An Aspect-Oriented Approach for Modeling Self-Organizing Emergent Structures

Linda M. Seiter
John Carroll University
20700 N. Park Blvd.
University Hts, OH 44118
1-216-397-1948
lseiter@jcu.edu

Daniel W. Palmer
John Carroll University
20700 N. Park Blvd.
University Hts, OH 44118
1-216-397-3024
dpalmer@jcu.edu

Marc Kirschenbaum
John Carroll University
20700 N. Park Blvd.
University Hts, OH 44118
1-216-397-4684
kirsch@jcu.edu

ABSTRACT

Multi-agent systems must be engineered to ensure that desirable system-level properties will consistently emerge from the complex interactions of the underlying agents, while also guaranteeing that undesirable behavior will be suppressed. We present an Aspect-Oriented Programming (AOP) framework for modeling, visualizing and manipulating emergent structure in multi-agent systems. By encapsulating the macroscopic structure, we can identify undesirable patterns of behavior at a higher level of abstraction. The identification of such patterns allows us to implement a feedback loop to steer the behavior of the lower level agents towards actions favorable for the emergence of a reliable solution. AOP facilitates the modeling of the system-wide behavior, thus it serves as a valuable tool for building confidence that a given multi-agent system will consistently meet its requirements.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability, validation.*
D.3.3 [**Programming Languages**]: Language Contructs and Features – *data types and structures, modules/packages.*
I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence - multi-agent systems, intelligent agents

General Terms: Algorithms, Design, Reliability, Experimentation

1. INTRODUCTION

The complexity and size of contemporary software systems are reaching boundaries that are difficult to manage. The number of integrated components in an average software system, the size of test suites, the complexity of an execution trace, the monumental task of data mining; these are all features of current software development processes that no longer scale well. It is daunting to design and successfully bring to fruition software systems of great complexity using traditional software engineering techniques.

Researchers are turning toward the use of multi-agent programming to conquer the challenges of building highly complex, decentralized systems. Multi-agent systems (MAS) involve collections of autonomous agents that interact and cooperate. Over time, the interactions produce macroscopic behaviors that serve to implement the system requirements. It can be quite challenging to design and implement multi-agent systems using object-oriented languages, due to the complex structure and behavior of the agents. Agent state is often composed of beliefs, goals and plans; while agent behavior must be designed to accommodate properties such as autonomy, learning, adaptation, mobility, interaction and collaboration. It is very difficult to modularize the many concerns of agenthood, as they are overlapping and not orthogonal.

Garcia, Kulesza and Lucena have proposed the use of Aspect-Oriented Programming (AOP) as a means of implementing multi-agent systems in order to ensure attention is paid toward managing the crosscutting concerns at the agent level[4]. AOP encourages a clean separation of an object's core functionality from its crosscutting concerns[6]. A crosscutting concern is any desirable abstraction that defies encapsulation within a single software module such as a class or method; it cuts across the encapsulation boundary. As an example, Java does not supply the appropriate language primitives to encapsulate concerns such as error checking, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, and multi-object protocols. *Scattering* occurs when the implementation of a single concern is spread across multiple software modules. *Tangling* occurs when one module contains code related to more than one concern. In terms of agent-based simulations, cross cutting concerns that result in scattering include interaction and collaboration. Tangling occurs when implementing agent concerns such as autonomy, adaptation, and mobility.

Agent-oriented methodologies to date have primarily focused on techniques for engineering agents such as the coding of their beliefs, goals, rules, and interactions. There has been less focus on engineering the emergent macroscopic behavior that arises from the agent interactions, primarily due to the lack of well-defined models and processes for dealing with the macro-level. DeWolf and Holvoet propose a customization of the popular Unified Process to address the engineering of multi-agent systems, requiring each stage of the software engineering life cycle to include consideration of macroscopic structures and behaviors that arise from the agent-level[3]. Thus, even with the benefit of AOP to design and implement the cross-cutting agent properties, there still exists the tremendous challenge of verifying the macroscopic structures that dynamically emerge from the agent interactions. While we want to encourage the exploration of promising solutions through flexible and adaptive agent interactions, we must also provide a guarantee that simulations will reliably evolve toward a solution and away from trouble.

In this paper we present a framework for verifying emergent structure and behavior as it dynamically arises during a multi-agent simulation. We initially use visualization to enhance our understanding of the emergent behavior, allowing us to produce a model of the macroscopic level. The ability to modularize macroscopic behavior allows us to incrementally evolve a multi-agent system from the use of weak emergence (human observer is external to the system) toward strong emergence (observer internalized into the system). The insights obtained from the visualization are used to employ a feedback loop from the macroscopic level down to the underlying agents that optimizes agent behavior in order to produce a solution quickly and reliably. We use aspect-oriented programming techniques to clearly separate the code that implements the microscopic agent-level from that of the macroscopic system-level.

