Fungal biocontrol in coffee: a case study in agroecology

by

Douglas W. Jackson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Ecology and Evolutionary Biology)
in The University of Michigan
2012

Acknowledgements

In one sense, acknowledgements are an absurdity. Every human accomplishment, without exception, is the result of the collective efforts of innumerable actors. Singling out a few for recognition is akin to calling five raindrops out by name to thank them for watering one's corn: it leaves a lake of contributors unacknowledged. In another sense, however, any occasion to acknowledge a few of the larger raindrops is an opportunity to be treasured. I feel a tremendous amount of gratitude for the support of the following people whose efforts made my doctoral experience possible and enjoyable.

It will come as no surprise to those who know him that the support of my doctoral advisor, John Vandermeer, has been extraordinary. As a mentor, role model, fount of enthusiasm, and friend, John is simply outstanding. With his exceedingly rare combination of incisiveness, generosity of spirit, and tireless dedication to understanding and improving the world, he embodies a standard for how to live life that is truly inspiring.

Each of my committee members provided invaluable guidance that greatly improved my research. Ivette Perfecto played a crucial role throughout my graduate school career as a leader of the Finca Irlanda research group, a rich source of natural history knowledge and intuition, and as a reliable judge of the scientific merit of experimental ideas, results, and analyses. As with John, she is a wonderful inspiration for how to successfully employ science for the people. Tim James has been a fantastic collaborator, and has shown amazing patience in teaching me the basics of lab work and population genetics analysis. But more importantly, he has been a great friend and source of counsel. Pej Rohani helped to provide a firm technical grounding for my modeling work, both through his direct help and via his published articles and textbook that I continue to reference often. The rigor of my modeling work was undoubtedly enhanced just by knowing that it would be read by such an accomplished theoretician.

Professor Bill Kovalak's faith in me when I was still an engineer tentatively testing the waters of ecology gave me the courage to make the leap. Though she is unaware of it, Catherine Badgley's contribution to the book "Fatal Harvest" also came at a critical time, and gave me the necessary conviction to continue my plan of entering graduate school when it appeared to be a hopeless dream. Beverly Rathcke and Krista McGuire were instrumental in helping me in the very early stages of my graduate school career; without their encouragement, I might not have made it over the first daunting hurdles that I faced while switching careers. I wish Beverly could have been here to see me pass this milestone, as none of this would have been possible without her. She is missed dearly.

Early in my graduate school experience, my new lab mates – Shalene Jha, Jahi Chappell, Krista McGuire, Zach Miller, Heidi Liere, and Javier Ruiz – were incredibly helpful and welcoming. They set a precedent of high achievement that I still rely on to this day as a benchmark. Along with Stacy Philpott, their example serves as a model worthy of emulation. All of the other members of the Perfectomeer lab, both official and unofficial, have been like a family. Their support and camaraderie continue to be indispensable. Without Senay Yitbarek's daily "new theories," Dave Allen's eagerness to engage in stimulating conversation about any and every interesting topic – from baseball to baking – and frequent runs/conversations about science and life with Aaron Iverson, my time as a graduate student would have been much less fun and much less productive. Sahar Haghighat provided a final push to help me complete the writing process. Field work on the farm in Mexico was greatly enriched by everyone in the extended family: Kate Ennis, Dave Gonthier, Jes Skillman, Ashley Larson, Theresa Ong, Jane Remfert, Linda Marin, Hsunyi Hsieh, Casey Taylor, Katie Goodall, George Livingston, Kim Williams-Guillén, the Toledo Contingent, Gustavo López-Bautista, Aldo A De la Mora Rodríguez, Gabriel Humberto Dominguez Martinez, Braulio Estaban Chilel, and all of the other Finca Irlanda compañer@s. I especially thank Kate Zemenick and Andy MacDonald, aka Team *Lecanii*, for their hard work and friendship.

The EEB community also played a crucial role in making this work possible. I am in awe of how hard the office staff works to ensure that all of the administrative details are taken care of, and how composed they are despite the number of plates they have to keep spinning. In particular, Jane Sullivan deserves a medal for her tremendous efforts and patience in managing the sundry last-minute complications that tend to arise during a graduate school career. My Ph.D. cohort, especially Jingchun Li, Ed Baskerville, and Andres Baeza, have been wonderful traveling companions.

As an international collaborative effort, my work would not have been possible without the kind assistance of our colleagues in Mexico. I thank Don Walter Peters for permission to work on his farm and for his commitment to conservation. El Colegio de la Frontera Sur (ECOSUR) in Tapachula, Mexico provided much-needed assistance and on-site laboratory facilities. In particular, Graciela Huerta, a collaborator and co-author of Chapter II, was extremely generous with her time and mycological expertise. Ricardo Alberto Toledo Hernández and Juan Cisneros Hernández at ECOSUR provided valuable assistance with laboratory work.

I am very grateful for the world-class intellectual community at U of M, particularly the New World Agriculture and Ecology Group, the Complex Systems Advanced Academic Workshop, the Tropical Biology Seminar, and the Mycology Discussion Group. Rick Riolo and the Center for the Study of Complex Systems provided much technical assistance and the computer resources that made all of my computer simulation work possible.

Science could not happen without the generous support of funding providers. I am particularly thankful for the support provided by the Graham Environmental Sustainability Institute. Through their efforts to foster a community of leaders in sustainability, they give me hope for the future. The International Institute, the Rackham Graduate School, and the EEB department also provided critical funding.

In addition to their aforementioned support, I thank the following people who were co-authors of various parts of this dissertation, as reflected in the bibliography: John Vandermeer, Ivette Perfecto, Kate Zemenick, Graciela Huerta, Jes Skillman, and Dave Allen.

I am extremely fortunate to have had the following people standing behind me throughout this lengthy journey: my brother Nick, who has been an inspirational model of perseverance; my mom, who always had faith in me; my sister Sandi, who personifies the best that family can offer; and my brother Gerry, who would be a treasured friend even without Hamilton's rule. Finally, because words would not begin to do her unwavering support and friendship justice, I will conclude with a simple thanks to Aley Joseph.

Table of contents

Acknowledgements **ii**

List of Figures **vii**

List of Tables **x**

List of Appendices **xi**

CHAPTER I. Introduction **1**

Summary of dissertation 5

CHAPTER II. Occurrence in the soil and dispersal of *Lecanicillium lecanii*, a fungal pathogen of the green coffee scale, *Coccus viridis*, and coffee rust, *Hemileia vastatrix* **17**

Methods 19

Results 28

Discussion 30

CHAPTER III. Spatial and temporal dynamics of a fungal pathogen promotes pattern formation in a tropical agroecosystem **37**

Methods 39

Results 44

Discussion 52

CHAPTER IV. Indirect biological control of the coffee leaf rust, *Hemileia vastatrix*, by the entomogenous fungus *Lecanicillium lecanii* in a complex coffee agroecosystem **59**

Methods 63

Results 65

Discussion 76

CHAPTER V. The evolution of imperfect prudence **80**

The model 83

Results 85

Discussion 95

CHAPTER VI. Self-organization of background habitat determines the nature of population spatial structure **101**

The metapopulation-source/sink continuum: theory 103

Self-organization of habitat patches and consequences for equilibrium patch occupancy 111

An empirical example 116

Conclusions 120

CHAPTER VII. Detection of imminent, non-catastrophic regime shifts **122**
Methods 125
Results 132
Discussion 138
CHAPTER VIII. Conclusion **142**

APPENDICES **145**
BIBLIOGRAPHY **279**

List of figures

II.1. Location of *A. instabilis* ant nests (solid circles) in 45 ha plot **21**

II.2. Locations of soil samples on transects leading away from foci of two *L. lecanii* epizootics **21**

III.1. Graph of transformed Ripley's K versus radius of sampling circle **44**

III.2. Representation of two intensively sampled sites in the study area **46**

III.3. Scales and fungus in coffee bushes surrounding two shade trees **48**

III.4. Number of healthy (white) and infected (black) scales on the branches of 4 shoots on a single coffee bush. **49**

III.5. The black line represents the high-fitness region in v, t_1 parameter space in which it is possible to generate spatial patterns of ant nests that are qualitatively and quantitatively similar to the pattern observed in the field. **50**

III.6. Example snapshot of the ant and fungus CA model for $v = 0.35$, $t_0 = 0$, $t_1 = 0.63$, and $a = 0.007$ **51**

III.7. Time series for a representative run of the ant and fungus CA model for $v = 0.35$, $t_0 = 0$, $t_1 = 0.63$, and $a = 0.007$ **51**

III.8. Data from two sites at two different times, illustrating the stages of development of the fungal pathogen, *L. lecanii* and its host *C. viridis* **54**

IV.1. The basic biology of the system **62**

IV.2. Protocol for *C. viridis* and *L. lecanii* surveys, adapted from Perfecto and Vandermeer (2006) **64**

IV.3. Abundance of *L. lecanii* and prevalence of *H. vastatrix* in Site A in a) 2009 and b) 2010 **66**

IV.4. Number of leaves per plant with *H. vastatrix* lesions versus the distance to the center of mass of the *L. lecanii* concentration in a) 2009 [$R^2 = 0.148$, $P < 0.001$] and b) 2010 [$R^2 = 0.133$, $P < 0.001$] in Site A **67**

IV.5. Abundance of *L. lecanii* and prevalence of *H. vastatrix* in Site B in a) 2009 and b) 2010 **68**

IV.6. Number of leaves per plant with *H. vastatrix* lesions versus the distance to the center of mass of the *L. lecanii* concentration in a) 2009 [$R^2 = 0.018$, $P = 0.004$] and b) 2010 [$R^2 = 0$, $P = 0.921$] in Site B **69**

IV.7. Prevalence of *L. lecanii* in 2009 and change in prevalence of *H. vastatrix* from 2009 to 2010 in Site A **70**

IV.8. Change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to the center of mass of the *L. lecanii* concentration in 2009 in Site A [$R^2 = 0.315$, $P < 0.001$] **71**

IV.9. a) Coefficients of determination (R^2) and b) effect sizes (slopes) for linear regressions of the change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to points within Site A **72**

IV.10. Abundance of *L. lecanii* in 2009 and change in prevalence of *H. vastatrix* from 2009 to 2010 in Site B. Diameters of open circles are proportional to the estimated number of infected *C. viridis* on coffee plants, with the largest circle corresponding to 468 infected scales **73**

IV.11. Change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to the center of mass of the *L. lecanii* concentration in 2009 in Site B [$R^2 = 0.011$, $P = 0.017$] **74**

IV.12. a) Coefficients of determination (R^2) and b) effect sizes (slopes) for linear regressions of the change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to points within Site B **75**

V.1. Evolutionary dynamics of 100 representative realizations of the model initialized above and below the ESS host reproduction probability, *g*, of approximately 0.20 **86**

V.2. Pairwise invasibility plot showing the probability that an invading strategy with a host reproduction probability g_I can beat a resident strategy g_R **87**

V.3. The spatial structure of the host population for various host reproduction probabilities, *g* (shown in the upper righthand corner of each square) **88**

V.4. Relative cumulative frequency of hosts of a given age at death for various host reproduction probabilities (*g*) **89**

V.5. Relative cumulative frequency of the total number of descendants per host upon death for various host reproduction probabilities (*g*) **90**

V.6. Representative snapshot of the model after the evolutionary equilibrium has been achieved **91**

V.7. Illustrative example of the growth of clusters of rapidly-reproducing hosts (orange cells, $g = 0.8$) being constrained by more slowly reproducing hosts (blue cells, $g = 0.2$) **91**

V.8. Change in the relative cumulative frequency of hosts of a given age at death for various host reproduction probabilities (*g*) attempting to invade a resident host population with $g = 0.2$. **92**

V.9. Change in the total number of descendants per host upon death for various host reproduction probabilities (*g*) attempting to invade a resident host population with $g = 0.2$ **93**

V.10. Grand means of the host population size versus the host reproduction probability (± 1 s.e.m.) **94**

VI.1. Diagrammatic illustration of the two extreme forms of population organization in a fragmented habitat **105**

VI.2. Graphical composition of the three essential functions to produce the relationship between migration rate (*m*) and extinction rate (*e*), based on the fundamental monotonic relationship between c_m and n_T **106**

VI.3. Average fraction of patches occupied as a function of the scaling parameter of the original "self-organized" habitat distribution, from simulation experiments **111**

VI.4. The mean fraction of habitat patches occupied for landscapes with different amounts of habitat **113**

VI.5. Representative landscapes corresponding to the data shown in Figure VI.4 **115**

VI.6. Ripley's K for the scenarios identified in Figures VI.4-VI.5 **115**

VI.7. Cluster size distributions for the scenarios identified in Figures VI.4-VI.6 **116**

VI.8. Relationship between cluster scale and number of singletons (blue curves descending) and number of points in the largest patch (red curves ascending) **118**

VI.9. Time series of 35 populations of *Coccus viridis* at 7 distinct locations in a 45 ha plot on an organic coffee farm in Chiapas, Mexico **119**

VI.10. Distributions of ant nests in a 45 ha plot, represented as 17 m diameter gray circles, along with X's marking the locations where at least one neighboring coffee bush contained green coffee scale insects in a) 2009 and b) 2010 **120**

VII.1. Snapshot of the stochastic, spatially-explicit model **128**

VII.2. Median fraction of sites with infected individuals, averaged over 100 realizations of the model, as a function of the rate of translocation from the environmental reservoir to the infectious class (v) **134**

VII.3. Median number of sites with infected individuals, averaged over 100 realizations of the model, as a function of the mortality rate of latent pathogens, ρ **135**

VII.4. Translocation rate (v), total number of infecteds, correlation length (Ψ), skewness, and variance for two regime shift scenarios **136**

VII.5. Mortality rate of latent spores (ρ), total number of infecteds, correlation length (Ψ), skewness, and variance for a scenario in which ρ was increased across a regime shift threshold **137**

List of Tables

II.1. Locations of positive *G. mellonella* and *C. viridis* baiting results **29**

V.1. Default parameter values **85**

VII.1. Model parameters and default values **131**

List of Appendices

A. Computer code for Chapter V: The evolution of imperfect prudence **146**

B. Computer code for Chapter VI: Self-organization of background habitat determines the nature of population spatial structure **223**

C. Computer code for Chapter VII: Detection of imminent, non-catastrophic regime shifts **237**

CHAPTER I

Introduction

Few things are more fundamental to human health and happiness than food. Not surprisingly, the future of agriculture, which is the primary means by which modern humans obtain food, generates vigorous debate. Unfortunately, this debate is frequently ineffective, often being waged in a polarized war between two camps that occupy opposite sides of an idealogical divide. On one side of this divide are those who view what has come to be called "conventional" agriculture as the best (the only) way to feed the growing human population. On the other side are those who point to the tremendous environmental and social costs of conventional agriculture and call for a return to an approach rooted in traditional knowledge, concern for the environment, and a rejection of the entire corporate, industrial agriculture paradigm. As is often the case with these kinds of debates, one wonders if there might be another, better alternative that transcends the artificial axis separating the naiveté of the technocrats and the romanticism of many who reject their vision. The goal of my research has been to contribute to the efforts of those who are attempting to create this alternative. Thankfully, there are hopeful signs that an alternative is possible. For anyone with an interest in creating a world in which humans can comfortably and happily exist in peace, this is good news.

Conventional agriculture is an approach that is fundamentally based in reductionism. Problems are identified, disassembled into their component parts, and then solved in a serial fashion. Your cotton is being attacked by bollworms? Plant Bt cotton. Now mirids have emerged as a problem? Search for a pesticide or genetic modification that will control this secondary pest. And so on. While the reductionist technique has been responsible for numerous scientific and technological advances in fields such as physics and engineering, the inherent complexity of ecosystems – even the drastically simplified systems typical of conventional agriculture – leads to a rippling of unintended consequences when such attitudes are applied. Although it could be argued that the Green Revolution program of Norman Borlaug, now championed in its modern form by Bill Gates and a bevy of agricultural input suppliers, led to massive increases in yields and saved millions of human lives, an equally compelling argument could be made that it was this program's singleminded focus on technical solutions (narrowly defined) that led to the litany of environmental and health crises associated with conventional agriculture. Cultural eutrophication, soil erosion, habitat fragmentation, farmers' toxic work environments, the dissipation of rural communities, and a profusion of other negative effects can be traced back to the reductionism of conventional agriculture.

Considering the incontrovertible shortcomings of conventional agricultural practices, it is not surprising that countermovements have arisen, most prominently under the umbrella of "organic agriculture." Unfortunately, this movement is too often defined by what should not be, and not often enough by what should be. Many synthetic fertilizers, pesticides, herbicides, and fungicides have been shown repeatedly to harm the environment and human health. So, clearly, they should be avoided if possible, and this is exactly what proponents of organic agriculture call for. However, it is not enough to advocate for the avoidance of certain substances and techniques without addressing the underlying issues that led to their use in the first place. Although many of the driving forces behind the adoption of agrochemical-intensive agriculture had very little to do with agricultural problems per se, agriculture does present various challenges that must be addressed. Replacing a facile technophilia with an equally facile nostalgia is insufficient.

So, what would a desirable alternative entail? The general, philosophical answer is clear. As Lewontin and Levins argue, agriculture was traditionally labor intensive, is currently capital intensive, and urgently needs to become knowledge intensive (Lewontin and Levins 2007). Many of the ills associated with conventional agriculture are a consequence of the drive by corporations to inject capital – large, expensive machinery, patented seeds, synthetic agrochemical inputs – into the agricultural system, regardless of any negative secondary consequences – provided that these deleterious side effects can be profitably externalized. A strategy based on profound, non-monetized agricultural knowledge, in contrast, would make positive outcomes, and not positive returns, the central focus of agriculture.

But what is really meant by "knowledge intensive"? After all, agriculture has always been a very complex, difficult endeavor, requiring incredible skill and knowledge to successfully cope with sundry interacting biological components and the vicissitudes of the environment. In the words of Adam Smith (1776):

> "No apprenticeship has ever been thought necessary to qualify for husbandry, the great trade of the country. After what are called the fine arts, and the liberal professions, however, there is perhaps no trade which requires so great a variety of knowledge and experience. The innumerable volumes which have been written upon it in all languages may satisfy us, that among the wisest and most learned nations, it has never been regarded as a matter very easily understood. And from those volumes we shall in vain attempt to collect that knowledge of its various and complicated operations which is commonly possessed even by the common farmer; how contemptuously soever the very contemptible authors of some of them may sometimes affect to speak of him."

If it is true that successful agriculture inherently requires a formidable amount of knowledge, and I think it is true, what is meant by a transition to "knowledge-intensive agriculture"? The answer lies in the type of knowledge that local practitioners (Smith's "common farmer") typically possess. Whereas farmers usually have knowledge that is specific, heuristic, phenomenological, and not generalizable to other systems and other contexts, modern science gives us an ability to make observations and perform analyses that are qualitatively different from what was previously possible, thereby generating a fundamentally different type of knowledge that can, ideally, complement more traditional types of knowledge. Modern science provides tools and techniques that allow us to detect and analyze patterns that are too subtle to be detected by practitioners; that span multiple systems; that unfold across large temporal and spatial scales; and that involve significant non-linearities and emergent properties.

The alternative approach, then, involves embracing and confronting the complexity of agricultural systems using the tools of modern science. It is a scientific approach, but it is not a reductionist science. It entails a deep understanding of the processes, interactions, and functions that characterize the components of agriculture, from genomes to biomes, in their actual context. This approach, most commonly referred to as agroecology, is the basis that underpins my research philosophy.

I conceive of agroecology as having three essential components. First, a thorough knowledge of the system, grounded in natural history, is crucial. Second, all of the appropriate tools from the science of ecology should be brought to bear on the system, including mathematical modeling, genetics, computer simulation, evolutionary ecology, biochemistry, etc. Finally, there should be a continuous, iterative dialog between theory and empiricism, with experiments and observations giving rise to theory which in turn suggests further experimentation and observation.

In my own research, I have worked to understand how *Lecanicillium lecanii*, a mycoparasitic and entomopathogenic fungus, provides the ecosystem service of pest control in an organic coffee farm in Mexico, using this system as both a focus in its own right and as a source of inspiration for more abstract, theoretical ideas about spatial ecology and evolution. In the chapters that follow, I will describe a portion of this research in an arc that will hopefully give the reader a clear sense of the progress that has been made towards understanding the role that this fungus plays in the complex coffee agroecosystem, but also, from a more philosophical perspective, demonstrate one example – albeit an imperfect one – of a single iteration of the dialog between empiricism and theory.

The arc through which my research unfolded, from basic empirical research into the natural history and biocontrol potential of *L. lecanii*, to theoretical models inspired by the *L. lecanii* system, and finally to a simulation model with potential implications for the monitoring and management of *L. lecanii* as a conservation biological control agent, derived partly from my philosophy of agroecological research and partly from necessity. Given the relative paucity of information in the literature about the ecology of *L. lecanii* under field conditions, there was a strong need to fill in certain gaps in our natural history knowledge, which gave me an opportunity to develop a foundational intuition about the study system while contributing to the basic literature about *L. lecanii*. Chapter II

(Jackson et al. In press), Chapter III (Jackson et al. 2009) and Chapter IV (Jackson et al. 2012) represent this phase. This field experience proved invaluable for sparking the theoretical ideas that I explore in Chapter V and Chapter VI (Jackson et al. In review). Finally, in an attempt to close the loop, I apply some of the combined empirical knowledge and theoretical insights that I gained during this process to a simulation model of the study system. This work is contained in Chapter VII.

