbert Lewald, Oscar Barambones Caramazana, and José María González de Durana García. An agent-based multi-scale wind generation model. 2009.

[32] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[33] Chris Marriott and Jobran Chebib. The effect of social learning on individual learning and evolution. *arXiv preprint arXiv:1406.2720*, 2014.

[34] Stefano Mazzoleni, Francesco Giannino, Marco Colandrea, Massimo Nicolazzo, and Jonathan Massheder. Integration of system dynamics models and geographic information systems. *Simile*, 600(5), 2003.

[35] D.H. Meadows and D. Wright. *Thinking in Systems: A Primer*. Earthscan, 2009.

[36] Zbigniew Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. *Evolutionary Programming*, 4:135–155, 1995.

[37] Melanie Mitchell. *Complexity: A guided tour*. Oxford University Press, 2009.

[38] G.E. Mobus and M.C. Kalton. *Principles of Systems Science*. Understanding Complex Systems. Springer New York, 2014.

[39] George Mobus. Understanding systems, in prep.

[40] George E Mobus and Paul S Fisher. Conditioned response training of robots using adaptrode-based neural networks. In *Publ by IEEE*, 1992.

[41] Robert Muetzelfeldt and Jon Massheder. The simile visual modelling environment. *European Journal of Agronomy*, 18(3):345–358, 2003.

[42] Igor Nikolic and Amineh Ghorbani. A method for developing agent-based models of socio-technical systems. In *Networking, sensing and control (icnsc), 2011 ieee international conference on*, pages 44–49. IEEE, 2011.

[43] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. A Bradford book. MIT Press, 1997.

[44] John L Russell. Kepler's laws of planetary motion: 1609–1666. *The British Journal for the History of Science*, 2(01):1–24, 1964.

[45] Ralf Salomon. Increasing adaptivity through evolution strategies. *From animals to animats*, 4:411–420, 1996.

[46] Matthias Scheutz and Paul Schermerhorn. Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. *Journal of Parallel and Distributed Computing*, 66(8):1037–1051, 2006.

[47] EBNF Syntaxt Specification Standard. Ebnf: Iso/iec 14977: 1996 (e). *URL http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf*, 70, 1996.

[48] J. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World with CD-ROM*. McGraw-Hill Education, 2000.

[49] Wayne W Wakeland, Edward J Gallaher, Louis M Macovsky, and C Aktipis. A comparison of system dynamics and agent-based simulation applied to the study of cellular receptor dynamics. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.