The framework consists of:

1. The object-oriented agent model: defining microscopic structure and interactions.
2. The aspect-oriented emergent model: defining global structure and interactions.
3. The aspect-oriented observers: viewing the emergent model.
4. The aspect-oriented feedback: optimizing emergent behavior.

Since many existing multi-agent systems use object-oriented models for implementing the agent level, the framework is presented in terms of an object-oriented agent model. However, the agent model itself may also be designed using the aspect-oriented architecture proposed by Garcia, Kulesza and Lucena[4].

2. MODELING EMERGENCE IN MULTI-AGENT SYSTEMS

A multi-agent system is based on computation through interaction rather than discrete algorithm, thus it is often not possible to formally prove the correctness of the system. De Wolf and Holvoet propose as an alternative the use of empirical methods for verifying the macroscopic behavior[3]. Macroscopic properties are quantified with macroscopic variables, which are measured from simulations and used to obtain statistical results. But how are these macroscopic variables represented within software? This is a fundamental challenge because they correspond to global state arising from non-linear interactions over a potentially significant expanse of time. In this paper we demonstrate the use of AOP for modularizing the macroscopic variables. Aspects allow us to encapsulate data from many objects and their interactions, intercepting numerous points of execution over the course of a simulation.

Emergence is described as the surprise that occurs when observers witness a property on a macroscopic or global level that is not observable, reducible, nor explainable at the underlying microscopic level. This shows that "a whole is more than the sum of its parts". The goal of building and programming agent-based programs is to engineer these surprises to our advantage in order to tackle complex problems. An example of emergence in the distributed MAS domain is the ability to effectively route messages without any knowledge of the network topology based on cooperating agents using only local information.

Baas and Emmeche define the following framework for emergent structures[2]: Let $\mathbf{S_1}$ be a collection of general systems or "agents", $\mathbf{Int_1}$ be interactions between agents, and $\mathbf{Obs_1}$ be observation mechanisms for measuring the properties of agents to be used in the interactions. The interactions then generate a new kind of structure $\mathbf{S_2 = R(S_1,Obs_1,Int_1)}$ which is the result of the interactions among the agents. $\mathbf{S_2}$ is an emergent structure, which may be subject to new interactions $\mathbf{Int_2}$ as well as new observational mechanisms $\mathbf{Obs_2}$.

$\mathbf{P}$ is an emergent property$\leftrightarrow \mathbf{P} \in \mathbf{Obs_2(S_2)}$ and $\mathbf{P} \notin \mathbf{Obs_2\ (S_1)}$.

A property is considered emergent if it is observable or explainable within the emergent structures and their interactions, yet the same observation mechanism can not observe nor explain the property in the realm of the initial underlying structure. As an example, consider the patterns that arise from a rush hour traffic jam. It would be difficult to predict or explain the waves of traffic that result in a traffic jam by simply observing a single car moving in relation to the cars that immediately surround it. A helicopter view however can clearly observe the wave patterns. Observation is both a critical and labor-intensive component for explaining the emergence of these macroscopic structures. It can be a daunting task to pour through massive quantities of data, looking for patterns and attempting to derive some intuition toward understanding why a set of simple low-level agent behaviors result in a particular collective behavior. A visual presentation of the emergent structures and behaviors provides researchers with a better means for explaining how and why they arise. Ultimately, we evolve the external human observer of the system into an internal feedback mechanism that helps guide the simulation toward reliable behavior. In the next section we describe a model for visualizing, encapsulating and eventually optimizing the emergent structures that arise during a graph coloring simulation.

2.1 Graph Coloring

To demonstrate the application of AOP, we present a multi-agent system developed to solve graph coloring. Each vertex in a graph is assigned a color such that adjacent vertices have different colors. The goal is to determine a minimum sized color set for a given graph. Coordination problems reduce to graph coloring in that the vertices represent tasks, colors represent resources available to perform the tasks, and an edge between two vertices represents a constraint in performing the two tasks with the same resource. The minimum number of colors required to successfully color the graph determines the minimum number of resources needed to complete the coordination task.

Deterministic algorithms for graph coloring require an expensive exhaustive search. We have implemented several non-deterministic algorithms that employ agents to find the minimum coloring for both planar and non-planar graphs, with favorable results, including several fully distributed versions across a heterogeneous network[8]. Figure 1 shows a view of a simple planar graph that a set of agents are trying to solve using 4 colors. Note that while Figure 1 shows some of the edges overlapping, there exists an orthogonal layout of this graph that avoids overlap.