Summary of dissertation

Study system

The study site upon which much of this work was focused is located on a 300 hectare organic coffee farm in Chiapas, the southernmost state of Mexico. This farm, Finca Irlanda, is the oldest certified organic coffee farm in the world, and has been studied intensively by researchers from the University of Michigan and the University of Toledo since the mid 1990s. A primary goal of this research has been to reveal the complex network of interactions present in this agroecosystem, and to understand how this system is affected by changes in management intensity.

At the core of the agroecosystem is a keystone mutualism between an arboreally-nesting ant, *Azteca instabilis*, and its hemipteran partner, the green coffee scale (*Coccus viridis*). The scale insects attach themselves to the coffee plants, typically along the midveins of the coffee leaves and on new, tender shoots. Using their piercing mouthparts, they access the phloem and suck the sugar-rich sap of the coffee plants. Since they are sedentary, the scale insects would be highly vulnerable to attack by predators and parasitoids if it were not for protection provided by the ants, which tend the scale insects in a classic ant-hemipteran mutualism. The ants build carton nests in shade trees that are planted throughout the farm and forage on the scale insects in the coffee plants below, providing protection from the scales' natural enemies in exchange for a carbohydrate-rich honeydew that the scales excrete.

Much of the research in this system has been carried out in a 45 ha study plot. A biannual census of shade trees and *A. instabilis* nests in this plot has been performed since 2004. Of approximately 7,000 to 11,000 shade trees in the plot (depending on the management intensity), only approximately 3-9% are inhabited by colonies of *A. instabilis*. However, despite the relative rarity of this mutualism, it has been shown to play a key role in maintaining autonomous pest control in the farm (Vandermeer et al. 2010a). Under the protection of the ants, the scale insect populations can reach very large numbers, on the order of a few thousand individuals per coffee plant. This provides a large resource that serves to maintain populations of various predators in the system, such as spiders and twig-nesting ants (Vandermeer et al. 2002, Perfecto and Vandermeer 2008b, Vandermeer et al. 2010a). *Azteca instabilis*, through its active patrolling and tending of the scale insects, also inadvertently protects the larvae of a predatory beetle, the coccinellid *Azya orbigera*, thereby providing both abundant food (the scale insects) and predator-free space for this important pest control agent (Liere and Perfecto 2008).

The ant-scale mutualism is also the principal determinant of the abundance and spatial distribution of *L. lecanii*. *Coccus viridis* are the primary hosts of *L. lecanii*, and they are particularly susceptible to epizootics of *L. lecanii* when their populations become very large and densely packed; this typically occurs only under the vigilance of *A. instabilis*. In sites subject to an *L. lecanii* epizootic, it is common for fungal mortality of scale insects to exceed 90% (Jackson et al. 2009). The great abundance of scale insects, coupled with a high prevalence of *L. lecanii,* results in *A. instabilis* nest sites being important sources of *L. lecanii* inocula. Given the ability of *L. lecanii* to attack both *C. viridis* and the coffee rust, *Hemileia vastatrix*, the spatial distribution of ant nests may thus play an important role in determining the extent to which these two important coffee pests are controlled.

Chapter II: Persistence of L. lecanii *propagules in the environment and dispersal of* L. lecanii *propagules*

One of the most prominent features of the study system is a pronounced wet-dry seasonality. During the wet season, which typically lasts approximately six months, from the end of May through November, there is rain virtually every afternoon and through the night. During the remainder of the year, rain is relatively infrequent. The activity of *L.*

lecanii is strongly influenced by this seasonality, for two reasons. First, the abundance of its host, *C. viridis*, is drastically reduced during the dry season (Jackson et al. In review). Second, the average relative humidity is below what is necessary for *L. lecanii* to thrive (Reddy and Bhat 1989). As a result, *L. lecanii* seems to be completely inactive during the dry season, although some remnants of infected cadavers of *C. viridis* from the previous wet season do persist throughout the dry season (personal observations).

This strong dependence of *L. lecanii* on the window of opportunity provided by the wet season presents two mysteries that were previously unaddressed in the literature. First, how and where does *L. lecanii* persist during the dry season? Second, via what mechanisms is *L. lecanii* dispersed? Knowledge of both of these is essential for understanding how *L. lecanii* is maintained in the system and how it is able to reestablish itself, spread, and successfully initiate epizootics every wet season. Dispersal is also a fundamental process that defines the ecology of fungal pathogens (Pell et al. 2010).

The existing literature did provide some clues regarding the first question. Previous studies had shown that propagules of various fungal entomopathogens can be recovered from the soil bank (Meyling and Eilenberg 2006, Tuininga et al. 2009), but information about this for *L. lecanii* in coffee agroecosystems was previously absent from the literature. Therefore, testing the hypothesis that the soil may provide an important environmental reservoir for *L. lecanii* was an early goal of my research program.

There were also a few obvious candidates for dispersal mechanisms. The ants, *A. instabilis*, seemed a very likely candidate given the high rates of activity and interaction with the hosts, *C. viridis*, inherent in their tending and foraging behavior. Rain splash also seemed to be a probable mechanism, particularly if the soil were acting as an important environmental reservoir. Since rain is very frequent in the wet season, rain splash could offer ample opportunities for propagules to successfully disperse from the soil. Additionally, high relative humidity is necessary for *L. lecanii* conidia to germinate (Reddy and Bhat 1989), so rain splash dispersal could offer a favorable microclimate as well as locomotion. Finally, wind, because it is a very common method for the dispersal of fungal spores (Aylor 1990), was seen as a likely possibility.

To test for the presence of *L. lecanii* propagules in the soil, we collected soil samples at the beginning of the wet season from locations both near and far from known epizootics in the previous year. In an attempt to define the dispersal kernel, the samples near the previous epizootics were taken along transects radiating out from the centers of the epizootics. The remainder of the samples were collected as far as possible from *A. instabilis* nests (and hence from previous *L. lecanii* epizootics). These soil samples were then baited with *Galleria mellonella* larvae and *C. viridis* on coffee leaves to detect the presence of *L. lecanii* propagules. The results of this assay demonstrated that viable propagules of *L. lecanii* can be recovered from the soil, including from locations far removed from recent epizootics.

Dispersal of *L. lecanii* via *A. instabilis* workers was tested using a combination of laboratory and field ant exclusion experiments. In both the laboratory and field experiments, coffee seedlings with populations of susceptible scale insects were assigned to two treatment groups: ant inclusion and ant exclusion. Individuals of *A. instabilis* that had been exposed to active *L. lecanii* infections were allowed to forage on the ant inclusion seedlings, but blocked from the ant exclusion seedlings. In the laboratory experiment, only scale insects on the ant inclusion seedlings became infected by *L. lecanii*, which demonstrates that *A. instabilis* can transport propagules of *L. lecanii*. In the field experiment, there was no significant difference in the prevalence of *L. lecanii* between the two treatments, suggesting that there are other important transport mechanisms besides *A. instabilis* in the field.

Dispersal by rain splash and wind were tested simultaneously in a laboratory experiment. Coffee seedlings with populations of uninfected scale insects were allocated to four treatment groups: control, wind, rain, and rain-wind. Each coffee seedling was placed in a tray of soil that had been inoculated with *L. lecanii* conidia and then exposed to a daily treatment of artificial rain splash, wind, both, or neither, depending on the treatment group. Both the rain and rain-wind groups experienced significant infections, whereas the wind-only and control groups were free of infected individuals.

Taken as a whole, the results of Chapter II suggest that *L. lecanii* does persist in the soil during the dry season; that it is widely distributed throughout the study site in this environmental reservoir; and that rain splash, *A. instabilis*, and potentially other dispersal mechanisms can spread *L. lecanii* propagules from the soil and throughout the coffee plants.

Chapter III: Spatial and temporal dynamics of L. lecanii *and the potential for* L. lecanii *to promote the self-organization of* A. instabilis *nests*

The *A. instabilis-C. viridis* mutualism has been shown to exert a disproportionate influence on the distribution and dynamics of other organisms in this coffee agroecosystem, i.e., it is a keystone mutualism (Vandermeer et al. 2010a). Accordingly, the spatial distribution of *A. instabilis* nests in the farm is of fundamental importance to the maintenance of biological control. Despite the shade trees in which the ants nest being distributed in a significantly uniform pattern, the ant nests are significantly clustered, with a mean/variance ratio of approximately 0.42 (Vandermeer et al. 2008). Explaining how this spatial distribution is generated has been a central goal of the Finca Irlanda group's research.

The most obvious explanation for the low-density, clustered distribution of *A. instabilis* nests would be some underlying environmental heterogeneity, e.g., variation in the size or species of shade tree. However, no such relationship is detectable in any of the census data. In the absence of an environmental explanation, the most likely explanation is that the spatial pattern emerges endogenously via local interactions. Using a simple cellular automata (CA) model, Vandermeer et al. (2008) demonstrated that such a self-organization process could in fact explain the generation of the observed spatial pattern. The CA model relied on two simple processes: satellite expansion, or budding, of ant colonies into unoccupied shade trees in neighboring sites, and density-dependent mortality. Satellite expansion of ant colonies is a well-known phenomenon. The density-dependent mortality factor, however, requires some explanation. There are a number of possible explanations, including a parasitic fly (*Pseudacteon* sp., Phoridae) that attacks *A.*

instabilis directly (Vandermeer et al. 2008) and a predatory beetle, *Azya orbigera,* that consumes the ants' mutualistic partner, *C. viridis* (Liere et al. In review).

An equally likely explanation is that *L. lecanii* contributes to the density-dependent mortality of *A. instabilis* colonies. Chapter III approaches this question using a combination of observational data and computer simulation. In addition, this chapter documents the spatial distribution and dynamics of *L. lecanii* across multiple spatial scales.

The observational data follow the history of shade tree occupancy by *A. instabilis* and the distribution of *L. lecanii* in two sites within the study plot. From these data, it appears that the ant colonies are migrating, or perhaps dying, in response to *L. lecanii* epizootics, which lends support to the hypothesis that *L. lecanii* is a significant contributor to colony turnover. The plausibility of this hypothesis is further bolstered by the computer simulation, which shows that the spatial distribution of ants can be generated by replacing the general density-dependent mortality factor in the original CA of Vandermeer et al. (2008) with an explicit model of *L. lecanii.* Regarding the spatial distribution and dynamics of *L. lecanii,* the plot-level censuses did not reveal a clear spatial pattern, but the finer scale surveys show distinct patterns in the spread of infection over time.

Although there remains some uncertainty about the true cause of the density-dependent mortality factor – and it is likely to be a combination of all of the candidates considered to date, as well as factors that are still unknown – this chapter shows that a plausible argument could be made in favor of *L. lecanii* as a principle cause. This raises the fascinating possibility that *L. lecanii* is simultaneously dependent upon and a determinant of the spatial distribution of its own hosts.

Chapter IV: Indirect biological control of the coffee leaf rust, Hemileia vastatrix*, by* L. lecanii

As a consequence of its potential to influence the spatial structure of the keystone mutualism between *A. instabilis* and *C. viridis, L. lecanii* is likely a key player in the maintenance of biological control in Finca Irlanda. However, there is also more direct evidence of its importance: there has been shown to be a significant relationship between *L. lecanii* and the prevalence of the coffee leaf rust, *Hemileia vastatrix* (Vandermeer et al. 2009). It is hard to overstate the importance of *H. vastatrix* as a disease of coffee. McCook (2006) describes the devastation it wrought in entire coffee growing regions, including the destruction of coffee in Sri Lanka and southern India.

Although a negative correlation between *L. lecanii* and *H. vastatrix* within a single season was reported previously by Vandermeer et al. (2009), the natural history discovered in the work described in Chapters II and III suggested that there might be a one-year lag between high abundances of *L. lecanii* and suppression of *H. vastatrix.* In this chapter, we test this hypothesis using multi-year surveys of *L. lecanii* and *H. vastatrix*. The data support the hypothesis, and enhance our understanding of the biological control services that *L. lecanii* provides in this system.

Chapter V: The evolution of imperfect prudence

In Chapter V, the first of the two most theoretical chapters in this dissertation, I explore the hypothesis that the spatial distribution of a locally-dispersing host might serve as an anti-pathogen phenotype, and that this group-level phenotype could arise via natural selection despite being counter to the short-term interests of individual hosts. This work was inspired, albeit loosely, by the observation that the low-density spatial distribution of *A. instabilis* nests in the coffee agroecosystem presents a much more challenging landscape for dispersal-limited pathogens than if the nests were much more densely distributed, and that this distribution is a consequence of the satellite expansion rate of the ants.

The basic concept underlying this chapter is that the spatial structure of a host will determine the ability of a locally-transmitted pathogen to spread through the population. For example, a host population distributed in small, isolated clusters will be resistant to the spread of a dispersal-limited pathogen; if the pathogen infects a cluster, it will only be able to exploit the small number of available susceptibles in that patch. At the other extreme, if the host population consists of a single well-connected network of hosts, the pathogen will spread throughout the entire host population (depending, of course, on the details of the infection process).

This scenario presents a contradiction for the host. On the one hand, competition for space will favor those hosts that reproduce as quickly as possible. On the other hand, a cluster of hosts that reproduces rapidly will tend to form large clusters that expand and coalesce with neighboring clusters, thereby forming a well-connected landscape that the pathogen can easily percolate through. The advantage of fast reproduction is conferred to individual hosts over the short term, while the advantage of restrained reproduction is a longer term, group-level benefit. Basic arguments against group selection suggest that the former will always dominate over the latter, but this balance can be reversed through the effects of spatial structure.

In this chapter, I develop a spatially-explicit, evolutionary, probabilistic cellular automata (CA) model to demonstrate that reproductive restraint of hosts, known in the literature as "prudence," can evolve in a viscous, spatially-structured host-pathogen system. This model shows that prudent hosts can indeed evolve, in theory. This phenomenon, which is a type of evolution of cooperation, prevents the host population from being extirpated by the pathogen. However, the degree of reproductive restraint that the hosts evolve to – the Evolutionarily Stable Strategy (ESS) – is not ideal in terms of maximizing the average size of the host population or decreasing the variability of the population. Most of the previous work on the evolution of cooperation has focused on the extent to which the performance of the cooperative behavior exceeds that of a purely selfish strategy. This emphasis on the performance of evolved cooperation relative to pure selfishness, while a natural choice in some ways, leads to a tendency to overstate the power of autogenous processes to generate desirable outcomes. As the results of this chapter show, evolution can lead to surprising levels of cooperation, but this cooperation may be suboptimal by some measures compared to what could be achieved by a more rational strategy that focuses on specific outcomes.

Chapter VI: Self-organization of background habitat determines the nature of population spatial structure

The theory considered in this chapter was also heavily influenced by the spatial distribution of *A. instabilis* nests in Finca Irlanda. By virtue of the self-organization process that generates this spatial distribution, the size distribution of the clusters of nests can be described by a power law. This implies that there are a large number of small clusters and a few very large clusters, which contrasts with the normal (Gaussian) distribution that we often expect to encounter in nature.

The implications of a power-law cluster-size distribution for organisms that use these clusters as habitat patches is explored using the concept of a metapopulation/ source-sink continuum. Depending on the slope of the power law that defines the cluster size distribution, the landscape will either be characterized by a large number of patches with relatively short distances between patches (the metapopulation end of the continuum); a small number of large patches with relatively long distances between patches (the source-sink end); or something in between.

Using a simple patch occupancy model, I show that populations inhabiting landscapes that fall in the intermediate range of this continuum may have the lowest rates of patch occupancy, and may be much more likely to go globally extinct. On the metapopulation end of the continuum, patches are in close enough proximity that migration between patches counteracts the local extinction of the organism in individual patches, resulting in a continuous dynamic of local extinction and subsequent rescue of individual patches. On the source-sink extreme of the continuum, there exists at least one patch that is large enough to sustain a population in perpetuity; this patch acts a source that continuously rescues the smaller neighboring patches, which have much higher extinction rates. At an intermediate power law slope, the patches are neither numerous enough, nor close enough together, nor large enough to sustain the population as either a metapopulation or a source-sink population.

The synthetic landscapes that were used to investigate the metapopulation/source-sink continuum were constructed by drawing patch sizes from power law distributions and then randomly placing these patches. This method captures the cluster size distribution of self-organized landscapes, but ideal self-organized landscapes are scale free, meaning that there is clustering at all spatial scales. The implications of this higher-level spatial structure for the resident organism was explored by comparing the patch occupancy rates on a randomly-constructed landscape to a truly self-organized landscape generated by the CA model of Vandermeer et al. (2008). The habitat patches in the CA landscapes were then randomly scattered to form a third type of landscape, termed "dispersed CA." This third category retained the self-organized cluster size distribution but not the higher-level clustering of patches. For a given amount of habitat, patch occupancy was consistently higher in the self-organized CA landscapes and lowest in the dispersed CA landscapes, suggesting that the higher-level spatial structure inherent to a self-organized landscape could promote persistence of an organism inhabiting this landscape.

To tie this theory back to a real system, we examined the patch occupancy dynamics of *C. viridis* in clusters of *A. instabilis* nests. These data suggest that the self-organizing attributes of the arboreal ants create the patch structure that in turn generates a source/sink dynamic for the green coffee scale insect.

Chapter VII: Detection of imminent, non-catastrophic regime shifts

From a management perspective, maintaining a thriving population of *L. lecanii* is almost certainly beneficial for coffee production. The monetary and health benefits of the ecosystem services provided by the biological control of *H. vastatrix* and *C. viridis* by *L. lecanii* would be difficult to quantify, but they are undoubtedly substantial. Therefore, it would be useful to be able to detect an imminent collapse of the *L. lecanii* population. The goal of predicting the onset of ecosystem collapse has gained substantial interest recently as part of the more general ambition to develop leading indicators of regime shifts in dynamic systems.

In this chapter, I use a spatially-explicit, continuous space, stochastic model of the *L. lecanii* system to ask 1) whether regime shifts are likely to occur in this system and 2) if the leading indicators proposed in the literature be used to detect imminent regime shifts in this system.

The simulation model used to test these questions is based on the host-pathogen reservoir model of Hochberg (1989), which includes an environmental reservoir for the pathogen. Pathogens in this reservoir are unable to infect the host, but have a much slower rate of mortality. Based on the field observations and laboratory experiments detailed in Chapters II and III, it is likely that the soil is an important environmental reservoir of *L. lecanii*. Therefore, the maintenance of *L. lecanii* is probably heavily dependent on the survival of latent spores of *L. lecanii* in the soil and the rate of translocation of propagules from the soil to susceptible scale insects on the coffee plants above.

To test this hypothesis, I ran sweeps of the two parameters in the model that control these rates (the latent spore survival rate and the translocation rate). Three regime shift scenarios leading to a drastic reduction or loss of *L. lecanii* from the system were observed. All of these regime shifts were of the non-catastrophic variety, in contrast to the catastrophic regime shifts that are typically considered in the literature. Catastrophic regime shifts are associated with fold bifurcations, which imply hysteresis and a discontinuous jump in the state of the system in response to a small change in a forcing parameter. Although the leading indicators in the literature were primarily developed using systems containing fold bifurcations, it is thought that they could also be used to predict non-catastrophic regime shifts such as the ones present in the *L. lecanii* model (Scheffer et al. 2009).

Two proposed leading indicators were applied to the infection data in an attempt to detect a signal of the impending regime shifts. The first, a method based on spatial autocorrelation, failed to exhibit any signal of the incipient regime shifts. This was not a complete surprise, as this method is known to perform poorly when there is only weak coupling via dispersal between sites, as is the case in the *L. lecanii* simulation model.

The second approach to predicting regime shifts relies on a combination of changes in the spatial variance and the spatial skewness. A peak in the spatial skewness combined with a continued increase in variance is thought to be an unambiguous signal of an impending regime shift. None of the regime shifts observed in the simulation model displayed such a clear signal, although there were significant, noticeable changes in the skewness and variance prior to the most rapid of the three regime shift scenarios.

On balance, it seems unlikely that the proposed leading indicators could be used as-is to predict these non-catastrophic regime shifts in the actual coffee agroecosystem using data with realistic spatial and temporal resolution. However, the relatively large changes in the spatial variance and skewness give some hope that these statistics could be used as general, albeit somewhat ambiguous, signals that the system is changing in a potentially deleterious manner.