The letter in each vertex represents the current choice of color (Y-yellow, G-green, B-black, R-red). A bold edge depicts a color conflict between adjacent vertices. An agent is assigned to each vertex and will detect color conflicts among adjacent vertices. The agent on a vertex that is conflict free is in a happy state and will retain their color choice for the next iteration, while an agent on a vertex with a color conflict will randomly choose whether to change its color. While the simulation eventually produces a solution; it is not clear why. The view in Figure 1 does not lead to many insights other than to hint that clusters of stable vertices periodically form, grow in size and merge, leading to an eventual solution. However, stable clusters occasionally dissolve due to conflicts with vertices on the boundary of a cluster. The view of the agent model shown in Figure 1 does not contain sufficient information to explain the evolution of the stable regions.

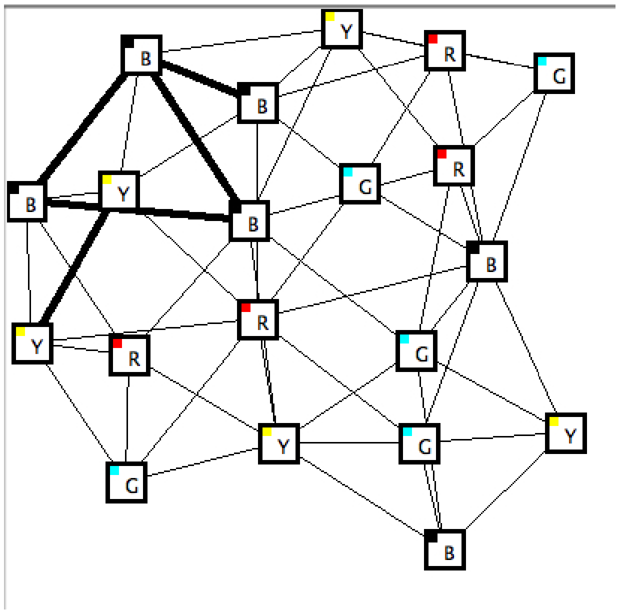


Figure 1: 4-color graph at 58th iteration
(see http://www.jcu.edu/math/faculty/lseiter/graph/4color.html for alternate version of graph with colored vertices).

2.2 Modeling Emergence in Graph Coloring

The graph coloring solution arises over time out of the simple agent interactions. Complex negotiations occur within clusters of vertices, leading to an eventual group resolution. Our initial software implementation provided the view in Figure 1, which does not provide insight as to how a stable region comes into being nor is it easy to observe the interactions that occur between clusters of vertices. After observing many simulations of graph coloring using only the view from Figure 1, we decided that it would be more helpful to observe the simulation using a view that depicted the vertices in terms of the absence of color conflict (i.e. the presence of a partial solution), rather than the actual color choice. We also wanted to observe how the solution spread through the graph, so the notion of using an increasing color scale that depicted the history of the solution was proposed. The hope was that this view would provide insight into how stable regions formed as well as how they dissolved.

Following the emergent framework proposed by Baas and Emmeche, we developed a model for visualizing emergence in the graph coloring simulation, shown in Figure 2. The microscopic level consists of the original graph coloring software, namely the 4-color graph along with the agents that interact to resolve local coloring conflicts. The micro-level observation mechanism is the view from Figure 1.

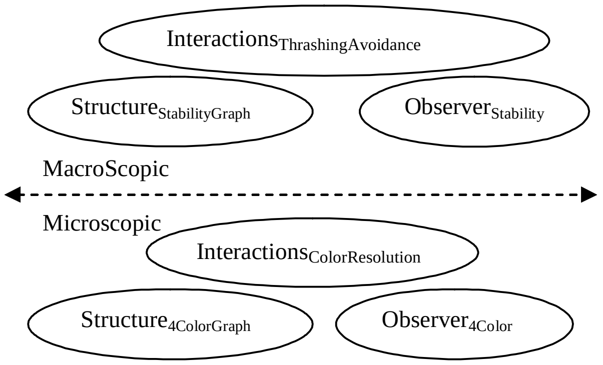

Interactions$_{\text{ThrashingAvoidance}}$
Structure$_{\text{StabilityGraph}}$
Observer$_{\text{Stability}}$
MacroScopic
Microscopic
Interactions$_{\text{ColorResolution}}$
Structure$_{\text{4ColorGraph}}$
Observer$_{\text{4Color}}$

Figure 2: Emergence Framework For Graph Coloring

As the simulation proceeds the agent interactions result in the formation and spread of stable regions. The stability graph is an emergent structure, arising dynamically from the microscopic level. Figure 3 shows the stability graph that corresponds to the 4-color graph from Figure 1. Note that the vertex colors in the stability graph do not correspond to color assignment (resource allocation) as in the original 4-color graph, but rather we use a grayscale to depict the happiness/stability of a vertex and its current color choice. White represents instability; implying that the vertex has recently changed its color and its happiness level is 0. Shades of gray indicate increasing stability, the happiness level of a vertex is increased by 1 each iteration the vertex does not change its color. Once a happiness threshold is reached, the color switches to black to indicate long-term stability.

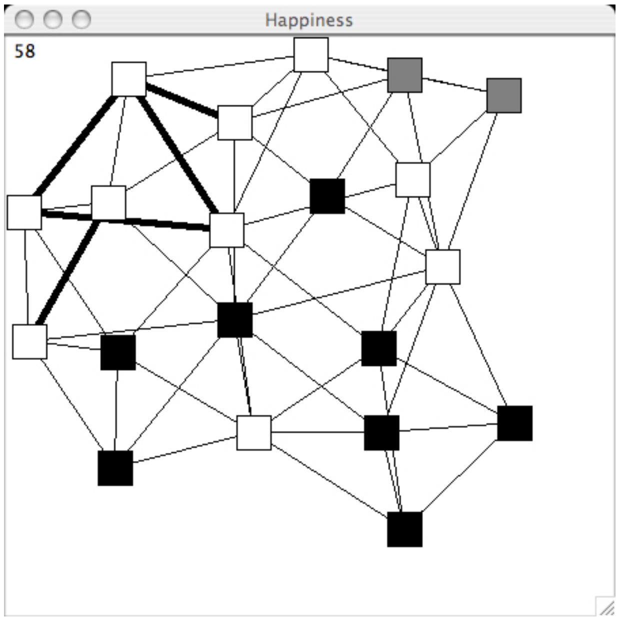


Figure 3: Stability graph at 58th iteration.

The ability to observe the stabililty graph evolve over the course of the simulation provided numerous insights into how the graph coloring solution was developing at the lower agent level. Happiness/stability was quickly achieved by vertices with low degree (few edges). The stability spread outwards from a happy vertex to form clusters encapsulating other low-degree neighbors. However, the boundary between two clusters would usually result in a color dispute that rippled back into the previously stable cluster and caused it to dissolve.

Once we were able to view the stability graph evolve during a simulation, it quickly became apparent that the agents at the lower level were hyperactive. While it is useful to encourage agents toward random exploration of the problem space early in the simulation, as time goes by it is necessary to shift toward focused exploitation as partial solutions begin to arise as a result of the agent collaborations. The agents were hyperactive in that they were always willing to give up a partial solution (i.e. dissolve a stable region) whenever a color conflict occurred. A resistor was needed to calm the hyperactive agents. This was implemented as a feedback loop to monitor the vertices that were in stable regions, dampening their probability to change when color conflicts were found based on their happiness. The longer a vertex had been happy the more resistant it was to changing its color if a conflict suddenly occurred. This allowed the stable regions to retain their stability, calming the agents and preventing the quick dissolution of partial solutions that we initially witnessed. The *thrashing avoidance* functionality is an example of “downward causation” in which the existence of structure at the global level is able to influence the underlying microscopic level, as depicted in Figure 2. Our initial implementation of thrashing avoidance occasionally led to deadlock, where a set of vertices became too resistant to changing their color. This was easily resolved by softening the resistance of a stable vertex by adding a small amount of randomness to the conflict resolution strategy.

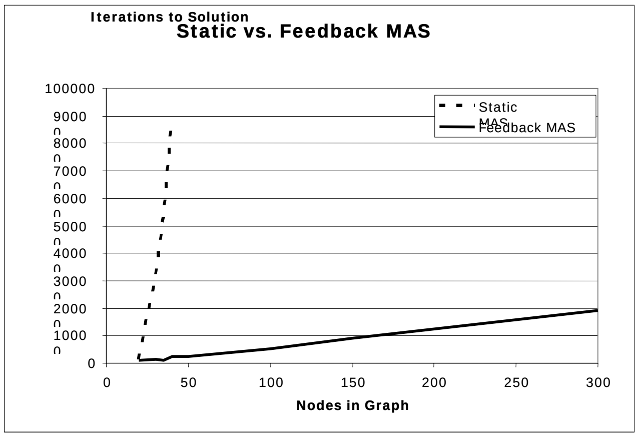


Figure 4: Static vs. Feedback Performance

We have tested the multi-agent graph coloring algorithm both with and without the feedback mechanism on randomly generated planar graphs. Figure 4 demonstrates that the feedback mechanism appears to scale exceptionally well for planar graphs. The number of iterations required to color the planar graphs grow linearly with the number of nodes (or agents) on the graph.