CHAPTER II

Occurrence in the soil and dispersal of *Lecanicillium lecanii,* a fungal pathogen of the green coffee scale, *Coccus viridis,* and coffee rust, *Hemileia vastatrix*

Conservation biological control, based on management practices that promote the survival and effectiveness of natural enemies of potential pest species, has attracted considerable attention as an enabler of sustainable crop production (Barbosa 1998, Gurr et al. 2000, Bale et al. 2008, Cullen et al. 2008, Fiedler et al. 2008, Jackson et al. 2009). Fungi are promising candidates for conservation biological control programs, as they are known to attack a variety of pest organisms (Butt et al. 2001), including arthropods (Shah and Pell 2003, Cruz et al. 2006), plants (Hasan and Ayres 1990, Te Beest et al. 1992, Charudattan and Dinoor 2000, Sauerborn et al. 2007), and plant pathogens (Kiss 2003, Fravel 2005). However, effective conservation biological control using fungal pathogens will require a thorough knowledge of their ecology (Pell et al. 2010), which is still lacking, particularly in semi-natural habitats such as complex agroecosystems (Hesketh et al. 2010).

The fungal entomopathogen and mycoparasite *Lecanicillium lecanii* (Zimmerman) Zare and Gams is a promising candidate for use in conservation biological control in our study system – an organic, shade coffee agroecosystem in Chiapas, Mexico. *Lecanicillium lecanii* has been shown to be an important natural enemy of the green scale, *Coccus viridis* Green (Hemiptera: Coccidae) in coffee (Easwaramoorthy and Jayaraj 1978, Reddy and Bhat 1989, Uno 2007, Jackson et al. 2009). It also is known to attack the coffee rust, *Hemileia vastatrix* Berkeley and Broome (Shaw 1988, Eskes 1989, González et al. 1995, Vandermeer et al. 2009, Jackson et al. 2012), and therefore may be crucial for suppressing this potentially devastating coffee disease (McCook 2006, Suffert et al. 2009).

In addition to its direct, negative effects on potential coffee pests, *L. lecanii* may have an important influence on a keystone mutualism between an arboreal-nesting ant, *Azteca instabilis* F. Smith (Hymenoptera: Formicidae), and *C. viridis*. *Azteca instabilis* tends *C. viridis* in a typical ant-hemipteran mutualism, wherein the ants protect the scale insects, which are sedentary as adults, from predators and parasitoids. In exchange, the scales excrete a carbohydrate-rich honeydew that the ants consume. Recent studies have shown that this mutualism may play a key role in maintaining multiple natural pest control agents in this agroecosystem (Vandermeer et al. 2010a). Because the ants also inadvertently protect the larvae of the coccinellid scale predator *Azya orbigera* Mulsant (Coleoptera: Coccinellidae), the *A. instabilis*-*C. viridis* mutualism provides enemy-free space and high prey density for this important biological control agent (Liere and Perfecto 2008). This mutualism also contributes to the management of the coffee berry borer, *Hypothenemus hampei* Ferrari (Coleoptera: Scolytidae) through the deterrent effect of *A. instabilis* foragers (Perfecto and Vandermeer 2006).

Lecanicillium lecanii may strongly influence the location and abundance of *A. instabilis* colonies, and hence may determine the extent of the aforementioned biological control effects of the ant-hemipteran mutualism. In this system, *L. lecanii* often becomes a local epizootic, killing nearly all of the *C. viridis* on a single coffee plant or a small group of neighboring plants. Therefore, *L. lecanii* reduces the amount of carbohydrate food available to an ant colony, which may have an indirect negative effect on colony survival. The potential for *L. lecanii* to cause the ant nest density-dependent mortality of *A. instabilis* colonies — one of the fundamental processes underlying the spatial self organization that generates the low-density, clustered spatial distribution of ant nests in this farm — has recently been demonstrated through a combination of field studies and computer modeling (Jackson et al. 2009).

Although a substantial amount of research has been done on the systematics (Zare et al. 2000, Gams and Zare 2001, Sung et al. 2001, Zare and Gams 2001, Zare et al. 2001, Kouvelis et al. 2008) and production (Feng et al. 2000, Kamp and Bidochka 2002, Gao et al. 2007, Gao et al. 2009, Shi et al. 2009) of *L. lecanii*, much less is known about its basic ecology and natural history, including in the context of coffee agroecosystems.

In the current study, we investigated mechanisms contributing to the development of local epizootics of *L. lecanii*. Epizootics in this system are strongly influenced by the pronounced seasonality in this region, which is characterized by a wet season and a dry season. During the dry season, scale populations, and hence the prevalence of *L. lecanii*, are drastically reduced. *Lecanicillium lecanii* is re-established every wet season following the resurgence of the scale populations. Therefore, the initiation and progression of epizootics depend on one or more initial infection events following the onset of the wet season (primary dispersal) and the subsequent spread of infection from infected *C. viridis* individuals to susceptible individuals (secondary dispersal).

Three fundamental questions follow from the basic epizootiology of this system: 1) where do the propagules of *L. lecanii* persist during the dry season, 2) what are the mechanisms of primary dispersal, i.e., how are propagules initially dispersed onto the coffee plants and the scale insects during the wet season, and 3) what are the mechanisms of secondary dispersal, i.e., how is the fungus spread within and between coffee plants following an initial infection? In this study, we investigate a subset of the mechanisms that may be operative in this system. We hypothesize that the soil provides an environmental reservoir for *L. lecanii*, and that propagules are transmitted from the soil to susceptible scale populations via rain splash or wind dispersal. We also explore the possibility that *A. instabilis* itself is primarily responsible for the dispersal of *L. lecanii* conidia within and between coffee plants, in effect sowing the seeds of its own destruction.

Methods

The field study was performed in a 45 ha plot located at Finca Irlanda, an approximately 300 ha, organic coffee farm in the Soconusco region of Chiapas, Mexico (15° 11' N, 92° 20' W). The farm is a shade coffee plantation, with coffee plants growing beneath trees that have been planted in an approximately uniform distribution. The locations of every shade tree in the 45-hectare plot are known from biannual censuses;

the locations of *A. instabilis* colonies, which nest in the shade trees, are also recorded during each census. All experiments were performed during the months of July and August, during the wet season (typically early May through November), which is within the peak season for the growth and spread of *L. lecanii* (unpublished data).

Soil sample baiting

Two independent soil sample baiting methods were performed to detect the presence of viable propagules of *L. lecanii* in soil samples. The first employed larvae of the wax moth *Galleria mellonella* L. (Lepidoptera: Pyralidae), and is a standard method for detecting entomopathogenic fungi in soil (Zimmermann 1986). As an alternative, less time consuming, and potentially more sensitive method for detection of *L. lecanii*, we used populations of *C. viridis* on coffee leaves to detect the presence of *L. lecanii* propagules.

We obtained soil samples from a total of 40 locations: 10 locations far from *A. instabilis* nests, and therefore far from where epizootics of *L. lecanii* had occurred the previous year; 15 locations near the center of a previous epizootic, site A; and 15 locations near the center of another epizootic, site B (sites and locations indicated in Figures II.1 and II.2). The first 10 locations were chosen to determine the potential for *L. lecanii* propagules to persist in the soil even without a recent influx of propagules from a nearby epizootic. The other 30 locations were chosen to determine if the prevalence of propagules in the soil decreases with distance from the center of recent epizootics.

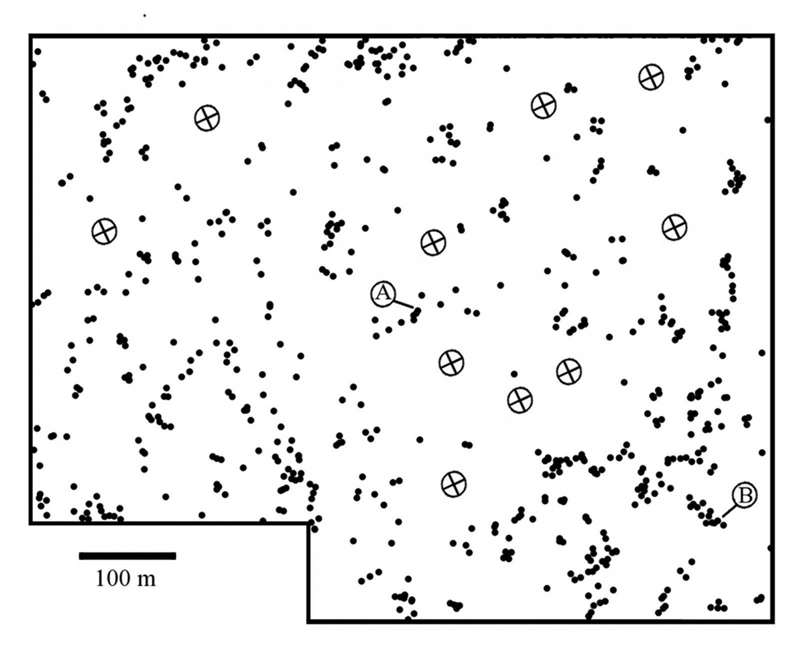


Figure II.1. Location of *A. instabilis* ant nests (solid circles) in 45 ha plot. Soil sample locations far from *A. instabilis* nests, and therefore far from recent epizootics of *L. lecanii* (circles with crosses); Site A; and Site B.

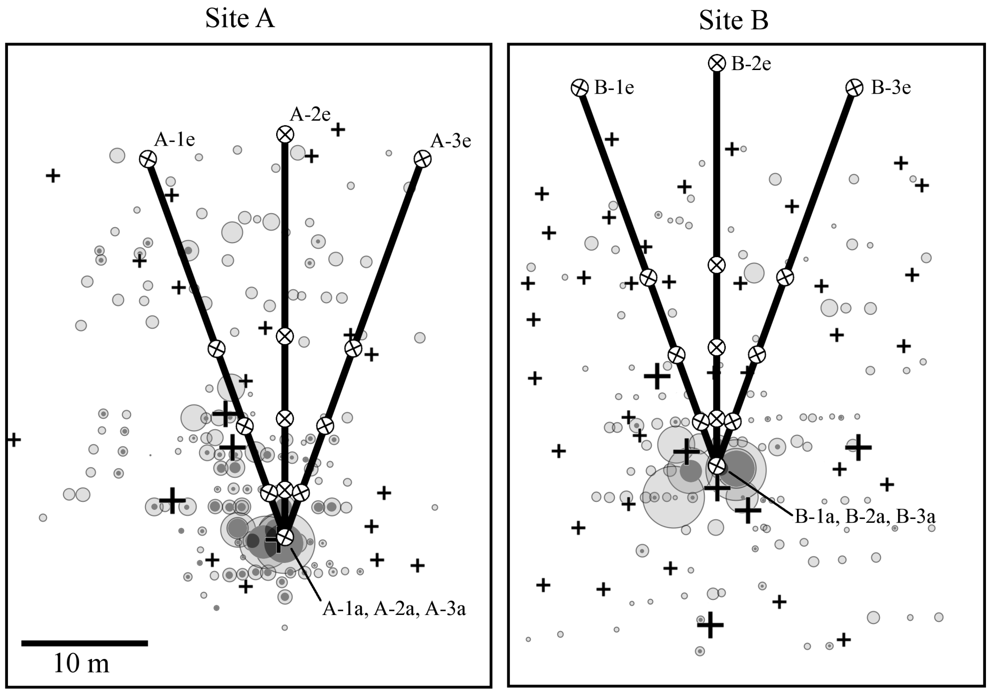


Figure II.2. Locations of soil samples on transects leading away from foci of two *L. lecanii* epizootics. Small crosses indicate locations of shade trees. Large crosses indicate shade trees occupied by *A. instabilis* colonies. Light gray circles are proportional to the number of healthy *C. viridis* on individual coffee plants in the previous year, and dark gray circles are proportional to the number of *C. viridis* infected with *L. lecanii*. Circles with crosses show the locations of soil samples. Survey data are adapted from Jackson et al. (2009)

Soil samples were taken to a depth of 10 cm using a 2 cm-diameter, manual core sampler. The litter layer, when present, was included in the samples. At each location, we took 10 samples from a 40 cm X 80 cm rectangular area. The core sampler was thoroughly cleaned and rinsed with ethyl alcohol between samples. The 10 samples from each location were combined in separate polyethylene bags. After collection, the soil was spread on paper in a sterile environment and allowed to dry for 24 hours at ambient temperature in the dark. We then homogenized the soil by rolling it and passing it through a sieve (Niblack and Hussey 1987).

After the soil was allowed to dry, we placed 90 cc (approximately 80 g) of soil from each sample in a plastic container and moistened the soil evenly with 20 mL of distilled water. We prepared laboratory-reared *G. mellonella* larvae by placing them in 56 °C water for 7 seconds in order to reduce their activity and discourage them from producing silk webbing in the soil. Each sample was baited with 10 larvae. The plastic containers were then sealed with perforated lids and incubated at room temperature (26-28 °C) for 2 weeks. The larvae were inspected daily, and dead larvae were removed and placed in humidity chambers for later evaluation. In lieu of the usual step of inverting the containers to ensure that the larvae penetrate the soil evenly, the soil was thoroughly mixed during the daily inspection process. At the end of the incubation period, larvae exhibiting fungal growth were inspected using a stereomicroscope at 400x magnification to identify the fungi morphologically.

For the second soil sample baiting, we collected branches with uninfected *C. viridis* populations from three adjacent coffee plants located within the 45 ha plot; there were no scale insects with any visible signs of infection by *L. lecanii* on any of these three plants or the adjacent coffee plants. The average number of large (greater than approximately 0.7 mm in width) scales was 35.8 per leaf (s.d. = 14.3). We then divided the branches into sections of three leaves, selecting one section at random for each soil sample. We suspended 10 g of soil from each sample in 10 mL of distilled water and, using a small paintbrush to apply the suspension, inoculated the scale insects on a leaf. This procedure was immediately replicated for the other two leaves assigned to the soil sample, i.e., a separate suspension was prepared for each leaf. As a control, 10 groups of leaves with scale insects (30 leaves) were treated with distilled water. The leaves were placed in humidity chambers at 100% relative humidity and incubated for 2 weeks. Fungal infections were identified morphologically using a stereomicroscope (400x magnification).

Rain splash and wind dispersal

The potential for rain splash and wind dispersal of conidia from the soil was tested using coffee seedlings containing susceptible scale insect populations placed in four treatments: rain, rain-wind, wind, and control. The average number of scale insects per seedling was 112.6 (s.d. = 92.7). For this and all other experiments, we counted only adult scales larger than approximately 0.7 mm in width. The seedlings used in this and all other experiments were obtained from the farm's nursery, where they were planted and reared in 10 x 20 cm black polyethylene bags. Four seedlings were randomly assigned to each treatment, for a total of 16 seedlings. The seedlings were placed in the four corners of white 60 × 60 × 60 cm insect rearing tents (BugDorm-2, MegaView Science Co., Ltd., Taiwan). A plastic tray (26.5 × 17.5 × 6.0 cm) with soil that had been inoculated with an aqueous suspension of *L. lecanii* conidia was placed in the center of each group of four seedlings. Approximately 0.45 mL of suspension was added per cubic centimeter of soil at the start of the experiment. The conidial concentration, approximately 1.9 X 105 conidia/mL, was determined using a hemacytometer.

The inoculum was an aqueous suspension of *L. lecanii* conidia cultured from spores and hyphae acquired from an infected *C. viridis* obtained within the 45 ha plot. The *L. lecanii* isolate originated in a single *C. viridis* individual from a population affected by a severe epizootic, with nearly 100% prevalence of *L. lecanii*, and therefore was likely of average, or possibly above average (for our study site), pathogenicity to *C. viridis*. Following isolation of *L. lecanii* from the scale insect, conidia were mass-produced via solid-state fermentation using cooked rice as a substrate. We then suspended the resultant conidia in water and added Tween 80 surfactant to the suspension.

Seedlings in the rain and rain-wind treatments were removed from their tents once every 24 hours during the two week experiment to be exposed to artificial rain splash. During the rain treatment, the seedlings were placed around their respective plastic trays, with one seedling on each edge. Two minutes of simulated rain were created using a 2.5-gallon plastic bladder connected to a hose with a spray nozzle and filled with room-temperature tap water. Prior to the experiment, the volume and intensity of the simulated rain was compared and adjusted to qualitatively match rainfall typical of the study site. The simulated rain was focused on the center of the plastic tray such that the rain impinged primarily on the soil but also fell on the seedlings. After one minute, the plants were moved in a clockwise manner to an adjacent edge of the tray to account for the rectangular shape of the tray, i.e., so that each plant was exposed to equivalent rain splash intensity. The bottoms of the plastic trays were perforated to allow the water from the simulated rain to drain. To prevent any potential loss of conidia from the inoculated soil, we placed the rain-wind treatment tray underneath the rain treatment tray while the simulated rainfall was performed on the rain treatment, and vice versa. To balance the net washout of conidia, we alternated the order of the simulated rain treatment, i.e., every other day the same treatment was rained on first. The plants from all of the treatments were taken out of their cages and left outside while the simulated rain was being applied so that each plant spent the same amount of time outside of the tents. The seedlings were always returned to the same corners of the tents in order to avoid cross contamination between plants.

After all plants were returned to their tents, the wind and rain-wind treatments were exposed to simulated wind that was created by small electric fans (one fan per tent). The fans were run for 30 minutes at maximum speed, which is qualitatively similar to the typical maximum daily wind speed at the study site. The orientation of each fan was changed daily by rotating the fan 90 degrees clockwise; this was done to vary the direction of the airflow impinging on the plants.

Seedlings were inspected daily for scale individuals exhibiting the white halo of mycelia characteristic of infection by *L. lecanii*. A final count of infected and healthy *C. viridis* adults was performed after two weeks, at the conclusion of the experiment.

Ant exclusion

Two ant exclusion experiments were performed: a laboratory experiment, in which most potential conidia dispersal mechanisms were eliminated, and a field experiment, which included the full complement of potential conidia dispersal pathways (e.g., wind, rain splash, arthropods, and other animals).

For the laboratory ant exclusion experiment, eighteen small coffee seedlings inhabited by populations of *C. viridis* were obtained from the farm's nursery. The *C. viridis* populations on six of the seedlings showed signs of being infected with *L. lecanii*, with some of the scales surrounded by the white halo of mycelia indicative of *L. lecanii* infection. The scales on the other 12 seedlings showed no signs of infection. The average number of scales on these 12 seedlings was 99.8 per plant (s.d. = 38.5). The six seedlings harboring infected scales were set aside as sources of fungal conidia, and the 12 infection-free seedlings were designated for use in the treatments.

For each replicate, three plastic flower pots were attached in a line to a wooden board, with approximately five cm separating the pots. An infected seedling was planted in the center pot and then covered with an enclosure of clear plastic in order to prevent transmission of fungal conidia by air currents or flying insects. The top of the plastic enclosure was rolled up and sealed with metal clips to allow for periodic access to the seedling. A small opening covered with mosquito netting was included on one side at the top of the enclosure as a vent to prevent condensation from accumulating inside. Two fungus-free seedlings were then planted in the two adjacent pots. These seedlings were also covered with plastic enclosures, with the vents on both of these enclosures facing in the opposite direction from the infected seedling's vent. To allow the passage of ants from the center seedling to the ant inclusion treatment seedling, an approximately 2.5 cm-diameter clear plastic tube penetrating the enclosures was routed between the two seedlings. An identical tube was routed between the center seedling and the ant exclusion treatment, with the exception that one end of the tube was covered with mosquito netting to prevent ants from entering the tube. Hot glue was used to thoroughly seal the enclosures to ensure that ants could not escape and that other arthropods could not enter the enclosures. Six identical replicates were constructed.

At the beginning of the experiment, a single coffee leaf with scales heavily infected by *L. lecanii* was tied to the base of each infected seedling in order to increase the amount of conidia available for the ants to spread. The coffee leaves were collected from the site of a severe epizootic, with nearly 100% prevalence of *L. lecanii*, and therefore it is probable that the pathogenicity of the strain(s) of *L. lecanii* used as inoculum were of at least average (for our study site) pathogenicity to *C. viridis*. Approximately 150 *A. instabilis* ants were then placed in the enclosures with the seedlings and leaves harboring infected scales. After three weeks, the scales on the seedlings were counted and the number of scales showing signs of infection by *L. lecanii* was noted.

For the field ant exclusion experiment, twenty coffee seedlings inhabited by *C. viridis* populations, with a mean of 202.1 scales per plant (s.d. = 136.9), were placed in plastic pots and arranged in a circle around a shade tree containing an active *A. instabilis* colony. Since the purpose of the *A. instabilis* colony was simply to provide a source of ant foragers, all of the field ant exclusion replicates were located near a single, vigorous colony. The plants were placed two meters from the base of the shade tree, with 20 cm separating each pot. To encourage discovery of the seedlings by the ants, bridges of plastic twine were tied between the shade tree and the bases of the seedlings.

The seedlings were assigned in an alternating manner to either the ant exclusion treatment or the ant inclusion treatment, i.e., 10 seedlings were assigned to each treatment type. A piece of a coffee leaf covered with approximately 10 *C. viridis* that had been infected by *L. lecanii*, obtained from a site subject to a severe epizootic, was tied around the stem at the base of each coffee seedling to provide a source of conidia. All inoculum was again sourced from the location of a severe epizootic. An approximately eight cm wide strip of flagging tape was wrapped around the base of the seedlings, just beneath the infected coffee leaf; Tanglefoot® (Tanglefoot Co., Grand Rapids, Michigan, USA) was applied to the flagging tape on the ant exclusion seedlings. Surrounding vegetation was cleared to ensure that no bridges were available whereby the ants could access the seedlings from neighboring vegetation. All ants were removed from the ant exclusion seedlings by hand, using a small paintbrush, following the application of Tanglefoot®.