3. MODELING EMERGENT BEHAVIOR WITH AOP

AspectJ[1] was used to encapsulate the macroscopic structure and behavior depicted in Figure 2. We developed aspects responsible for representing the emergent model (the stability graph), logging its state, presenting the view of the emergent stability model, synchronizing the new view with the original 4-color view, and implementing the feedback loop to improve the performance of the coloring simulation (thrashing and deadlock avoidance). The architecture of the AOP implementation for graph coloring is shown in the appendix.

An aspect is a software module that encapsulates a set of pointcuts, advice and introductions[6]. A *pointcut* is a label that describes a set of joinpoints, where a *joinpoint* is a point of execution in a program such as a method call or a method return. Thus, a pointcut may be declared that represents all points of execution in a program that involve an invocation to the *setColor* method defined in a *Vertex* class. When program execution reaches a particular point of execution, one might want to augment the code with additional functionality that should be executed before, after or instead of (around) the execution point. Augmenting a joinpoint with before/after/around methods is referred to as adding *advice*. In addition to augmenting the control flow of a program by adding advice, an aspect may add new fields to existing classes. This is referred to as an *introduction*. The AspectJ compiler takes a set of aspects which contain pointcuts, advice and introductions. The compiler then weaves the new functionality into the Java bytecode of a set of existing classes. The Java source code remains untouched.

Vertex happiness reflects the number of iterations that the vertex has been conflict free. Edge happiness reflects the number of iterations for which the two adjacent vertices have been compatible. We use vertex and edge happiness aspects to extend the static agent model (colored vertices, undirected edges) to encapsulate the state that arises from the agent interactions during color resolution. Using the AspectJ language, we wrote an aspect *VertexHappiness* that:

1. Introduces new instance variables to each Vertex and Edge object to encapsulate happiness.
2. Defines a pointcut on the correctColor() methods of the Vertex and Edge classes that detect color conflicts.
3. Adds advice to the color conflict resolution method. If a conflict does not exist the vertex is happy with its color and its happiness is incremented. If a conflict is found then happiness is reset.

After adding structure to represent stability and capture all agent interactions related to color conflict resolution, the challenge was to create views of the stability graph that could be synchronized with the view of the underlying 4-color graph. It was vital to allow the user to watch the transformation of the 4-color graph and the stability graph side by side. The traditional model-view-controller architecture in practice results in poor separation of concerns. Although the model is supposed to be unaware of the GUI, an observer infrastructure must be implemented within the model to allow synchronization of the model and its views.

AOP on the other hand supports the separation of concerns when adding and coordinating multiple views. This was successfully accomplished without modifying *any* of the original graph coloring source code and without requiring preexisting extension points for observation. It was necessary to develop an aspect that synchronized the logging of changes to the happiness properties (once per iteration of the swarm simulation) with the logging of the underlying 4-coloring graph model. The happiness logging aspect *VertexHappinessWriter*:

1. Defines a pointcut on the construction of the vertices and edges in the basic graph model, to intercept the calls that create the initial 4-color graph model. The advice allows the stability graph to be created with the same size adjacency list.
2. Defines a pointcut on the setting of the new happiness attribute introduced to the Vertex class, along with a pointcut on the logging methods of the animation probes that resided in the original software packages. Per iteration, the logging of the happiness attribute is synchronized with the logging of the agent data model.

To produce a view of the stability graph that had the same layout as the 4-color graph and synchronize its update with that of the 4-color view the *HappinessViewer* aspect does the following:

1. Defines a pointcut on the method that performs the layout of the graph in the 4-color view. The advice produces a duplicate graph with the same layout for the stability view.
2. The aspect creates several objects for reading in the vertex and edge happiness log files, and declared a pointcut on the method for processing each frame in the 4-color view. The advice synchronizes the reading of the log files and the processing of each frame of the stability graph view with the 4-color view.

The happiness viewer aspect was put to the challenge in the middle of the project, when we changed graph models and decided to use an alternative layout in the underlying 4-coloring program. Without having to change a single line of code in any of the aspects, AspectJ simply rewove the aspects and the new graph model layout was used to depict the stabililty graph.

The final aspect to be defined was the *HappinessFeedback* aspect, which implemented the thrashing avoidance by calming the hyperactive agents once a certain level of stability was reached:

1. Defines a pointcut on the resetHappiness method that was defined in the VertexHappiness aspect.

2. Provides advice to override the default color conflict resolution strategy. Under normal circumstances, the happiness was reset whenever a color conflict occurred. However, the *around* advice checks the happiness level of the vertex and its surrounding neighborhood to determine whether the vertex should be resistant to change.