The seedlings were left in the field from 15 July to 4 August. They were inspected daily to ensure that no ants had gained access to the ant exclusion seedlings. To encourage a more typical number of ants to discover and tend the scale insects on the ant inclusion seedlings, small pieces of tuna were placed at the bottom of all seedlings on 18 July. The leaves with fungus that had been tied to the base of the seedlings were beginning to show signs of decomposition by 27 July, so a single coffee berry with approximately five fungus-infected scales from the location of a major epizootic was attached with a wire-tie to the base of each seedling to provide a fresh source of inoculum. Following the experiment, prevalence of *L. lecanii* was assessed.

Statistical analyses were performed following the resampling, or bootstrapping with permutation, method described in Liere and Perfecto (2008). In this method, synthetic treatment and control populations are created by resampling without replacement from the original observations, and the difference in the relevant statistical measure (e.g., the mean number of infections) between the two synthetic populations is compared to the observed difference. This procedure is repeated many times, and a p-value is calculated based on the proportion of repeats for which the difference between synthetic populations is as extreme or more extreme than the difference between the actual populations. Data were resampled 10,000 times. The rain splash and wind dispersal data were resampled using a custom script in Matlab, while the ant exclusion data were analyzed using the Resampling Stats Excel add-in version 3.2 (Resampling 2006).

Results

Soil sample baiting

Of the 400 larvae used in the *G. mellonella* larvae baiting (10 larvae/sample X 40 samples), 202 were infected by one or more entomopathogenic fungi. Of these, six were infected with *L. lecanii*, based on morphological identification using the characteristic conidia and diagnostic phialides (Zare and Gams 2001). Two of the *L. lecanii*-infected larvae were from samples taken from the points nearest to the focus of the *L. lecanii* epizootic at Site B (B-1a and B-2a); one was from a sample taken at one of the fourth-furthest transect points at Site A (A-2d); and the other three larvae were from samples taken far from *A. instabilis* nests (Table 1). In no case was there more than one larva per soil sample infected by *L. lecanii*.

The *C. viridis* baiting method yielded eight positive identifications of *L. lecanii* from the 40 soil samples, at the following locations: the fourth-furthest point at Site A (A-1d); all five distances at Site B (B-2a through B-2e); and two locations far removed from *A. instabilis* nests (Table 1). All of the positive samples from Site B were taken from the middle transect. All three replicates from the third-furthest point at Site B were positive, and two of the replicates from the fifth-furthest point were positive, meaning that a total of 11 of the 120 assays (3 replicates per sample X 40 soil samples) were positive. None of the scale insects on the control leaves were infected.

Of the 14 sampling locations that tested positive for the presence of *L. lecanii*, only one location – a point nearest to the center of the epizootic at Site A – tested positive using both methods. That is, a total of 13 of the 40 sampling locations tested positive for *L. lecanii* using one or the other of the two methods.

Location	*G. mellonella*	*C. viridis*
A-1a		
A-1b		
A-1c		
A-1d		X
A-1e		
A-2a		
A-2b		
A-2c		
A-2d	X	
A-2e		
A-3a		
A-3b		
A-3c		
A-3d		
A-3e		
B-1a	X	
B-1b		
B-1c		
B-1d		
B-1e		
B-2a	X	X
B-2b		X
B-2c		X
B-2d		X
B-2e		X
B-3a		
B-3b		
B-3c		
B-3d		
B-3e		
Far from nests	3	2

Table II.1. Locations of positive *G. mellonella* and *C. viridis* baiting results. See Figures II.1 and II.2 for location information.

Rain splash and wind dispersal

At the conclusion of the experiment, three of the four rain treatment seedlings had scales infected by *L. lecanii*; one of the rain-wind treatment seedlings had infected scales; and none of the wind or control treatment seedlings had infected scales. The mean percentages of scales infected with *L. lecanii* were 0.0%, 3.2 ± 2.6% (SE), 0.0%, and 0.3 ± 0.3% for the control, rain, wind, and rain-wind treatments, respectively. The difference in the number of scales infected in the rain treatments compared to the control, wind, and rain-wind treatments was significantly greater than the random expectation ($P < 0.0001$, $P < 0.0001$, and $P < 0.01$, respectively). The difference in the number of scales infected in the rain-wind and the control treatments, however, was not greater than expected by chance ($P = 0.27$). There was no significant linear relationship between the number of scales per plant and the rate of infection ($P = 0.28$).

Ant exclusion

In the laboratory ant exclusion experiment, scales on five of the six ant inclusion seedlings exhibited the white mycelial mat characteristic of *L. lecanii* infection, while only one scale on the ant exclusion seedlings showed signs of possibly being infected. On the ant inclusion seedlings with *L. lecanii*-infected scales, the percentage of infected scales ranged from 1.8% to 12.5%. The mean percentage of scales killed by *L. lecanii* was significantly greater for the ant inclusion seedlings than for the ant exclusion seedlings (0.1 ± 0.2% [SE] without ants, 4.3 ± 1.8% with ants, $P < 0.01$).

In the field ant exclusion experiment, the percentage of infected scales on the ant exclusion seedlings ranged from 3.0% to 46.5%, while on the ant inclusion seedlings the range was 3.6% to 42.2%. The mean percentages of scales killed by *L. lecanii* with or without ants were not significantly different (17.4 ± 4.6% [SE] without ants, 18.2 ± 4.3% with ants, $P = 0.44$).

There was no significant linear relationship between the average number of scales per plant and the rate of infection in either the lab experiment ($P = 0.80$) or the field experiment ($P = 0.84$)

Discussion

These results suggest the following scenario for the development of epizootics in this coffee agroecosystem. During the dry season, the populations of *C. viridis* are markedly smaller than during the wet season. Therefore, individual populations of scale insects are below the epizootic threshold density, and *L. lecanii* persists primarily in the environmental reservoir provided by the soil. As the scale populations increase following the onset of the wet season, they are exposed to *L. lecanii* propagules splashed up from the soil, which provide the inocula necessary to initiate epizootics. Further development of an epizootic almost certainly requires transmission of conidia between individuals in the scale population, which can be effected by *A. instabilis* and other, as yet unknown, vectors. These processes lead to a rapid increase in the prevalence of *L. lecanii* shortly after the start of the wet season, which has been observed in our study site (unpublished data) and others (Reimer and Beardsley 1992).

The baiting results demonstrate that viable propagules of *L. lecanii* can be found in locations that are as far removed as possible in this system (up to approximately 50 m) from recent *L. lecanii* epizootics. This suggests that either 1) *L. lecanii* can persist in the soil for multiple seasons or 2) *L. lecanii* is not dispersal limited in this system.

The fact that the soil can act as an environmental reservoir for *L. lecanii* in this system has important implications for the epizootiology of this fungus. The temporal dynamics of diseases have been shown to be strongly influenced by the presence of a pathogen reservoir: Hochberg (1989) showed that intermediate levels of translocation of a pathogen from a reservoir result in damped oscillations and relative stability of an otherwise oscillatory system. While the results of the rain splash experiment demonstrate that translocation of *L. lecanii* from the soil is possible, further study will be necessary to determine the actual level of translocation under field conditions. In particular, the concentration of *L. lecanii* in the soil in the field, and how this concentration varies spatially and temporally, are unknowns that could significantly affect the realized translocation rate.

The spatial dynamics of this system will also be strongly affected by the apparent ubiquity of infectious propagules in the soil. Transmission of *L. lecanii* upwards from infected soil widely distributed within the farm would likely result in much more rapid and widespread infection at the onset of the wet season compared to transmission from multiple point sources, e.g., from isolated cadavers left over from epizootics that occurred in the previous wet season. The potential for *C. viridis* to escape foci of previous epizootics by dispersing is also likely to be greatly reduced by the widespread occurrence of *L. lecanii* propagules in the soil.

The results of the two soil sample baitings are also interesting from a methodological perspective. In none of the samples were multiple replicates of the *G.*

mellonella larvae infected by *L. lecanii*, which suggests that there is a large element of chance with this method, i.e., the presence of infectious material in a sample will not necessarily result in infection of the larvae. This may be due to the larvae failing to come into contact with the infectious material, possibly due to a very low density of infectious material in the sample; resistance of the larvae to infection; mortality due to other causes that occurs before the larvae can become infected; or *L. lecanii* being outcompeted within a single larva by another entomopathogenic fungus. Negative results of this method, therefore, should be treated with caution. Results from the *C. viridis* baiting were similarly subject to chance. However, the issue of other entomopathogenic fungi outcompeting *L. lecanii* was not a concern with this method, as *C. viridis* did not become infected by any fungi other than *L. lecanii*, perhaps because it is not susceptible to the broad range of entomopathogenic fungi that infected the *G. mellonella* larvae.

Another consideration raised by our study is that using a bait species known to be a target of the entomopathogen of interest may be a more powerful detection strategy than using a non-target bait species. Although there was not a significant difference in the total number of positive samples obtained using the two bait species employed in our study, Klingen et al. (2002) report that using a pathogen-specific host species as a bait yielded significantly more positives than using *G. mellonella*. Therefore, when considering the apparent rarity of *L. lecanii* in our study system (15% and 20% positive samples with the *G. mellonella* and *C. viridis* methods, respectively) and other agroecosystems [e.g., 0.4-2.6% in a study by Meyling & Eilenberg (2007)], the potential influence of the sensitivity of the bait species should be kept in mind. An understanding of the role of the soil as an environmental reservoir for fungal entomopathogens in a given system would likely benefit from a combination of standard baiting methods (e.g., the *G. mellonella* bait method), baiting methods that are specifically tailored to the system (e.g., the *C. viridis* method used here), and molecular approaches (Enkerli and Widmer 2010), including those that allow for quantitative assessments. A quantitative assessment of the abundance of *L. lecanii* propagules may reveal a dispersal kernel dependent on the distance from recent epizootics, which we were unable to detect using our experimental methods.

In the rain splash and wind dispersal experiment, the lower infection rate in the rain-wind treatment relative to the rain treatment suggests that there may be an important interaction between rain splash and wind in this agroecosystem. Wind increases the rate of evaporation of rain splash from the surface of the scale insects, and therefore may decrease infection rates due to desiccation of conidia. Airflow may also remove rain splash-dispersed conidia from the scale insects before they are able to germinate. This potential interplay between rain splash and wind may have important implications for management of shade levels in coffee agroecosystems. As shade level increases, the intensity of rain splash and wind will both decrease, which may serve to simultaneously decrease dispersal of conidia from the soil while increasing the probability of success of the conidia that are dispersed. Therefore, prevalence of *L. lecanii* may be maximized at an intermediate shade level. To our knowledge, although the effect of shade on prevalence following artificial inoculation has been studied (Easwaramoorthy and Jayaraj 1977), the effects of shade level on the occurrence of natural epizootics of *L. lecanii* has not been investigated.

Rain splash dispersal of fungal entomopathogens has not been studied extensively, but has been previously noted by other researchers, including dispersal of Beauveria bassiana from the soil onto leaves of corn plants (Bruck and Lewis 2002) and of the mealybug pathogen *Hirsutella cryptosclerotium* (Fernandez-Garcia and Fitt 1993). Fitt et al. (1989) identify characteristics of fungi that tend to be rain splash dispersed, such as mucilaginous conidia; Heale (1988) notes that *Verticillium lecanii* conidia are produced in mucilaginous heads and dispersed by water splash or insects. There is also a substantial literature on rain splash dispersal of fungal pathogens of plants (Madden 1997, Geagea et al. 2000, Ahimera et al. 2004, Huber et al. 2006).

The results from the laboratory ant exclusion experiment suggest that *A. instabilis* is capable of transporting conidia of *L. lecanii*, and hence may play a role in dispersing the fungus throughout populations of *C. viridis*. This would seem to indicate that transmission of conidia via ants between branches in a coffee plant, or perhaps between coffee plants themselves, is possible. However, the proportion of scale insects infected by the fungus was very low in the laboratory experiment relative to the field experiment, so the ants appear to be relatively poor dispersal agents. It is important to consider, however, that differences in pathogenicity of the inocula used in the two experiments could be partially responsible for the disparity in infection rates.

Our results are consistent with a previous study that showed that the common black ant, *Lasius niger* (Hymenoptera: Formicidae), is capable of retaining conidia of an entomopathogenic fungus previously grouped in the *V. lecanii* species complex (Sitch and Jackson 1997, Bird et al. 2004) and that by transporting conidia to tended aphids, it can serve as a vector. Bird et al. (2004) demonstrated that *L. niger* workers artificially inoculated with *Lecanicillium longisporum* (Zimmerman) Zare and Gams [*Verticillium lecanii* (Zimmerman) Viégas] conidia could infect aphid populations, causing significant mortality under laboratory, semi-field, and field conditions. Aphid mortality due to *L. longisporum* was greatest under laboratory conditions and least under field conditions, which contrasts with our results. However, relative mortality under lab and field conditions depends heavily on the specific attributes of the methodologies and the lab and field environments (e.g., microclimate, presence of other potential vectors, etc.), so it is not possible to draw any general conclusions from this discrepancy.

The coffee seedlings used in the field ant exclusion experiment are most representative of smaller coffee plants and the lowest branches of larger plants. Based on our results, it appears that other dispersal mechanisms besides *A. instabilis*-vectored dispersal from one scale insect to another dominate in these locations. There are a number of dispersal agents that could disperse *L. lecanii* conidia, such as rain splash from the soil or between *C. viridis* individuals, or any of the sundry flying and crawling arthropods that visit the coffee plants.

Roditakis et al. (2000) showed that aphids are capable of transporting conidia of *L. lecanii*, so it is likely that other arthropods in this system are also capable of spreading conidia of *L. lecanii*. Sitch and Jackson (1997) demonstrated that resistant arthropods from a variety of orders are capable of retaining *Verticillium lecanii* conidia, albeit at lower rates than target aphid species. A particularly intriguing possibility is that the predatory beetle *A. orbigera*, a key predator of scale insects in this system that is positively associated with the presence of the *A. instabilis*-*C. viridis* mutualism (Liere and Perfecto 2008), may be a primary vector of *L. lecanii*. Such a phenomenon would not be unprecedented, as the coccinellid aphid predator *Coccinella septempunctata* (Coleoptera: Coccinellidae) has been shown to be a potential vector of an entomopathogenic fungus when artificially inoculated, causing significant aphid mortality due to fungal infection (Roy et al. 2001). Whatever the dominant dispersal agents are, previous work showing a signal of dispersal-limited spread between coffee plants (Jackson et al. 2009) suggests that these mechanisms are primarily transmitting the fungus between adjacent plants.

It is important to note that *A. instabilis* very likely plays a central role in the dynamics of *L. lecanii* infection of *C. viridis* even if it is not primarily responsible for dispersal of conidia. There appears to be a minimum abundance and density of *C. viridis* that are necessary for an outbreak of *L. lecanii* to occur, i.e., an epizootic threshold density (unpublished data). When such an outbreak occurs, the fungus kills the vast majority of scales on entire coffee plants. Without *A. instabilis* tending the scales and providing protection from predators and parasitoids, the scale population is unlikely to reach a sufficient size for a fungal outbreak to occur (Reimer et al. 1993, Uno 2007). Therefore, *A. instabilis* is likely an important factor in determining the local prevalence of *L. lecanii*.

Our results suggest that a complete understanding of the epizootiology of *L. lecanii* will require knowledge of multiple phases of transmission and persistence: persistence in the soil, particularly during the dry season; translocation of propagules from the soil via rain splash; secondary dispersal between coffee plants, branches, and *C. viridis* individuals; and subsequent replenishment of the environmental reservoir in the soil. The spatial extent, phenology, and dynamics of epizootics in this system are all influenced by the details of these processes.

Understanding the development of *L. lecanii* epizootics in this system is crucial because of the role *L. lecanii* may play in the biological control of important coffee pests: directly, by attacking *C. viridis* and the coffee rust *H. vastatrix*, and indirectly, via its potential to influence the spatial distribution of the *A. instabilis*-*C. viridis* keystone mutualism. Consequently, enhanced understanding of the mechanisms controlling the occurrence of *L. lecanii* epizootics in this system, and appropriate management practices informed by this knowledge (e.g., coffee plant height and planting density, shade levels, etc.), appear to have an enormous potential benefit in terms of improved conservation biological control in this and other similar coffee agroecosystems.

CHAPTER III

Spatial and temporal dynamics of a fungal pathogen promotes pattern formation in a tropical agroecosystem

It is familiar knowledge in ecology first, that patterns in space are both striking and important and second, that complex interacting networks surround every population of every organism. It is only recently that these two issues have come together in a mutually reflective way, leading to a fundamental question of causality: does the spatial pattern determine the details of the interacting network or does the spatial pattern result from that network? While the existence of spatial patterns in extended landscapes has long been appreciated, it has frequently been assumed that they emerge from underlying habitat variables, which implicitly takes the pattern as an independent variable which determines the nature of population interactions of the species living in the landscape. It is only recently that a great deal of theoretical work has been devoted to demonstrating the possibility of the opposite causality, that the pattern itself is caused by the population interactions (Rohani et al. 1997, Bascompte and Solé 1998, Pascual et al. 2002, Rietkerk et al. 2002, Scanlon et al. 2007). Here we contribute to this debate with the suggestion that a fungal disease attacking the food of an ant ultimately causes the distributional patterns of the ant nests.

Recent studies have shown that the spatial distribution of the nests of an arboreal ant *Azteca instabilis* (Formica, Hymenoptera) in a coffee agroecosystem may emerge through self-organization (Perfecto and Vandermeer 2008b, Vandermeer et al. 2008). The ant *A. instabilis* builds nests in shade trees within the system and tends a species of scale insect (*Coccus viridis*, Coccidae, Hemiptera), which resides in the coffee bushes, in a classic ant/Hemipteran mutualistic association. The proposed self-organization process has been studied with the aid of a cellular automata model which involves local effects for both expansion and density-dependent mortality of the ant colonies (Vandermeer et al.

2008). The local expansion process is obvious from casual field observations, arising when ant colonies establish satellite colonies in neighboring trees. However, the cause of the density-dependent mortality is less evident. It has been attributed to the attack of a dipteran parasitoid (*Pseudacteon* sp., Phoridae), although evidence for this mechanism is only correlative (Vandermeer et al. 2008). Indeed there are a variety of other processes that could be responsible for the proposed density-dependent mortality. The most evident alternatives include a beetle (*Azya orbigera*, Coccinellidae, Coleoptera) that is a primary predator of *C. viridis* (Liere and Perfecto 2008), and an entomopathogenic fungus, *Lecanicillium lecanii*, that infects *C. viridis*. In this report, we discuss our investigation into the possibility that this latter candidate, the white halo fungus *L. lecanii*, could be the source of density-dependent control.

Lecanicillium lecanii, previously known as *Cephalosporium lecanii*, is part of what had been identified as the *Verticillium lecanii* species complex (Kouvelis et al. 1999, Gams and Zare 2001). These entomopathogenic fungi are known to attack a variety of arthropods, many of which are important agricultural pests (Hsiao et al. 1992, Chandler et al. 1993, Helyer 1993, Gindin et al. 1996, Michaud and Browning 1999, Gindin et al. 2000, Rodríguez Dos Santos and del Pozo Núñez 2003) including *C. viridis* in coffee (Easwaramoorthy and Jayaraj 1978, Reddy and Bhat 1989, Uno 2007). It is also marketed as a biocontrol agent (Hall 1981, Khalil et al. 1985a, Khalil et al. 1985b, Ravensberg et al. 1990, Feng et al. 2000). In our study site, *L. lecanii* often creates a local epizootic, killing nearly all of the *C. viridis* on a single coffee bush or a small group of neighboring bushes (personal observations). The importance of honeydew to Hemiptera-tending ants (Helms and Bradleigh Vinson 2008) suggests that such a decimation of a colony's scale populations would substantially decrease colony growth and survival. Therefore, *L. lecanii* may reduce the amount of carbohydrate food available to an ant colony, resulting in an indirect negative effect on colony survival. Analogous increases in ant colony mortality attributable to a natural enemy attacking an ant colony's mutualist partner have been reported for leaf-cutting ants, whose fungal cultivars are attacked by mycoparasites (Currie et al. 1999, Currie 2001, Reynolds and Currie 2004).

To better understand the role of *L. lecanii* in this system, we investigated the distributions of *L. lecanii* at multiple spatial scales and the temporal dynamics of these distributions. Knowledge of the spatial distribution of the fungus, in terms of incidence and severity, is clearly important for assessing the potential for the fungus to influence the self-organization process. How the spatial distribution changes throughout the course of a local infection is a basic component of *L. lecanii*'s natural history and a clear determinant of its impact on the spatial dynamics of the ant mutualist, *A. instabilis*. We also developed a coupled cellular automata model of the ant nests and fungus to demonstrate that it is possible to generate the observed spatial distribution of ant nests using a very simple model that distills the hypothesized pattern formation mechanism into a few simple functions.