This aspect allowed us to help maintain and spread stability while performing the graph coloring.

3.1 Emergence in robotic simulations

We have employed the AOP framework on another multi-agent system that simulates swarms of rovers for extraterrestrial exploration using alternative mobility models. The goal of the simulation is the uniform dispersion of agents over some area of Martian terrain. Each agent strives to be at least some minimum distance from other agents, while also staying within contact distance of some number of neighbors. The simulation had initially been developed without AOP. When an agent wandered too close to another, the algorithm used randomness to resolve the conflict, causing one or both agents to change course. As in the 4-color mapping problem, regions of stable agents formed. Yet very often a large group of stable, happy agents (uniformly distributed) would be disrupted by a single wandering agent that strayed too close to the group. We used AOP to develop a stability model for the robotic simulation, along with new views that depicted the simulation in terms of agent stability. We were also able to generate downward causation, allowing established regions of dispersion to resist wanderers that might dissolve the stability of a group.

3.2 A General Model Of Emergence

The graph coloring and Martian rover simulations were both initially developed without the use of AOP and without the emergent stability model. The original graph coloring simulation implemented the microscopic agent level, namely a simple color conflict resolution algorithm and presented a single view depicting the current color choices for a graph. The original martian rover simulation implemented a simple agent dispersion algorithm, and presented a single view depicted the current location of an agent. The emergent structures of stability had not yet been discovered when the original simulations were developed.

In an effort to understand how partial solutions evolved, we added the stability model first to the graph coloring, and then to the Martian rover simulations. All of the code related to modeling, visualizing and manipulating the emergent structures was written using Aspect-Oriented Programming. AOP allowed us to physically decouple the agent and emergent software modules, thus matching the conceptual decoupling of the two levels. Modularizing the emergent structure and behavior in aspects allowed plug-and-play experimentation with alternative feedback mechanisms for avoiding thrashing and deadlock. While it is possible to implement the stability model without the use of AOP, doing would result in a loss of modularity as the macroscopic level would be tangled into the agent code. This would hinder the flexible experimentation of alternative feedback techniques as well as alternative views of the emergent model.

Most importantly, the distinct decoupling of the stability model from the agent model in the graph coloring simulation allowed us to easily generalize the stability model for use in the Martian rover simulation, reusing much of the framework design as well as some of the actual code. In both simulations, an agent interacts with its “neighbors” (conceptual or physical) to resolve conflicts. A partial solution will spread over a population of neighboring agents. However, a conflict at the boundary of a stable region may destructively ripple through a previously stable area. The introduction of the happiness/stability fitness function allows a record of the partial solution history to be maintained and analyzed in order to improve the performance of the agent computation through the introduction of a resistor, namely the feedback loop that alters the underlying agent rules based on the existence of the higher level emergent structure. AOP is also beneficial for implementing observers, thus new views of emergent stability are easily added that synchronize with the agent level views.

While we have successfully applied the AOP emergence framework to two simulations, the framework is generally applicable to other multi-agent simulations that involve random exploration of a problem space through agent interactions. The appendix describes the AOP framework specific for the graph coloring simulation. However, the AOP code shown above the macroscopic boundary could be generalized for use with other simulations that involve agent interaction for conflict resolution.

1. Stability/happiness can be modularized by defining pointcuts on agent methods that implement conflict resolution.
2. An aspect can be developed to introduce new structure to the agent model for recording happiness/stability.
3. An aspect can be developed to synchronize the stability model with the agent model.
4. After identifying high-level patterns from the emergent model, an aspect can be developed to implement downward causation, altering the agent rules based on feedback from the emergent computation.

Note that while it is possible to reuse much of the design as well as some of the code to implement emergence capture among different simulations, the identification of the emergent patterns and the corresponding implementation of the appropriate pointcuts will require human effort and most likely not be easily automated.

4. Conclusion

To date much of the research involved in applying AOP for multi-agent systems has focused on reducing complexity at the agent level. Kendal initially proposed the use of aspect-oriented programming for implementing the various roles and collaborative capabilities that an agent must support[5]. Garcia, Kulesza and Lucena extended Kendall's work to encompass the many complex properties of agenthood using AOP, including interaction, adaptation, autonomy, learning, mobility and collaboration[4]. They also presented an incremental software engineering process for evolving complex agent systems that may comprise of numerous implementation frameworks. The SmartWeaver approach proposed by Pace et al employs AOP to assist in the development of multi-agent systems, providing support for the separation of certain concerns such as logging, event handling and concurrency from the general agency properties[7]. In contrast, our focus has been on using AOP to modularize and optimize the emergent behavior of a multi-agent system.