Methods

The study site is located at Finca Irlanda, a 300 hectare, organic coffee farm in the Soconusco region of Chiapas, Mexico (15° 11' N, 92° 20' W). The farm is a commercial polyculture, with coffee bushes growing beneath trees that have been planted in an approximately uniform distribution. The dominant shade trees are *Inga* spp., *Alchornea latifolia*, and *Trema micrantha* (Martinez and Peters 1996), some of which have extrafloral nectaries. Previous work had been done by Vandermeer et al. (2008) to map the locations of every shade tree in a 45 hectare plot within the farm and to conduct periodic censuses of *A. instabilis* nest locations. The 45 ha plot is a 600 m X 800 m rectangle with a 100 m X 300 m rectangle excluded from one corner of the plot due to the inaccessibility of the terrain. There are ≈11,000 shade trees in the 45 ha plot, of which ≈300 contain ant nests. The spatial distribution of the ant nests is clumped, with a mean/variance ratio significantly different from a random distribution, and a cluster size distribution that is thought to be characteristic of robust criticality (Vandermeer et al. 2008).

To assess the distribution of the fungus at a large scale, the 45 ha plot was divided into 50 m X 50 m quadrats. Using the available ant nest census data, the shade tree containing an ant nest that was closest to the center of each quadrat was identified. Since the purpose of the survey was to determine the potential for the fungus to contribute to the mortality of existing ant nests, quadrats without ant nests were excluded from the survey. Quadrats at the edge of the plot were also excluded to avoid including areas that might be influenced by unknown factors existing outside of the censused area. The incidence and severity of the fungus were measured in the coffee bushes neighboring each of 56 shade trees between July 8 and Aug 1, 2006. Due to the time required to locate and travel to each shade tree, the order in which the trees were surveyed was determined by their geographic location; time constraints prevented a random survey sequence.

Neighboring coffee bushes were defined as those directly adjacent to the shade tree or within 2 m, whichever resulted in a larger number of bushes. This was necessary because in some locations the nearest bushes were > 2 m from the shade tree, while in others it was impractical to survey all of the coffee bushes in an area with a radius larger than 2 m.

Every branch on every neighboring coffee bush was inspected to see if any scales had been infected. As suggested by its name, "white halo fungus," it is obvious when a scale is in the later stages of infection by *L. lecanii*; the mycelial mat of the fungus forms a distinctive white ring around the infected scale, which is normally a bright green color. If a fungal infection was detected in a location, the severity was ranked as high, low, or medium, as follows: high = one or more neighboring coffee bushes with a scale population with very high levels of mortality due to *L. lecanii*, i.e., having multiple branches with >50% scale mortality; low = one or more neighboring coffee bushes with <10 scales killed by *L. lecanii*; and medium = one or more coffee bushes with fungal mortality between the low and high levels.

The large-scale spatial distribution of the fungus was analyzed using Ripley's K, transformed such that the expectation for all sample sizes is zero for a random spatial pattern and greater than zero for clustered patterns (Goreaud and Pélissier 1999). The survey data were compared by the Monte Carlo method, using 1000 simulated Poisson patterns of fungus presence/absence at the sample locations used in the fungus survey, i.e., accounting for the underlying non-random distribution of the sample points.

To study the distribution and dynamics at an intermediate scale, we identified two clusters of *A. instabilis* nests. Site A had a cluster of four trees and had been intensively studied four years ago, including detailed surveys of scale insects on coffee bushes at various distances from the central *A. instabilis* nest. Four years ago only one of the four trees was occupied by an ant nest, three years ago two trees were occupied, and beginning in 2007 all four trees were occupied. Site B had no ant nests at all during the original census of 2004 and six trees occupied in the 2007 and 2008 censuses, with the three central nests appearing to be the oldest of the six. Thus, by 2008, site A was an "old" site, having been occupied by *A. instabilis* at least since our study began in 2004, while site B was a "new" site, clearly unoccupied in 2004 but having six trees occupied by 2007. At both sites A and B we examined coffee bushes at various distances from the central nest, establishing spatial coordinates for each of the trees examined. For each bush we chose the largest main stem, or for very small bushes we examined the entire plant, and systematically assessed each branch for scale insects and fungal (*L. lecanii*) attack. For making a rapid assessment, we categorized branches, with regard to scale insects, as 1) very low (between 1 and 5 scales), 2) low (between 5 and 25 scales), 3) medium (between 25 and 75), 4) high (between 75 and 125), and 5) very high (more than 125). These assessments were translated into numbers (very low = 2, low = 10, medium = 50, high = 100, and very high = 200), and data represented as average number of scales per branch. In site A 149 bushes and in site B 132 bushes were examined. Assessment of *L. lecanii* infection was based on a percentage, regardless of the number of scales involved (if there was only one scale insect on a branch but it was infected with fungus, the branch was categorized as 100% infected). Intensity of infection was then represented as the number of infected scales per branch.

At the level of individual shade trees, we identified two shade trees with ant nests and high levels of fungal infection, i.e., with large scale populations (>>100 scales) exhibiting a high incidence of fungal mortality. The locations of every coffee bush within 4 m of the central shade trees were measured. The total number of branches, the number of branches with uninfected scale populations, and the number of branches with infected scale populations were counted on each bush during three censuses. Branches with one or more individual scales infected by *L. lecanii* were categorized as “infected.” Shade tree #1 was censused on July 6, July 19, and August 5, 2006. Shade tree #2 was censused on July 8, July 25, and August 7, 2006.

To study dynamics at a smaller scale, a single coffee plant with scale populations in the beginning stages of infection was chosen. There were 4 individual shoots on this plant. Each branch was marked with a letter indicating the shoot (A-D) and a number indicating the branch, starting with the lowest branches, e.g., A1. The healthy and infected scales on each branch were counted on July 7, July 24, and August 6, 2006; only large ($> \approx 2$ mm) scales were counted.

To test the plausibility of the hypothesis that *L. lecanii* acts as the inhibitor in the spatial self-organization process (as elaborated more completely in the discussion), we created a cellular automata model (CA) representing the spatially explicit epizootiology of *A. instabilis* and *L. lecanii*. The model is a version of the ant CA developed by Vandermeer et al. (2008) modified to include the spatial distribution and dynamics of *L. lecanii*. As in the original ant CA model, the 45 ha study plot is represented by a 90 X 120 lattice. Each cell in the lattice can be in one of three states: empty, occupied by an ant nest whose scale insect populations are free of *L. lecanii*, or occupied by an ant nest whose scale insects are infected by *L. lecanii*. As in the original formulation of the model (Vandermeer et al. 2008), the probability of an empty cell being occupied by a new ant nest via satellite expansion of a neighboring nest is a linear function of the number of occupied nests in the eight-cell Moore neighborhood, N ($p_s = s_0 + s_1$N for N > 0; $p_s = 0$ for N = 0). The satellite expansion parameter values used in Vandermeer et al. (2008), which were based on field census data, are also used in this extended version of the model ($s_0 = 0.0035$, $s_1 = 0.035$). Ant nest mortality, which in the original model was a probabilistic function of the number of neighboring ant nests, is now a function *L. lecanii* infection. If infected, the probability of ant nest mortality is equal to the virulence of the fungus, v; otherwise, the probability of nest mortality is zero. If a nest at an infected site survives one time step, the site remains infected in subsequent time steps until the nest dies, i.e., there is no recovery. Transmission of the fungus into an uninfected ant nest is analogous to the ant nest satellite expansion function: the probability of transmission is a linear function of the number of infected nests in the Moore neighborhood, F ($p_t = t_0 + t_1$F for F > 0). In addition, there is a very small probability, *a*, that an uninfected site with no infected neighbors will become infected; this is necessary to prevent the fungus from becoming extinct, but is also biologically reasonable given that *L. lecanii* has been reported to persist in environmental reservoirs, e.g., in the soil (Eapen et al. 2005, Meyling and Eilenberg 2006); infection of isolated sites, with F = 0, represents a low probability transmission from an environmental reservoir.

The purpose of the model was to demonstrate that a simple model incorporating the hypothesized biology of the system (*L. lecanii*-induced mortality or migration of ant nests) could generate the observed spatial distribution. To explore the parameter space of the model, we employed a genetic algorithm (Goldberg 1989, Whitley 1994) to search for values of *v*, t_1 and a that could generate a spatial pattern of ant nests qualitatively and quantitatively similar to the pattern observed in the field. While the initial setup and configuration of a genetic algorithm may be slightly more complicated than some other possible optimization algorithms, e.g., hill climbing, our past experience with other spatially explicit models has shown the potential for results to depend on parameters in complex, non-linear, or counterintuitive ways. Since we had no a priori knowledge of the shape of the fitness landscape, we chose to use a genetic algorithm approach because of its ability to find solutions even when the fitness landscape is discontinuous, noisy, or complex. The quantitative targets were a mean/variance ratio of ≈0.42 (a significantly clumped pattern; Monte-Carlo method using 10,000 simulated Poisson patterns with the same density as the field census data; $P < 0.0001$) and a density of ≈0.03 nests/shade tree, which are values obtained from the field census data. To reduce the size of the search space, we fixed $t_0 = 0$ and $a < 0.01$ for all runs. We ran each simulation for 1000 time steps and calculated the average density and mean/variance ratio of the final 250 time steps. In addition, the fitness function used in the genetic algorithm included a term for the stability of the density and mean/variance time series to ensure that the model had reached steady state by the end of the run.

Results

At the large scale, of the 56 locations sampled, 32 (≈57%) exhibited signs of *L. lecanii* infection. The number of locations in each of the four severity categories were: absent, 24 (≈43%); low, 21 (≈38%); medium, 3 (≈5%); and high, 8 (≈14%). There was no obvious pattern underlying the spatial distribution of the fungus at the scale of the 45 ha plot. According to the Ripley's K analysis, below a sampling circle radius of ≈160 m the distribution of the fungus does not differ significantly from the random expectation (given the underlying distribution of sample points); at some spatial scales above a radius of ≈160 m, the distribution is significantly more uniform than random, but at other scales it does not differ significantly from the random expectation (Figure III.1).

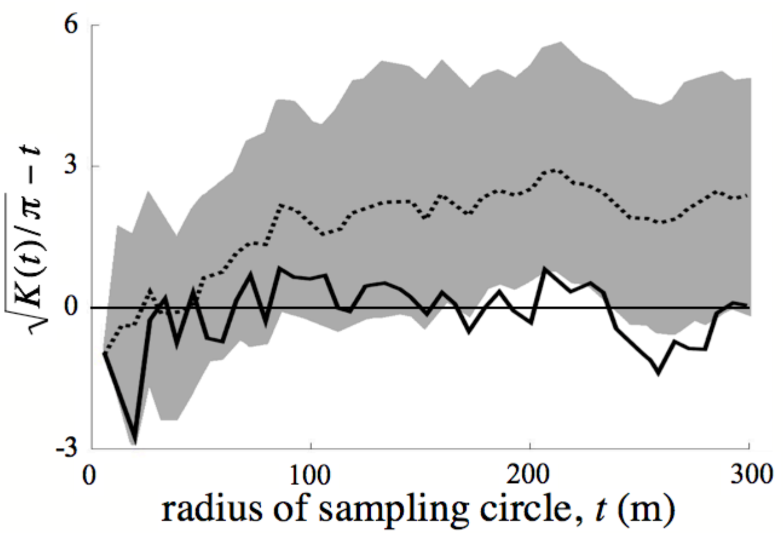


Figure III.1. Graph of transformed Ripley's K versus radius of sampling circle. The dashed line is the average value for 1000 random, simulated fungal distributions. The shaded area delineates 95% confidence intervals. Simulated distributions were created by randomly allocating the observed instances of fungal presence among the sample points, thereby accounting for the underlying spatial distribution of the sample locations. The solid line is the transformed Ripley's K for the field data.

At the meso scale, site A has been monitored for the past 4 years, so the sequence of occupation of individual shade trees by *A. instabilis* is known precisely, as shown by the arrows in Figure III.2. The distribution of both scale insects and the white halo fungus disease is also shown in Figure III.2. From previous sampling we know that the distribution of scale insects as a function of distance to the central tree (one of the two occupied four years ago) is decreasing (Figure III.2a). Except for a very large concentration of scale insects within about 5 meters from the shade tree containing *A. instabilis*, the density of scales declines rapidly as the distance from the central tree increases. At a distance of more than 10 meters, the majority of coffee bushes have only a few scale insects, with an occasional tree containing a larger cluster, always tended by a different ant species, although never at the level reached under the protection of *A. instabilis* (Figure III.2). It is worth noting that, although we did not explicitly search for *L. lecanii* four years ago, it is unlikely that it occurred very commonly since our field notes would have reflected its presence (indeed, it is most likely that *L. lecanii* was not present at all at this site four years ago).

Site B was sampled in the summer of 2008, but from previous surveys we know that the entire area surrounding where the six ant-occupied trees are currently located was free of *A. instabilis* colonies until recently (between one and three years prior to the summer of 2008). That is, this particular cluster of ant nests is young, having been established subsequent to 2005. The population densities of *C. viridis* are slightly lower than in site A, and the distribution of the white halo fungus disease is more restricted (Figure III.2b), both patterns of which can be explained by the young age of this cluster of ant nests. It is obvious from the field observations that three of the six occupied shade trees are not at all associated with large densities of the scale insects (Figure III.2b), suggesting that they were more recently occupied than the three shade trees around which the high densities of scale insects occur.

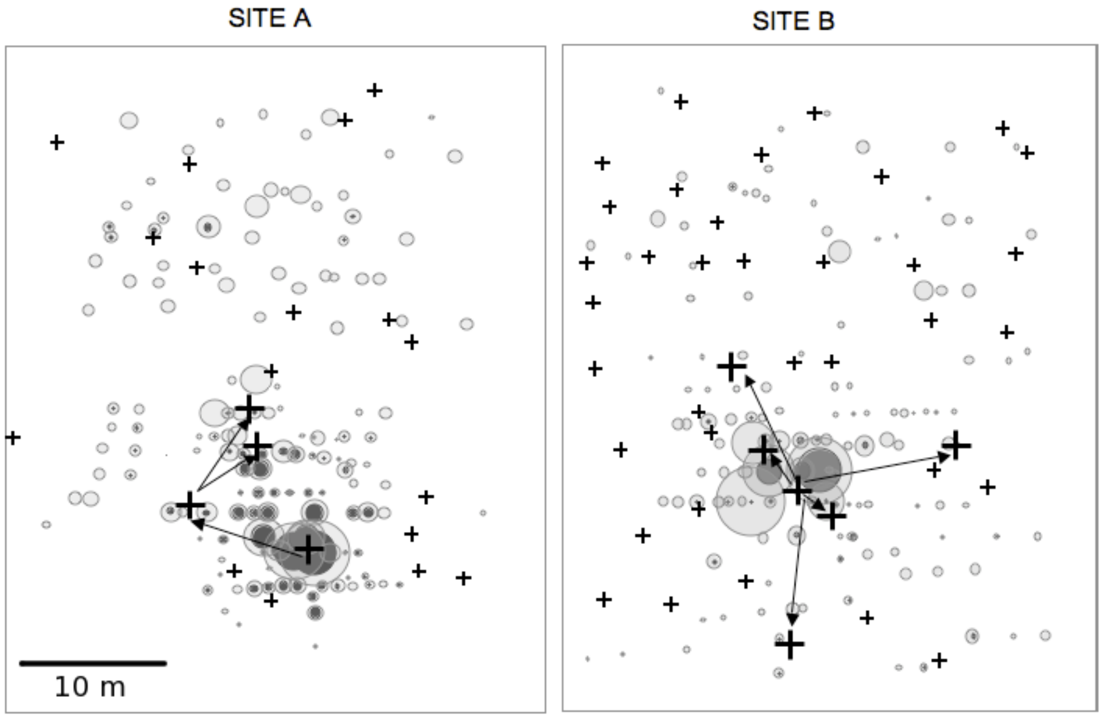
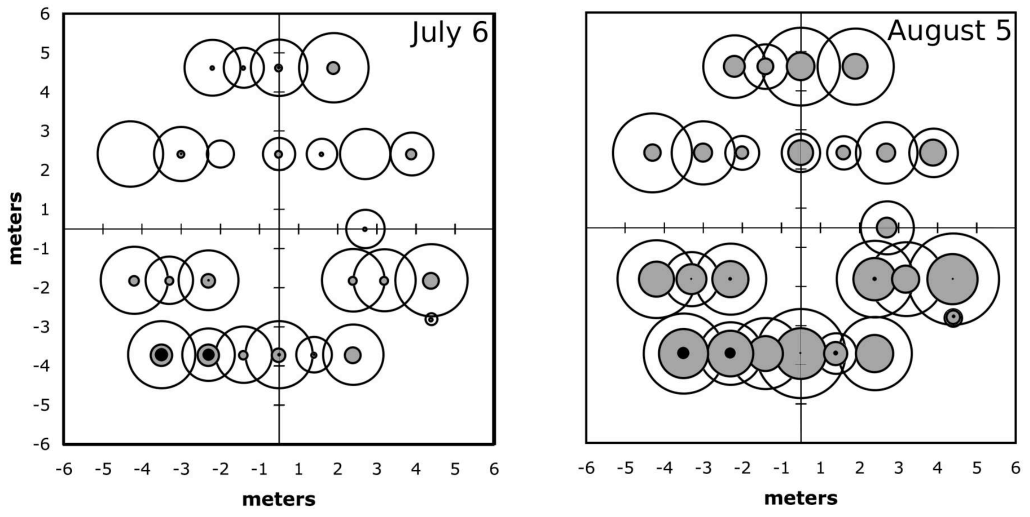


Figure III.2. Representation of two intensively sampled sites in the study area. Final sample in June/July, 2008 is shown. Site A was occupied by *A. instabilis* at least since 2004, while site B was newly occupied sometime within the past three years. The size of the slightly shaded bubbles is proportional to the number of scale insects per branch of a coffee bush located at that particular coordinate. The size of the darkly shaded bubbles is proportional to the intensity of fungal disease (caused by *L. lecanii*) on that bush. Large crosses indicate positions of shade trees occupied by *A. instabilis* and small crosses indicate positions of unoccupied (and presumably occupiable) shade trees in the system. Arrows indicate presumed direction of spread of the ant colony from historical records. Plots are both 40 X 50 meters.

At the level of individual shade trees, the initial and final distributions of scales and fungus around the two shade trees are shown in Figure III.3. In the initial survey of the coffee bushes surrounding shade tree #1 (Figure III.3a), the branches with the largest scale populations were located in two bushes in the lower left quadrant. Some of the scale populations on these bushes were already infected by *L. lecanii*. By the second census, the number of branches with scale populations in the lower quadrants, i.e., next to the bushes with the largest initial scale populations, had increased substantially, but the fungus was still largely confined to the two original bushes. Between the second and third censuses, scale populations had been established on multiple branches in all of the coffee bushes, but the level of fungus outside of the original two heavily infected bushes remained roughly equal to the initial level (6 bushes with 1-2 infected branches in the initial survey, 7 bushes with 1-4 infected branches in the final survey).

A higher proportion of coffee bushes in the neighborhood of the second shade tree (Figure III.3b) already had established scale populations infected by *L. lecanii* at the beginning of the census. Throughout the censusing period, there was an increase in the number of branches with scales, but there was not a substantial spread of fungus to previously uninfected coffee bushes; 3 plants that were initially uninfected had 1-2 infected branches by the time of the final survey.

(a)



(b)

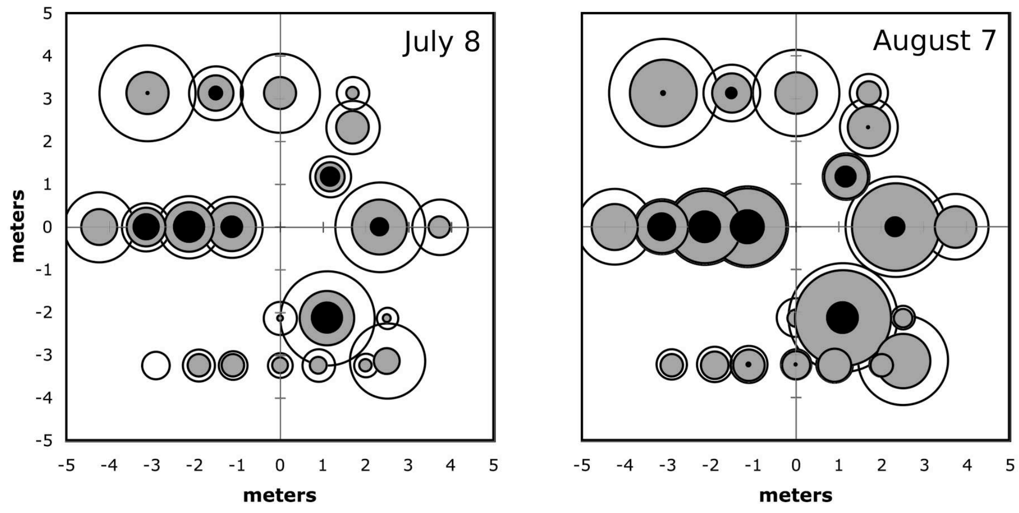


Figure III.3. Scales and fungus in coffee bushes surrounding two shade trees. Shade trees are located at (0, 0). The sizes of the white, gray, and black circles are proportional to the number of: total branches, branches with uninfected scale populations, and branches with infected scale populations, respectively. (a) shade tree 1, July 6 and August 5, 2006 (b) shade tree 2, July 8 and August 7, 2006

The distribution of healthy and infected scales on each of the four shoots of the coffee bush chosen for the individual coffee plant-level census are shown in Figure III.4. Shoots B and C, which had the largest scale insect populations, reveal pronounced humped distributions of the scales across the branches. Moving from the low branches (small branch numbers) to the high branches (large branch numbers) on the shoots, the size of the scale insect populations generally increased until the top few branches, which generally had much smaller populations per branch due to the relatively small physical size of these younger leaves. Incidence was relatively low in the initial census, with the majority of branches showing little or no evidence of infection. By the second census, *L. lecanii* infection was more prevalent, but the majority of the scales on all of the branches were still healthy. Between the second and third censuses, there was a general increase in the size of the scale populations and infection by *L. lecanii* spread to all of the branches, with the populations on many of the branches experiencing 50% or greater mortality due to *L. lecanii*.