Multi-agent systems allow agents the opportunity to choose from a diverse set of actions so that progress can be discovered and reinforced. Yet behaviors that arise from the agent interactions are not always desirable, and we need mechanisms for steering a computation toward a positive solution while avoiding problems such as thrashing and deadlock. In this paper we demonstrated the ability to capture and reason about emergent structure as it evolved over the course of a simulation. We were able to make use of behavioral feedback which gathered information about the macroscopic level and made it available to agents by dynamically tuning their rules, increasing the likelihood that agents chose actions favorable for the reliable emergence of a solution.

We presented a framework for implementing multi-agent computation using AOP, demonstrating a modular approach to modeling emergence. The framework consists of:

1. The object-oriented agent model: defining microscopic structure and interactions.
2. The aspect-oriented emergent model: defining global structure and interactions.
3. The aspect-oriented observers: viewing the emergent model.
4. The aspect-oriented feedback: optimizing emergent behavior.

We are applying this framework to a variety of multi-agent systems, our goal being the development of an emergence-oriented programming process that enables industry to successfully build and evolve complex software systems[8].

5. ACKNOWLEDGMENTS

Jason Shifflet developed the initial static 4-coloring application. His modular code simplified the task of developing pointcuts. Peter Kovacina conquered AspectJ and developed the aspects for the graph coloring software. Pat Cloutier implemented the Martian Rover simulation. We thank the Huntington-Codrington Foundation for supporting these outstanding undergraduates.

6. REFERENCES

[1] AspectJ. http://www.aspectj.org.

[2] Nils Baas and C. Emmeche, On Emergence and Explanation. *Intellectica* 1997/2, no.25, pp.67-83

[3] Tom De Wolf and Tom Holvoet. Towards a Methodology for Engineering Self-Organising Emergent Systems. *in Proc. of the Inter. Conference on Self-Organization and Adaptation of Multi-Agent and Grid Systems,* IOS Press, 2005.

[4] Alessandro Garcia, Uira Kulesza, Carlos de Lucena, *Aspectizing Multi-Agent Systems: From Architecture to Implementation..* Software Engineering for Multi-Agent Systems III, Springer-Verlag, LNCS, pp121-143, 2004.

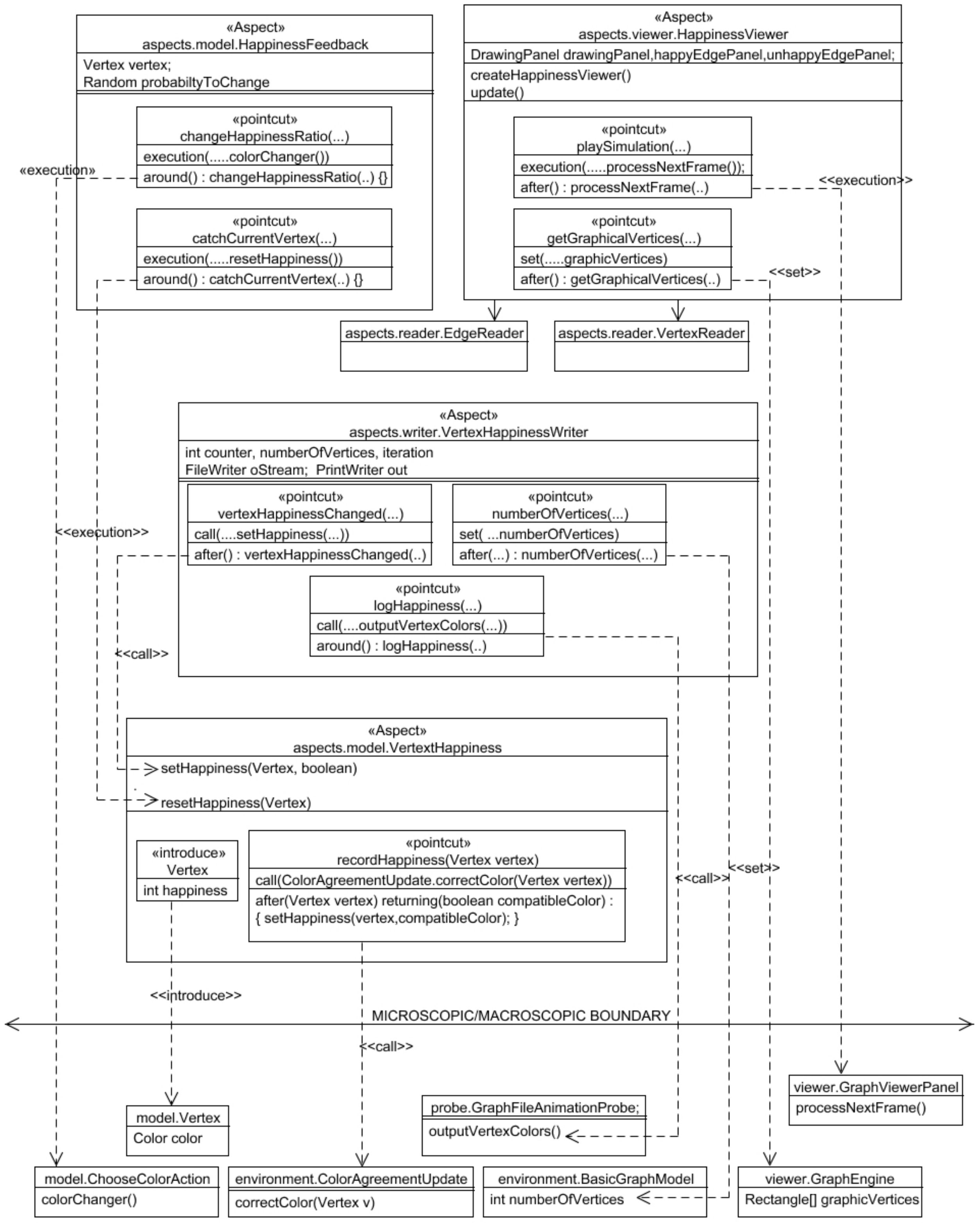
[5] Elizabeth Kendall, *Aspect-Oriented Programming for Role Models*, ECOOP, 1998.