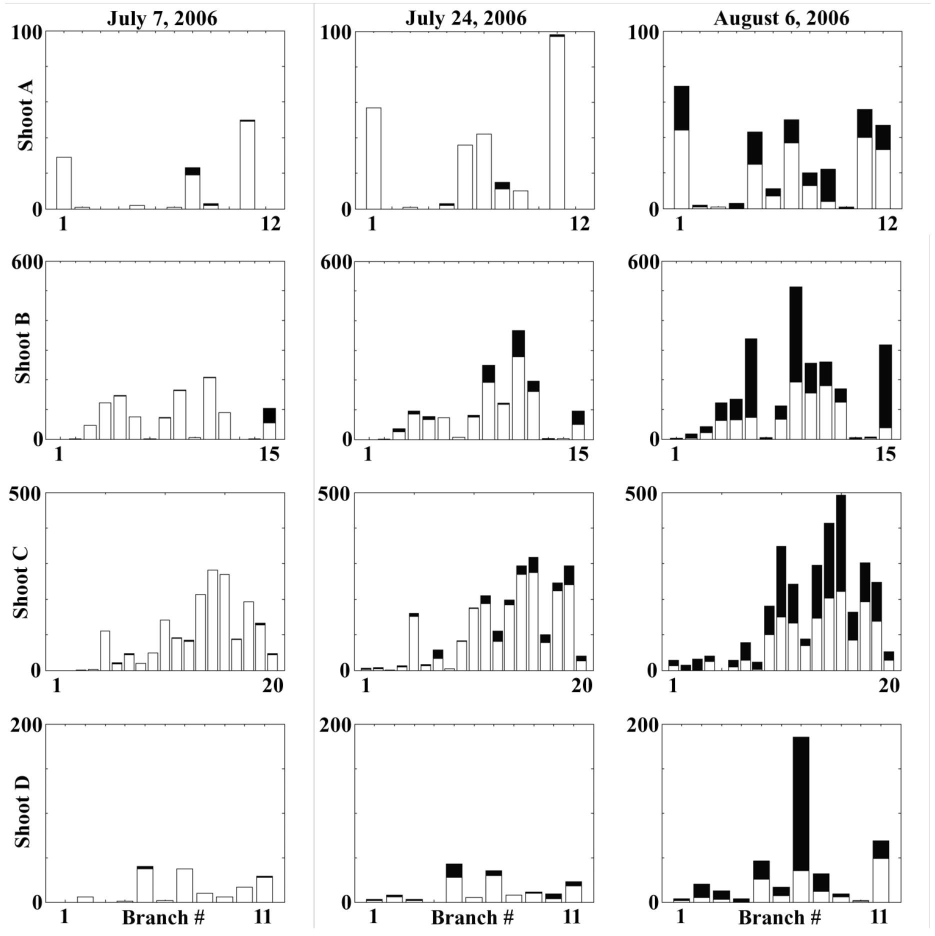


Figure III.4. Number of healthy (white) and infected (black) scales on the branches of 4 shoots on a single coffee bush. Branches with higher numbers are higher on the shoot. Data from three censuses (July 7th, July 24th, and August 6th, 2006) are shown.

Using the ant/fungus epizootiology CA model, we find it is possible to generate qualitatively and quantitatively similar ant nest spatial distributions using a range of values for *v* and t_1. As for parameter estimation using the genetic algorithm, the highest fitness parameter values were confined to a narrow band of *v* and t_1 values (Figure III.5), with parameter values away from this region unable to generate the target spatial pattern regardless of the value of *a*. In Figure III.6 we show a representative snapshot of the results of the model with parameters $v = 0.35$, $t_1 = 0.63$, and $a = 0.007$. As can be clearly seen, the qualitative nature of the nest clustering reported in Vandermeer et al. (2008) can be reproduced with this model. The ranges of densities and mean/variance ratios generated by this model (Figure III.7) also encompass the values for the field samples reported in Vandermeer et al. (2008).

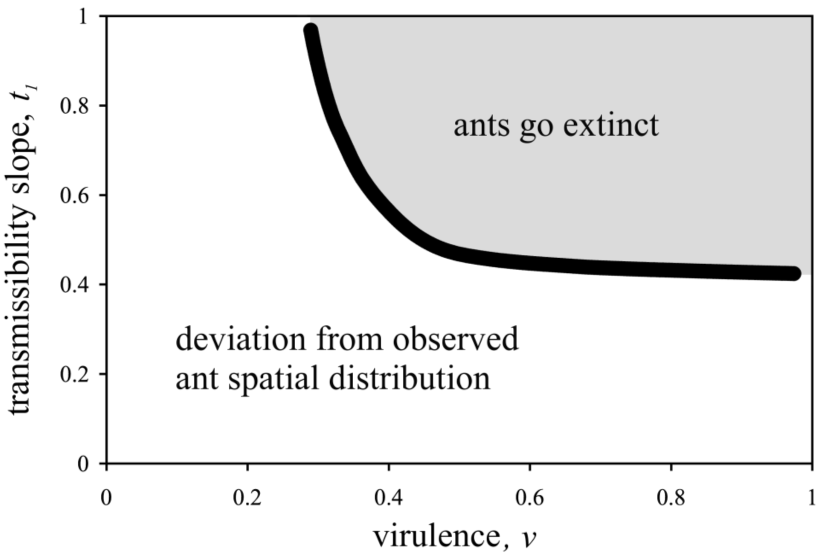


Figure III.5. The black line represents the high-fitness region in v, t_1 parameter space in which it is possible to generate spatial patterns of ant nests that are qualitatively and quantitatively similar to the pattern observed in the field. Away from this region, it is not possible to generate the observed spatial distribution for $t_0 = 0$ and $a < 0.01$; in the gray shaded region, the ants go extinct for most values of *a*.

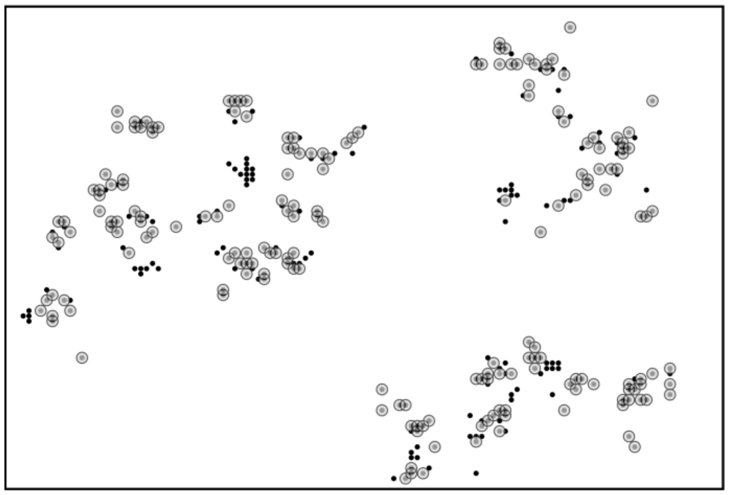


Figure III.6. Example snapshot of the ant and fungus CA model for $v = 0.35$, $t_0 = 0$, $t_1 = 0.63$, and $a = 0.007$. The black dots indicate the locations of ants nests. The shaded circles indicate nest sites infected by *L. lecanii*. Note that the model only considers the presence of the fungus and not its intensity.

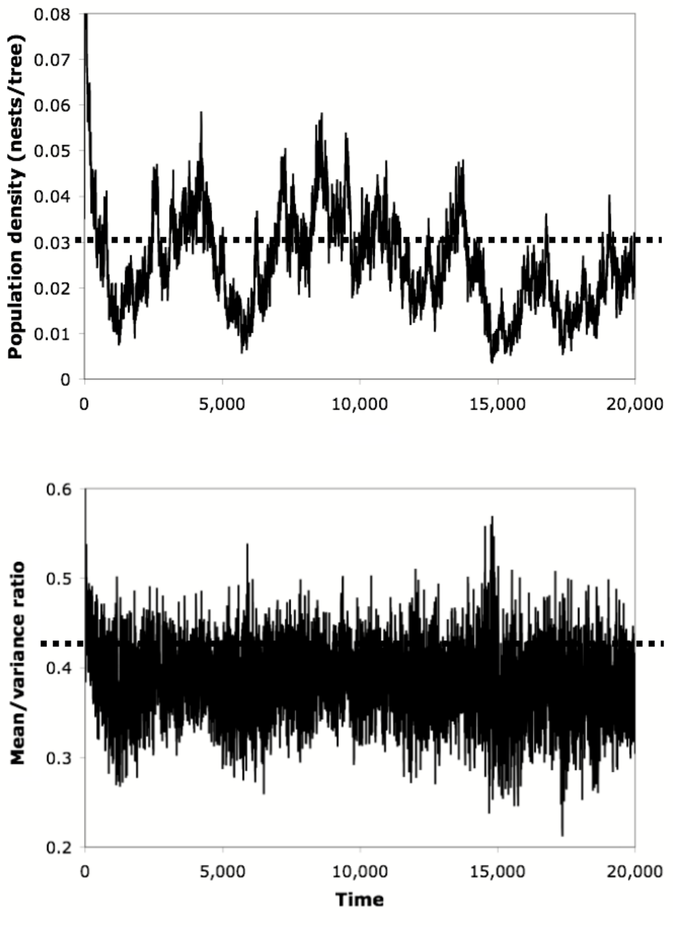


Figure III.7. Time series for a representative run of the ant and fungus CA model for $v = 0.35$, $t_0 = 0$, $t_1 = 0.63$, and $a = 0.007$. Dashed lines indicate the density and mean/variance ratio targets used for the genetic algorithm search. Each model time step corresponds to a six-month interval.

Discussion

Although the fungus is relatively common in the coffee bushes surrounding ant nests in the 45 ha plot (≈57% of shade tree neighborhoods sampled showed some signs of *L. lecanii* infection), it is doubtful that the local intensities most commonly encountered are sufficient for the fungus to significantly influence the spatial distribution of ant nests. Therefore, the frequency of high severity fungal infections, which only occurred in ≈14% of the shade tree neighborhoods sampled, is probably the most appropriate measure to consider when assessing the potential influence of *L. lecanii* on pattern formation of the ant nests.

Furthermore, at this large scale, no clear pattern of fungal distribution could be seen, although its presence is widespread. Casual observations prior to the formal survey led us to believe that the fungus was absent in the majority of the area and much more prevalent in one particular half of the plot. However, it was clear from our survey that it is difficult to determine with any certainty whether the fungus is present in a location without examining every single leaf and branch of every coffee bush, as there are many locations where the fungus infects only one or a few scales. Without a thorough search, detecting fungal infections in lightly-infected locations is unlikely. Therefore, it is perhaps not surprising that a more systematic survey failed to support our preliminary assessment. The fungus was not obviously more prevalent in a particular half of the plot, and the Ripley's K analysis indicates that the distribution of the fungus is not significantly different from random at most spatial scales; if anything, it tends towards a uniform distribution, which is directly opposite of what was suggested by our initial assessment.

Because sporadic infections are common, it is possible that *L. lecanii* is present everywhere in the plot, lying latent in an environmental reservoir. It has been reported that some strains of white halo fungus can persist as saprotrophs in the soil (Eapen et al. 2005, Meyling and Eilenberg 2006). If the variety in our samples is able to live in the soil, new infections of scale populations in a given location may be more a matter of fungal spores being transmitted from the soil as opposed to the spores being transmitted from active infections on other coffee plants (Jackson et al. In press). This explanation would be consistent with the wide-ranging, sporadic distribution of the fungus throughout the plot. However, it is important to keep in mind that repeated surveys at a higher spatial resolution might reveal an underlying spatial distribution pattern that was not possible to resolve with the method used in our study.

At the meso scale (Figure III.2) it is possible to deduce the general behavior of the fungal disease if we consider site A four years ago as a base line case (since we did not encounter the fungal disease at that time), site B in 2008 as an intermediate case and site A in 2008 as a more advanced case. The pattern that exists today, coupled with the pattern of migration of the *A. instabilis* nest, strongly suggests that the ant nest moves partially in response to the fungal infection of its main food source, leaving a trace of scale populations devastated by the disease near the locus of the original ant nest site, and scale populations built up but not yet infected nearer to the more recently occupied shade trees. In Figure III.8 we present the log of the intensity of fungal disease along with the log of the scales per branch for those three stages. The progression of both the disease and the scale insects is apparent, the scale insects slowly building up local population densities and dispersing, the fungal disease following in an epizootiological fashion.

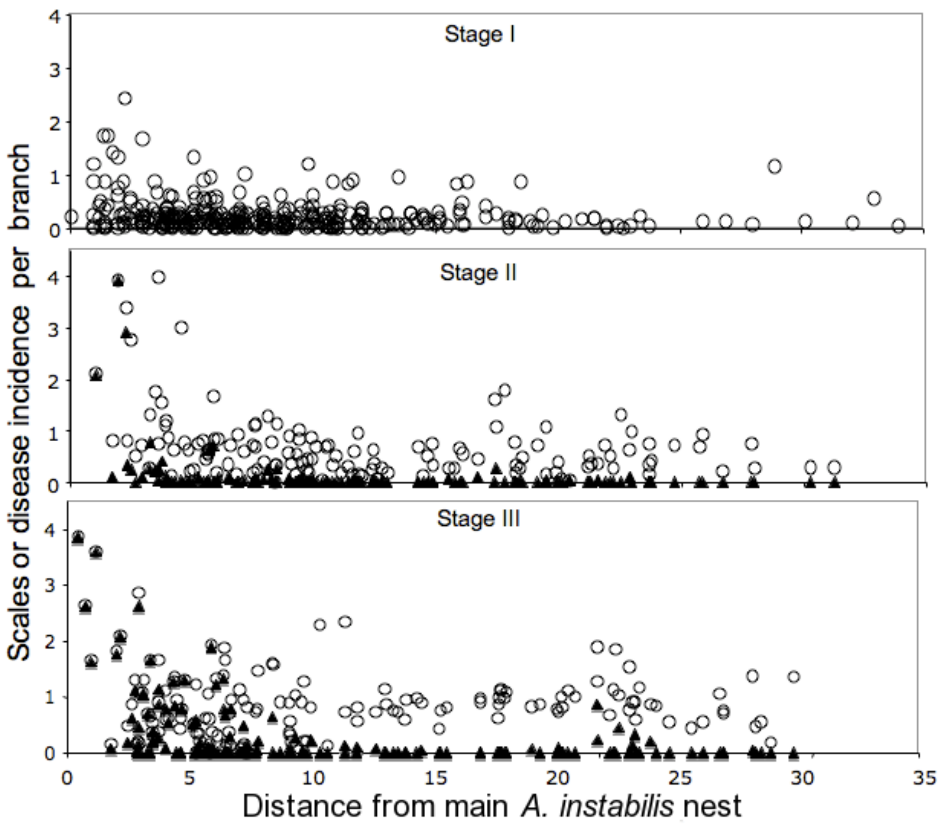


Figure III.8. Data from two sites at two different times, illustrating the stages of development of the fungal pathogen, *L. lecanii* and its host *C. viridis*. Open circles = *C. viridis* (the scale insect), closed triangles = *L. lecanii* (the fungus that causes white halo disease in the scale insect). In all three cases the log of the insect and fungal abundances are plotted as a function of the distance to the main ant nest in the system. In Stage I are the data from site A in 2004, when the fungal disease was absent and the scale populations seemed to be on the increase both locally and in space. In Stage II are the data from site B in 2008, seemingly representing a situation in which the fungal disease has only recently arrived in the area and is beginning its spread throughout the general area, but has not extended much more than five or six meters beyond the initial infective zone. In Stage III are the data from site A again, but from 2008, where we see the major expansion of the fungal disease that seems to be following the expansion of the scale insects in space. Stage IV (not pictured here) is represented in several cases in our plots, in which the entire system, ant-scale-fungus, have locally disappeared entirely.

Part of the dynamics of this system, as is evident from a casual examination of Figure III.2, is the maintenance of *C. viridis* in the absence of the major ant mutualist. That is, once one moves more than approximately 10 meters from the main nest, it is very unlikely that *A. instabilis* will be tending scales. Nevertheless, there are always some bushes to be found with a relatively high concentration of scale insects, although never on the same order of magnitude as when they are with *A. instabilis*. These outliers are always tended by other species of ants (personal observations). Indeed, in a separate study we have encountered almost 80 species of ants that are potential tenders of the scale insects (Philpott et al. 2006). However, none of them has ever been observed as being as effective as *A. instabilis*, and we have never encountered a coffee bush with more than 250 scale insects that was not under the protection of this primary mutualist. However, those other ants are critical to the system in that they maintain the scales over a large region, albeit at a relatively low density.

Also part of the dynamics is the evident fact that the scales are always present at very low numbers, even in the complete absence of tending ants. While it always appears to field workers that there are zero scales in the absence of one of their ant mutualists, careful searching invariably reveals one or two scales on almost every coffee bush in the plantation. It is most likely that this low but consistent population density is maintained by a continual rain of crawlers blowing around the farm, emanating mainly from the centers established by *A. instabilis*.

Moving to a lower spatial scale, the maps of the fungus in the coffee bushes neighboring a single shade tree suggest that scale populations primarily spread locally from bush to bush, since the bushes with high numbers of branches containing scales tend to be close to one another; this would be consistent with a propagation of scales from one or a few initial populations in a neighborhood of coffee bushes. The censuses performed at the shade tree-neighborhood level indicate that it would be necessary to initiate censuses earlier in the wet season in order to capture the spatial and temporal dynamics of the fungus spread at this spatial scale (Figure III.3).

It is important to note that *A. instabilis* very probably plays a central role in the dynamics of *L. lecanii* infection of *C. viridis*. Field observations suggest that there is a minimum abundance and density of *C. viridis* that are necessary for an outbreak of *L. lecanii* to occur. When such an outbreak occurs, the fungus becomes locally epizootic, killing the vast majority of scales on entire coffee bushes. Without *A. instabilis* tending the scales and therefore providing protection from predators and parasitoids, the scale population is unlikely to reach a sufficient size for a fungal outbreak to occur (Uno 2007).

Considering the results as a whole, a general picture of the overall spatial dynamics emerges. Scale insects initially arrive at coffee plants more-or-less as propagule rain, being carried by the wind. While local increase in scale abundance is clearly from local reproduction, there is also undoubtedly a general dispersion since almost every coffee plant in the entire coffee farm has one or two scale insects on it. The second stage in the overall dynamics depends on ants other than *A. instabilis*, generally. Acquiring the protection of one of these other ants allows the scale insects to build up a local population density above the normal background density, such that if *A. instabilis* did not exist in the system at all, the local build up of intermediate densities of scales would probably not change, but the size of propagule rain would, since the majority of propagules probably comes from the clusters of *A. instabilis* nests, as reflected in the distribution of scales as a function of distance from ant nests (Figures III.2 and III.8). The consequence of these fundamental scale insect spatial dynamics is an approximate general power function distribution of scale insects (Alonso and Pascual 2006, Pueyo and Jovani 2006, Vandermeer and Perfecto 2006a, Vandermeer and Perfecto 2006b), with a very few large clusters of individuals on a coffee bush (never as large as they get under protection of *A. instabilis*), but a huge number of coffee bushes with just a few individuals.

The dynamics of *A. instabilis* thus confronts the prospect of encountering these clusters of scale insects as it searches for additional nesting sites. When an *A. instabilis* nest seeks to expand its colony, it establishes a satellite colony in a nearby shade tree (occasionally a coffee bush) and begins the search for scale insects and other sources of carbohydrates. Other species of honeydew-producing insects are also sources (Livingston et al. 2008), but the major source is *C. viridis*. Since it is not the case that all individual shade trees harboring an *A. instabilis* nest are surrounded by coffee bushes with large concentrations of scale insects, it must be the case that occupation of a shade tree is not conditioned by the presence of this mutualist, but rather that the mutualism develops later, probably mainly from the initial clusters of scale insects produced by mutualism with other ants. It remains to be seen exactly what the survival probability of a nest in an individual tree is either with the development of a large *C. viridis* population or not, but it seems a reasonable speculation that a "trial" *A. instabilis* nest in a new tree may be abandoned if no *C. viridis* population can be cultivated soon. On the other hand, other honeydew-producing insects in the shade trees, as well as extrafloral nectaries in those shade trees may serve this purpose also.