[6] G. Kiczales, J. Lamping, M. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, J. Irwin. Bowman, B., Debray, S. K., and Peterson, L. L. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming.,* Springer-Verlag LNCS, 1997.

[7] A. Pace, F. Trilnik, M. Campo. *Assisting the Development of Aspect-Based Multi-Agent Systems Using the Smartweaver Approach.* In Proc. SELMAS, Springer, LNCS 2603, 2003.

[8] Daniel W. Palmer, Marc Kirschenbaum, Linda M. Seiter, Emergence-Oriented Programming, *IEEE SMC 2005, Inter. Conf. on Systems, Man and Cybernetics.* October 2005.

7. APPENDIX


«Aspect»
aspects.model.HappinessFeedback
Vertex vertex;
Random probabiltyToChange
«pointcut»
changeHappinessRatio(...)
execution(.....colorChanger())
around() : changeHappinessRatio(..) {}
«pointcut»
catchCurrentVertex(...)
execution(.....resetHappiness())
around() : catchCurrentVertex(..) {}
«execution»
«Aspect»
aspects.viewer.HappinessViewer
DrawingPanel drawingPanel,happyEdgePanel,unhappyEdgePanel;
createHappinessViewer()
update()
«pointcut»
playSimulation(...)
execution(.....processNextFrame());
after() : processNextFrame(..)
«pointcut»
getGraphicalVertices(...)
set(.....graphicVertices)
after() : getGraphicalVertices(..)
<<execution>>
<<set>>
aspects.reader.EdgeReader
aspects.reader.VertexReader
«Aspect»
aspects.writer.VertexHappinessWriter
int counter, numberOfVertices, iteration
FileWriter oStream; PrintWriter out
«pointcut»
vertexHappinessChanged(...)
call(....setHappiness(...))
after() : vertexHappinessChanged(..)
«pointcut»
numberOfVertices(...)
set(...numberOfVertices)
after(...) : numberOfVertices(...)
«pointcut»
logHappiness(...)
call(....outputVertexColors(...))
around() : logHappiness(..)
<<execution>>
<<call>>
«Aspect»
aspects.model.VertextHappiness
setHappiness(Vertex, boolean)
resetHappiness(Vertex)
«introduce»
Vertex
int happiness
«pointcut»
recordHappiness(Vertex vertex)
call(ColorAgreementUpdate.correctColor(Vertex vertex))
after(Vertex vertex) returning(boolean compatibleColor) :
{ setHappiness(vertex,compatibleColor); }
<<call>>
<<set>>
<<introduce>>
MICROSCOPIC/MACROSCOPIC BOUNDARY
<<call>>
viewer.GraphViewerPanel
processNextFrame()
probe.GraphFileAnimationProbe;
outputVertexColors()
model.Vertex
Color color
model.ChooseColorAction
colorChanger()
environment.ColorAgreementUpdate
correctColor(Vertex v)
environment.BasicGraphModel
int numberOfVertices
viewer.GraphEngine
Rectangle[] graphicVertices