A successful new *A. instabilis* nest seems to almost always result, eventually, in very large clusters of *C. viridis* within about 5 meters of the nest tree, resulting in a key deviation from the underlying statistical distribution of the scale insects themselves (Alonso and Pascual 2006, Pueyo and Jovani 2006, Vandermeer and Perfecto 2006b, Vandermeer and Perfecto 2006a). However, the very large concentrations of *C. viridis* provide a locus for the epizootic development of *L. lecanii*. The dispersal dynamics of the latter are not completely understood, but it is clear that at least three dispersal phenomena are involved: 1) individual scales become infected seemingly at random and not necessarily associated with the local population density of scale insects, 2) epizootic spread of the disease within a high-density population of the scale on an individual branch of a coffee bush occurs predictably, partially as a result of ant foraging (Jackson et al. In press), 3) local spread from a coffee bush to neighboring coffee bushes also occurs, but in an unpredictable and relatively slow fashion. It is this third mode of dispersal that may have the most important consequences as far as the *A. instabilis* is concerned, for it seems that a local epizootic of white halo fungus spreading locally from bush to bush is one of the causes of the ant nest searching out new shade trees for establishment of satellite nests.

So the general picture emerges of the *A. instabilis* nest establishing in a new shade tree and searching out local coffee bushes for local concentrations of *C. viridis*. Having encountered local concentrations, the mutualistic effect of the ant permits the scales to build up to extremely high population densities in bushes near to the shade tree containing the new nest. However these extremely large populations of scale insects become targets for the epizootic development of the white halo fungus which, once established in an area, appears to become endemic, following the ant colony around as new shade trees are occupied, eventually, perhaps, resulting in the death of an entire cluster of ant nests (or a large-scale abandonment of the area and migration to some more distant site). This basic natural history generates the rationale for the double CA model, as described in the methods section. The fundamental question to be answered is whether this natural history (as represented qualitatively in the CA model) is capable of producing the self-organized clustering pattern of ant nests that we see in the field; as shown in Figure III.6, the answer is affirmative.

The significance of this mechanism of self-organized spatial pattern is dual. First, it is arguable that the dynamics and propagation of *L. lecanii* creates the conditions for its own survival. Since epizootics only occur when the scale insect population reaches a critical size, and since that critical size only occurs when ants are tending the scales, it is clear that ants are necessary for the production of the epizootics. If the fungal pathogen drives the shifting pattern of the ant/scale mutualism, it could be said that the fungus creates the background conditions that are necessary for its survival because of its potential to influence the spatial distribution of *A. instabilis* nests. Second, expanding our knowledge of the spatial ecology of this fungal pathogen is important because of the role *L. lecanii* may play in the biological control of important coffee pests. In addition to attacking *C. viridis*, which has the potential to reach pest status if not under some natural control, *L. lecanii* has also been shown to be a hyperparasite of the coffee rust, *Hemileia vastatrix* (Shaw 1988, Eskes 1989, González et al. 1995). The magnitude and spatial extent of the control of *H. vastatrix* by *L. lecanii* clearly depend on the spatial distribution of *L. lecanii* (Vandermeer et al. 2009), so obtaining a better understanding of the spatial and temporal characteristics and propagation of *L. lecanii* is an important component of understanding and improving the biocontrol potential of *L. lecanii*, certainly in the case of the green coffee scale and possibly in the case of the coffee rust.

CHAPTER IV

Indirect biological control of the coffee leaf rust, *Hemileia vastatrix*, by the entomogenous fungus *Lecanicillium lecanii* in a complex coffee agroecosystem

Agriculture has long been recognized as playing a central role in the development and survival of modern civilizations. It is also well known that from the very beginning of agriculture there have been organisms coexisting in close association with the primary crops of interest, some of which have been able, under favorable conditions, to proliferate to such an extent that they become economically important pests. Through their devastating effects on agriculture, these pests have sometimes had profound and long-lasting effects. Prominent among these pests are plant pathogens, such as *Puccinia striiformis*, which causes stripe rust that can decimate entire fields of susceptible wheat varieties (Chen 2005), and the oomycete *Phytophthora infestans*, the causative agent of the potato blight that contributed to the Great Irish Famine and the resultant decimation of the population of Ireland (Fry 2008), to mention just two of the more well-known examples.

The coffee rust *Hemileia vastatrix* Berkeley and Broome is likewise a plant pathogen of great historical import, and one of the most important diseases of Arabica coffee in the world. Heavy infections cause decreased photosynthesis and increased defoliation (Kushalappa and Eskes 1989), and producers continue to incur significant costs due to crop losses and mitigation efforts, with yield losses of 6-13% and annual costs worldwide due to coffee leaf rust estimated to be US$1 billion (Hein and Gatzweiler 2006). In the late 1800's, *H. vastatrix* swept through the coffee growing regions of Sri Lanka (then Ceylon) and southern India, leading to the abandonment of coffee as a major crop in these areas (McCook 2006). In 1970, the detection of *H. vastatrix* in Brazil led to great concern that a rust epidemic in Latin America was imminent. However, a devastating Latin American epidemic of the magnitude that was experienced in South Asia has not yet materialized, although there is growing concern that the severity of coffee rust will increase under climate change (Ghini et al. 2011).

Coffee rust is currently controlled primarily through application of copper fungicides, the use of resistant cultivars, and cultural methods, such as reduction of shade cover. However, there are significant drawbacks to each of these approaches. Copper fungicides have been shown to increase the abundance of coffee leaf miners and coffee mites (Eskes et al. 1991), and there are significant concerns about their effects on human health (Loland and Singh 2004, Kanoun-Boulé et al. 2008). Development of durable genetic resistance in the face of variability in the pathogenicity of *H. vastatrix* continues to be a challenge (Brito et al. 2010). Finally, reducing shade cover in coffee growing regions has been demonstrated to have a strong, detrimental effect on biodiversity (Perfecto et al. 2003).

In light of the aforementioned problems with conventional control approaches, there has been continued interest in the biological control of coffee rust (Shiomi et al. 2006, Haddad et al. 2009) and other alternative control strategies (Avelino et al. 2004). The entomopathogenic and mycoparasitic fungus *Lecanicillium lecanii* (Zimmerman) Zare and Gams has been of particular interest, primarily in terms of its use as an augmentative biological control agent (Kushalappa and Eskes 1989, Canjura-Saravia et al. 2002), which entails the application of additional inoculum to bolster naturally-occurring populations of the biocontrol agent. However, results of field trials have been mixed. Alarcón and Carrión (1994) reported the successful establishment of *L. lecanii* on *H. vastatrix* in experimental plots that had been sprayed with a fungal suspension and the subsequent spread of *L. lecanii* into unsprayed control plots. In contrast, Eskes et al. (1991) saw no development of hyperparasitic growth of *L. lecanii* on *H. vastatrix* in the field, despite having demonstrated hyperparasitic activity on coffee rust in the laboratory. They attributed this failure to low air humidity, other environmental factors, or antagonists in the phylloplane.

Given the potential difficulty of employing *L. lecanii* as an augmentative biological control, a strategy based on conservation biological control may prove to be more effective. Conservation biological control involves management of agroecosystems such that the persistence and efficacy of natural pest controls is enhanced (Barbosa 1998, Pell et al. 2010).

The development of a successful conservation biological control program will require a thorough understanding of the ecology of both pest and pest control agent, which includes verifying that naturally-occurring *L. lecanii* can significantly reduce the prevalence or severity of *H. vastatrix* under field conditions. Some progress has been made towards this goal by Vandermeer et al. (2009), who showed, using field surveys of *L. lecanii* and *H. vastatrix* prevalence in an organic coffee farm in southern Mexico, that the presence of *L. lecanii* is correlated with a significant reduction in the prevalence of *H. vastatrix*.

Although the Vandermeer et al. study demonstrated that there is a significant negative correlation between the abundance of *L. lecanii* and the prevalence of coffee rust within the same year, we hypothesize, based on the natural history of the two fungi, that there may be an effect across years that could be as strong, or even stronger, than the within-year effect. *Hemileia vastatrix* is generally considered to be a biotrophic, autoecious rust, i.e., it can only survive on living host tissue, and there is no known alternate host (Kushalappa and Eskes 1989, Moricca and Ragazzi 2008). Therefore, antagonists must attack the rust directly on living coffee leaves when the rust is active. This implies that a high abundance of *L. lecanii* at the beginning of the period of high rust activity, as a result of proliferation of this antagonist in the previous year, may play a powerful role in the prevention of a rust outbreak by curtailing the reproduction of the rust before it has an opportunity to become locally epidemic.

In our study system, located at the same site as the Vandermeer et al. study, the abundance of *L. lecanii* is largely determined by the abundance of its primary host, the green coffee scale *Coccus viridis* Green (Hemiptera: Coccidae). *Lecanicillium lecanii* forms a conspicuous white halo of mycelia and sporulates freely in its advanced stages of infection on *C. viridis*. In the presence of its mutualistic partner, the arboreal nesting ant *Azteca instabilis*, *C. viridis* typically reaches very large population sizes – on the order of hundreds, or even thousands, of individuals per coffee plant. These large populations are susceptible to epizootics of *L. lecanii*, and therefore serve as a major source of inoculum. Therefore, the spatial distribution of the *A. instabilis* colonies determines where populations of *C. viridis* will flourish, thus indirectly influencing the spatial distribution of *L. lecanii*, which in turn may affect the prevalence of *H. vastatrix* (Figure IV.1).

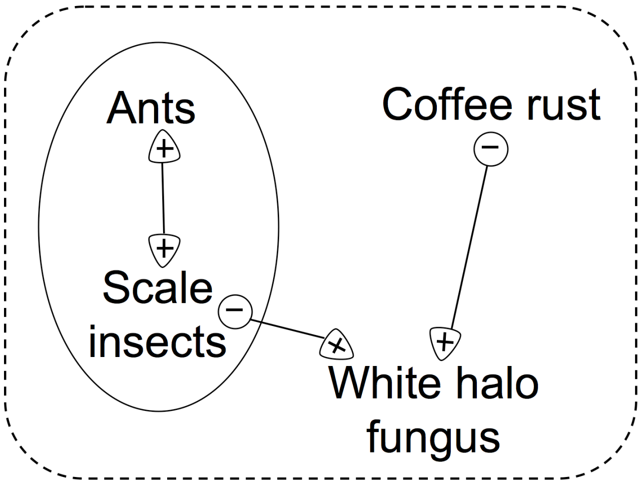


Figure IV.1. The basic biology of the system. The ants (*A. instabilis*) are mutualistically associated with the scale insects (*C. viridis*), indicated by positive arrows. The white halo fungus (*L. lecanii*) has a negative effect on the scale insects, indicated by a negative circle, as well as a negative effect on the coffee rust (*H. vastatrix*). The ants and scale insects occur in spatially restricted pockets on the farm, indicated by the oval containing them. The farm as a whole, indicated by the dashed rounded rectangle, contains the white halo fungus and the coffee rust.

It is possible that spores from active epizootics could directly attack *H. vastatrix* within the same season. However, the soil has been shown to serve as an environmental reservoir of viable propagules of *L. lecanii* (Meyling and Eilenberg 2006), and these propagules can be translocated from the soil onto the coffee plant via rain splash (Jackson et al. In press), so it is also conceivable that spores of *L. lecanii* accumulate in the soil during one wet season and attack the rust when it emerges from dormancy during the subsequent wet season (Waller 1982). This scenario would imply that the prevalence of *H. vastatrix* would be affected by the abundance of *L. lecanii* in the previous wet season, i.e., there would be a one-year lag in the effect of *L. lecanii* on rust prevalence.

To test this hypothesis, we compared the abundance of *L. lecanii* and the prevalence of *H. vastatrix* across two years, in sites subject to epizootics of *L. lecanii* associated with the *C. viridis*-*A. instabilis* mutualism.

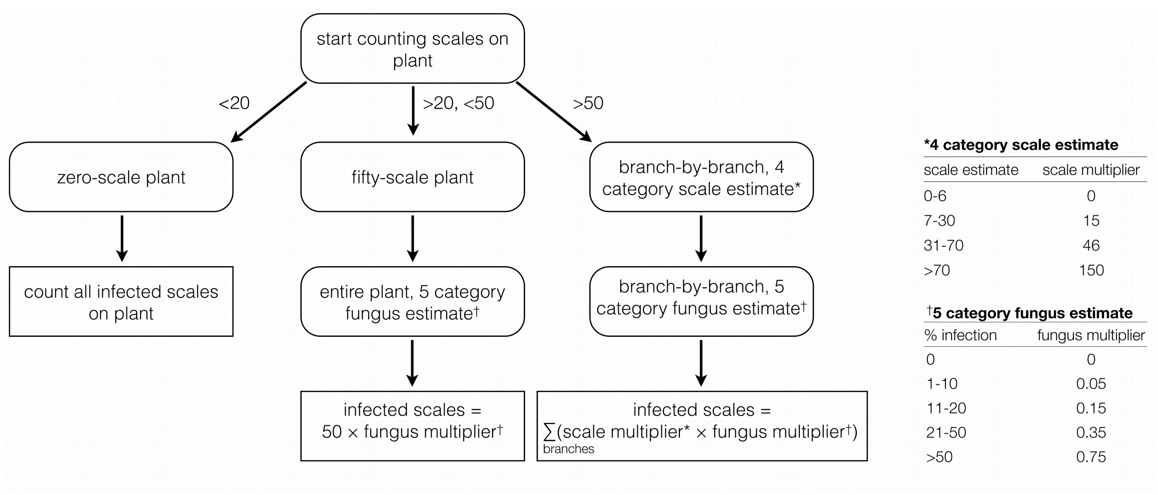
Methods

Experimental location and cropping system

The study was conducted in Finca Irlanda, a certified organic, shade-grown coffee farm in the Soconusco region of Chiapas, Mexico. Two experimental sites, Site A and Site B, were chosen in order to encompass active *A. instabilis* colonies. According to biannual censuses of the study sites, the *A. instabilis* nest in Site A was established in 2007, and Site B was occupied by one or more *A. instabilis* colonies from the first survey, in 2001. Site A included 470 coffee plants, in an area of approximately 50×50 m, and Site B comprised 415 plants, in an area of approximately 30×40 m.

Surveys

Surveys of *L. lecanii* and *C. viridis* were conducted in both sites in September 2009. The identity of *L. lecanii* as the prominent fungal antagonist of *C. viridis* in this system has been confirmed based on morphological identification using the characteristic conidia and diagnostic phialides (Zare and Gams 2001) and by DNA sequencing of infected scales (Jackson, unpublished data). A rapid-survey protocol, adapted from Perfecto and Vandermeer (2006), was employed to estimate the abundance of healthy and *L. lecanii*-infected *C. viridis* on every coffee plant in the study sites. For each plant, an individual-by-individual count of *C. viridis* adults (greater than approximately 7 mm in width) was started. If more than 50 scales were encountered on the plant, the individual count was abandoned in favor of a less time consuming branch-by-branch protocol. If less than 20 individuals were encountered on the plant, the total number of infected individuals on the plant was counted. If between 20 and 50 individuals were found, an estimate of the overall prevalence of *L. lecanii* was used to determine a fungus multiplier for the entire plant, and the total number of infected scales was estimated to be 50 times the fungus multiplier (Figure IV.2).


start counting scales on plant
<20
>20, <50
>50
zero-scale plant
fifty-scale plant
branch-by-branch, 4 category scale estimate*
count all infected scales on plant
entire plant, 5 category fungus estimate†
branch-by-branch, 5 category fungus estimate†
infected scales = 50 × fungus multiplier†
infected scales = ∑(scale multiplier* × fungus multiplier†) branches

***4 category scale estimate**

scale estimate	scale multiplier
0-6	0
7-30	15
31-70	46
>70	150

†5 category fungus estimate

% infection	fungus multiplier
0	0
1-10	0.05
11-20	0.15
21-50	0.35
>50	0.75

Figure IV.2. Protocol for *C. viridis* and *L. lecanii* surveys, adapted from Perfecto and Vandermeer (2006). The coffee plant is assigned to one of the three pathways depending on how many scales are found in an initial count. If more than 50 scales are on the plant, the rightmost path is executed, which entails switching to a branch-by-branch estimate of the number of scales and the abundance of *L. lecanii*. If less than 20 scales are encountered, the leftmost branch is followed, and the total number of infected scales is recorded. Otherwise, an entire-plant estimate of *L. lecanii* prevalence is used to estimate the number of infected scales, as specified by the center path.

For the branch-by-branch protocol, a scale multiplier was assigned to each branch based on an estimate of the number of scales on the branch. At the same time, a fungus multiplier was determined for each branch based on an estimate of the prevalence of *L. lecanii*. The total number of infected scales on the plant was then calculated as the sum over all branches of the product of the scale multiplier and the fungus multiplier (Figure IV.2).

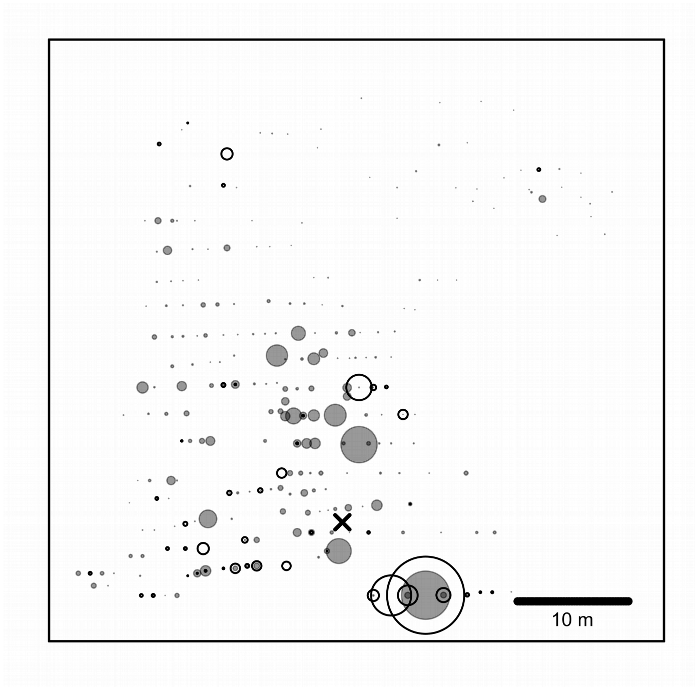
As an approximation of the center of the *L. lecanii* epizootics, we calculated the center of mass of the *L. lecanii* infections using the standard equation for center of mass, i.e., the average of the positions of the coffee plants weighted by the number of infected scales per plant. Due to the temporal and spatial dynamics of the epizootics, the true center of the propagule pressure of *L. lecanii*, which depends both on the influx of propagules into the soil and the subsequent transmission of propagules upwards to the coffee plants, would be very difficult to determine precisely. Therefore, we also analyzed the change in rust abundance as a function of distance to all other points within the plots, i.e., with no a priori assumptions about the locations of the centers of the epizootics.

Hemileia vastatrix surveys were performed in September 2009 and September 2010. Prevalence was defined as the total number of lesioned leaves per plant, and was determined based on an inspection of every leaf of every coffee plant. *Hemileia vastatrix* creates yellow-orange lesions on the underside of leaves that are readily detectable. To reduce the incidence of false positives, only lesions with obvious clusters of orange spores were counted.

Results

In Site A, both *L. lecanii* and *H. vastatrix* were concentrated in the lower half of the plot in the September 2009 survey (Figure IV.3a). In September 2010, the center of the *H. vastatrix* infection had very clearly moved to the upper region of the plot, and the rust was largely absent from the plants that been heavily infected the previous year (Figure IV.3b). There was a marked decrease in the total abundance of *L. lecanii* in Site A, dropping from approximately 1375 infected scales in 2009 to approximately 211 in 2010.

a)



b)

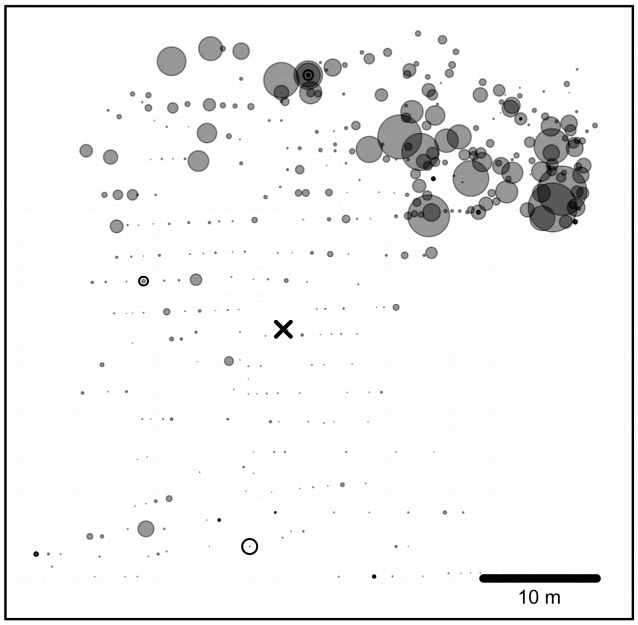
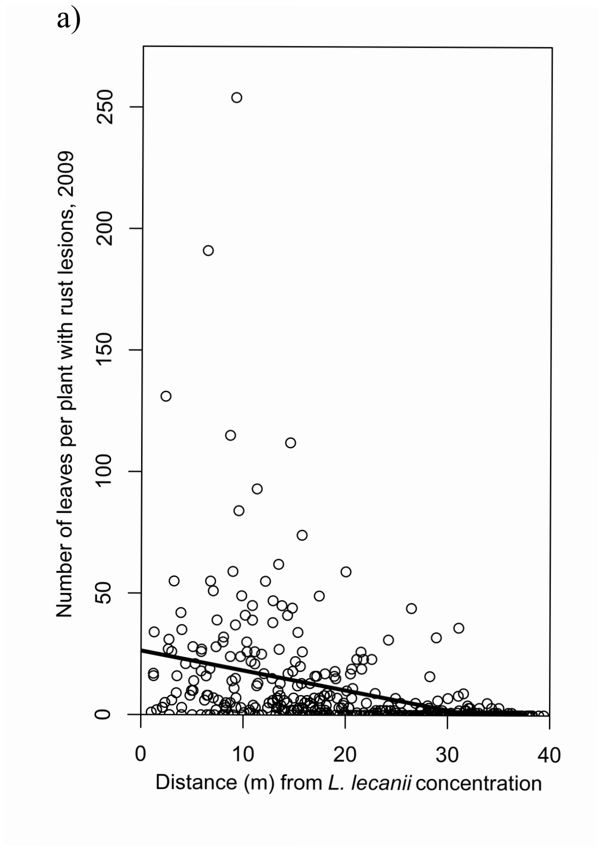


Figure IV.3. Abundance of *L. lecanii* and prevalence of *H. vastatrix* in Site A in a) 2009 and b) 2010. Diameters of open circles are proportional to the estimated number of infected *C. viridis* on coffee plants, with the largest circle corresponding to 308 infected scales. Crosses mark the centers of the *L. lecanii* concentrations. Note that the locations of the centers of the epizootics are influenced by fungal concentrations that are too small to see clearly at this scale. Dark gray circles are proportional to the number of leaves per coffee plant with lesions of *H. vastatrix*, with the largest circles corresponding to 254 lesioned leaves in 2009 and 258 in 2010.

The within-year relationship between the prevalence of *H. vastatrix* and the distance from the center of the *L. lecanii* epizootic in Site A was significantly negative in 2009 (Figure IV.4a, $R^2 = 0.148$, $P < 0.001$), i.e., the prevalence of rust decreased with increasing distance from the center of mass of the mycoparasite. In 2010, the inverse relationship was observed (Figure IV.4b, $R^2 = 0.133$, $P < 0.001$).



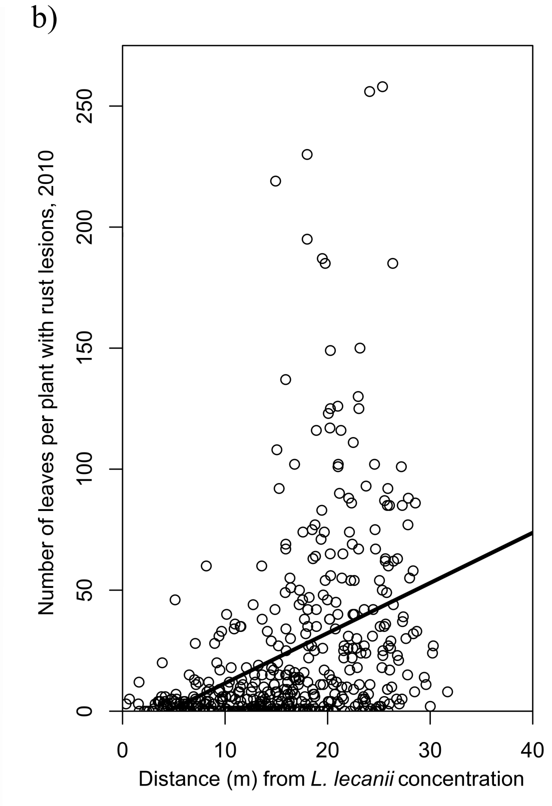
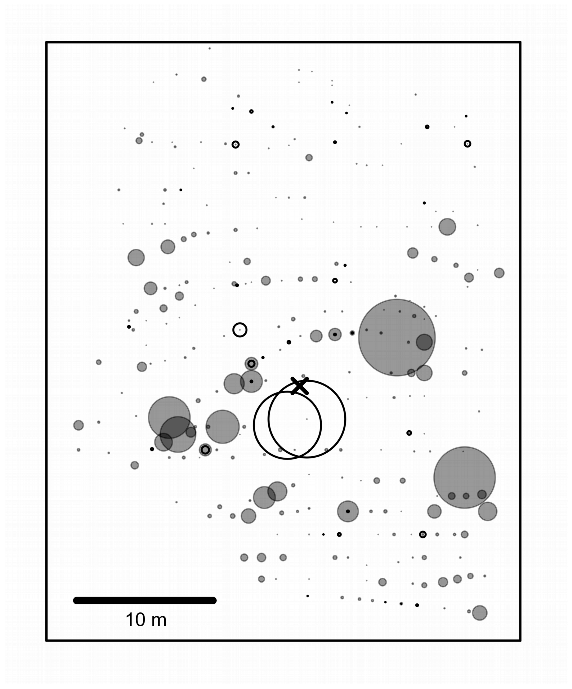


Figure IV.4. Number of leaves per plant with *H. vastatrix* lesions versus the distance to the center of mass of the *L. lecanii* concentration in a) 2009 [$R^2 = 0.148$, $P < 0.001$] and b) 2010 [$R^2 = 0.133$, $P < 0.001$] in Site A.

In Site B, there was also a positive within-year association between the prevalence of *H. vastatrix* and proximity to the *L. lecanii* epizootic in 2009 (Figure IV. 5a), though the amount of variance explained was much less than in Site A (Figure IV.6a, $R^2 = 0.018$, $P = 0.004$). As in Site A, there was a substantial decrease in the abundance of *L. lecanii* from 2009 to 2010, from approximately 1418 infected scales to 146 (Figure IV. 5b). In contrast with Site A, there was no significant relationship between rust prevalence in 2010 and distance from the 2010 *L. lecanii* epizootic (Figure IV.6b, $R^2 = 0$, $P <$ 0.9213).

a)



b)

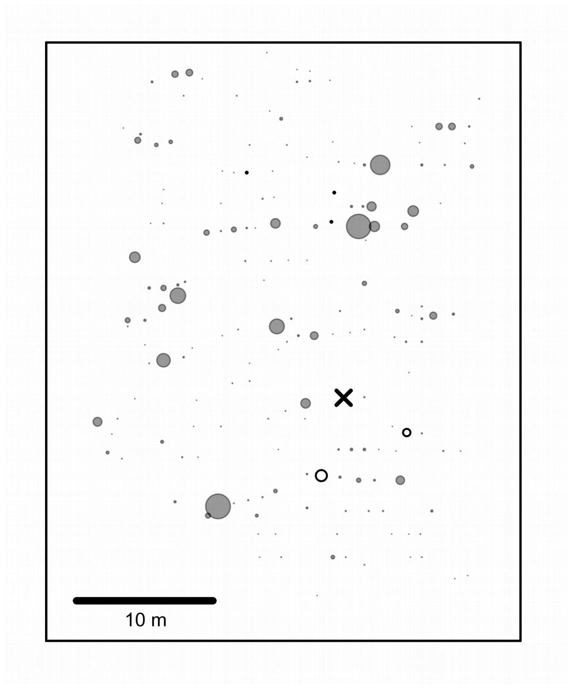


Figure IV.5. Abundance of *L. lecanii* and prevalence of *H. vastatrix* in Site B in a) 2009 and b) 2010. Diameters of open circles are proportional to the estimated number of infected *C. viridis* on coffee plants, with the largest circle corresponding to 468 infected scales. Crosses mark the centers of the *L. lecanii* concentrations. Note that the locations of the centers of the epizootics are influenced by fungal concentrations that are too small to see clearly at this scale. Dark gray circles are proportional to the number of leaves per coffee plant with lesions of *H. vastatrix*, with the largest circles corresponding to 217 lesioned leaves in 2009 and 68 in 2010.

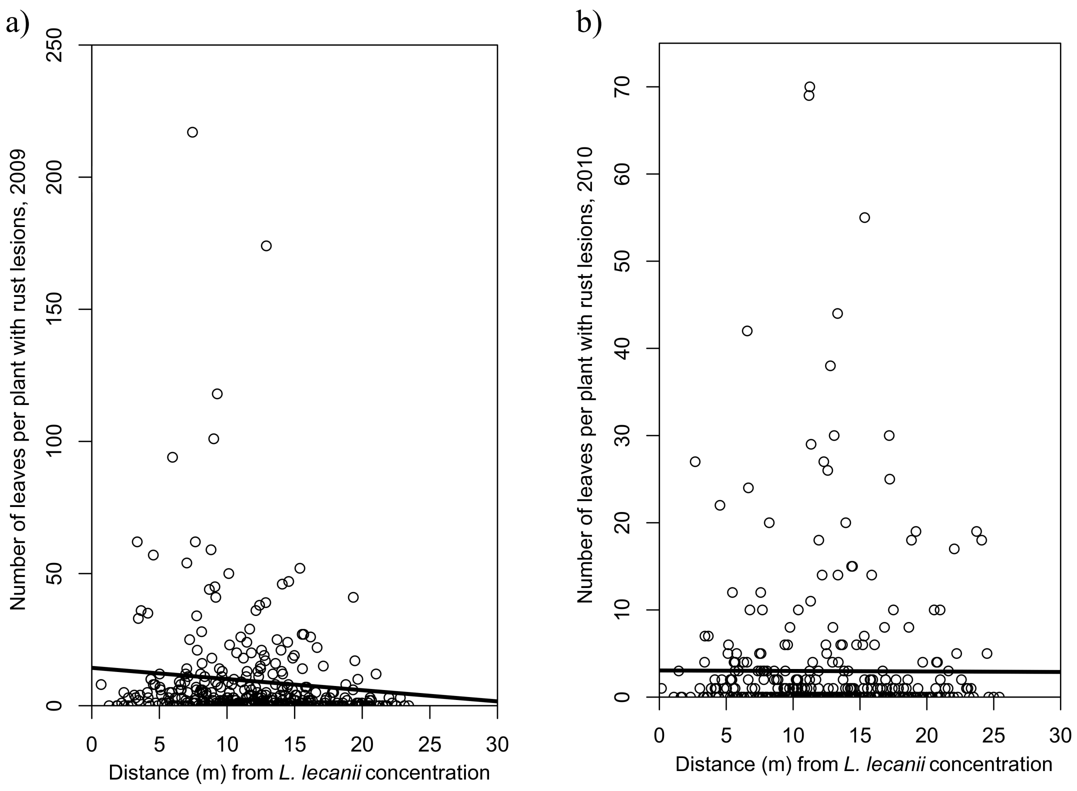


Figure IV.6. Number of leaves per plant with *H. vastatrix* lesions versus the distance to the center of mass of the *L. lecanii* concentration in a) 2009 [$R^2 = 0.018$, $P = 0.004$] and b) 2010 [$R^2 = 0$, $P = 0.921$] in Site B.

Looking at the change in rust from the first year to the second, prevalence decreased substantially in the lower region of Site A and increased markedly in the upper region (Figure IV.7). The linear relationship between distance from the center of mass of the *L. lecanii* infection in 2009 and change in *H. vastatrix* prevalence was significantly positive (Figure IV.8, $R^2 = 0.315$, $P < 0.001$). Performing similar linear regression analyses, but using other points within Site A as points of reference instead of the center of mass, reveals a peak in the R^2 values that corresponds with a qualitative estimate of the location of the 2009 *L. lecanii* epizootic (Figure IV.9a). Likewise, the largest effect sizes (slopes) are obtained by regressing distances relative to points that were near the region with the highest concentrations of *L. lecanii* (Figure IV.9b).

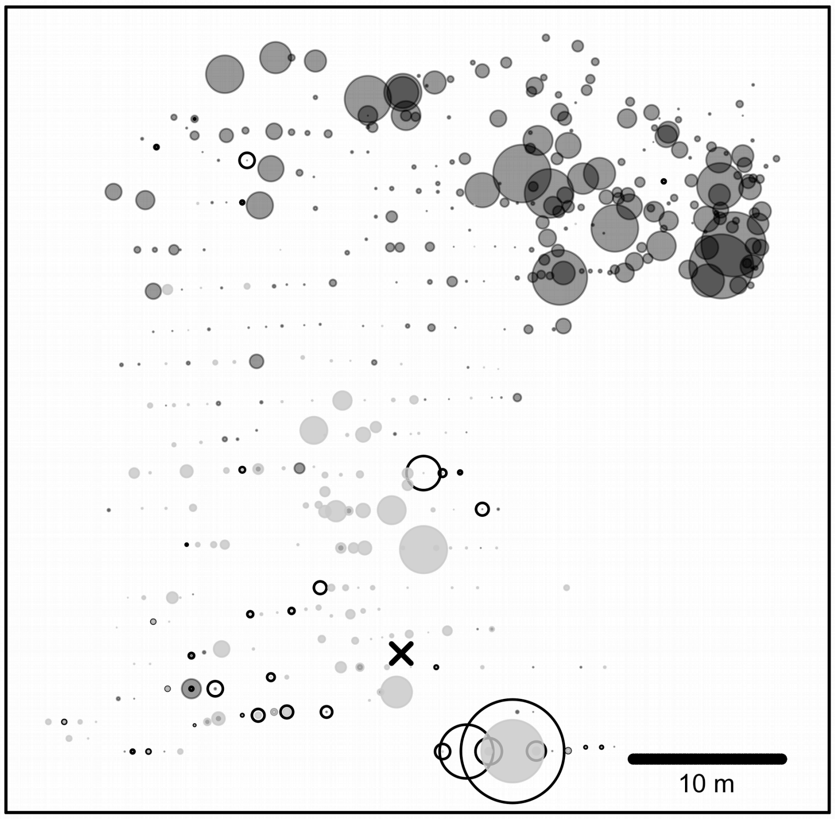


Figure IV.7. Prevalence of *L. lecanii* in 2009 and change in prevalence of *H. vastatrix* from 2009 to 2010 in Site A. Diameters of open circles are proportional to the estimated number of infected *C. viridis* on coffee plants, with the largest circle corresponding to 308 infected scales. Dark gray circles are proportional to the increase in the number of leaves per coffee plant with lesions of *H. vastatrix*. Light gray circles are proportional to the decrease in the number of leaves with *H. vastatrix* lesions. Both the dark gray and light gray circles are scaled to a maximum change of 257 lesioned leaves. Crosses mark the centers of the *L. lecanii* concentrations.

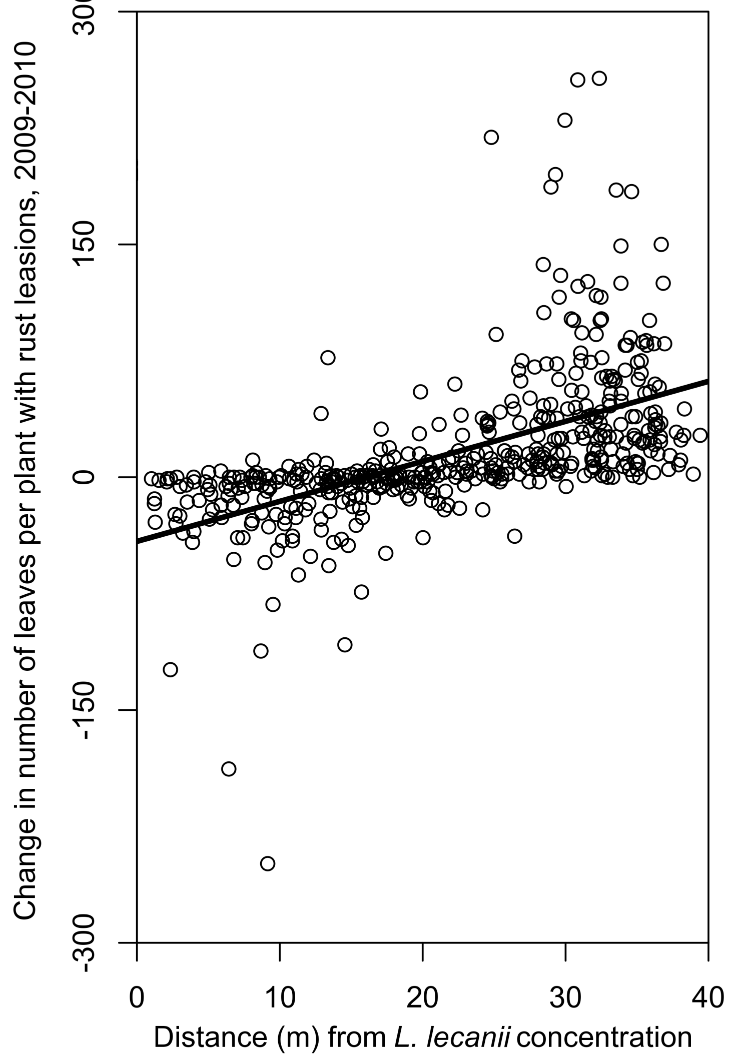
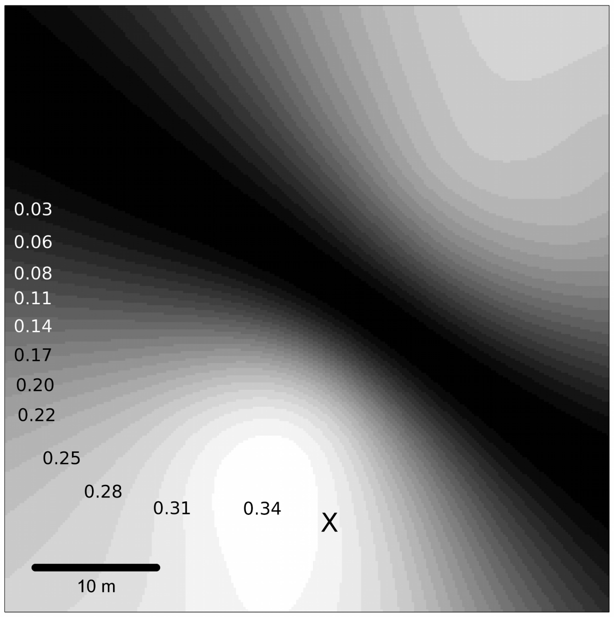


Figure IV.8. Change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to the center of mass of the *L. lecanii* concentration in 2009 in Site A [$R^2 = 0.315$, $P < 0.001$].

a)



b)

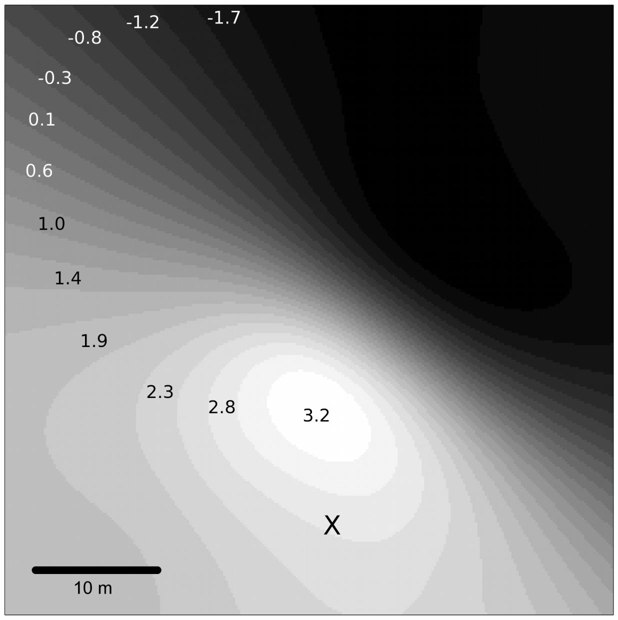


Figure IV.9. a) Coefficients of determination (R2) and b) effect sizes (slopes) for linear regressions of the change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to points within Site A. Crosses mark the center of mass of the *L. lecanii* epizootic in 2009.

In Site B, there was a similar tendency for *H. vastatrix* prevalence to decrease in the neighborhood of the *L. lecanii* epizootic, although it was less pronounced, possibly due to the smaller spatial extent of Site B compared to Site A (Figure IV.10). There was a significant positive relationship between distance to the *L. lecanii* center of mass and change in *H. vastatrix* prevalence, but only a small amount of variance was explained by distance (Figure IV.11, $R^2 = 0.011$, $P = 0.017$). Choosing a point nearer to the x axis (which is physically downslope from the calculated center of mass) as the reference point for the regression instead of the center of mass would increase the R^2 value, although the maximum amount of variance explained is small regardless of the location chosen as the reference point (Figure IV.12a). The maximum effect size is obtained using reference points that are near to the center of mass and the qualitative center of the 2009 epizootic (Figure IV.12b).

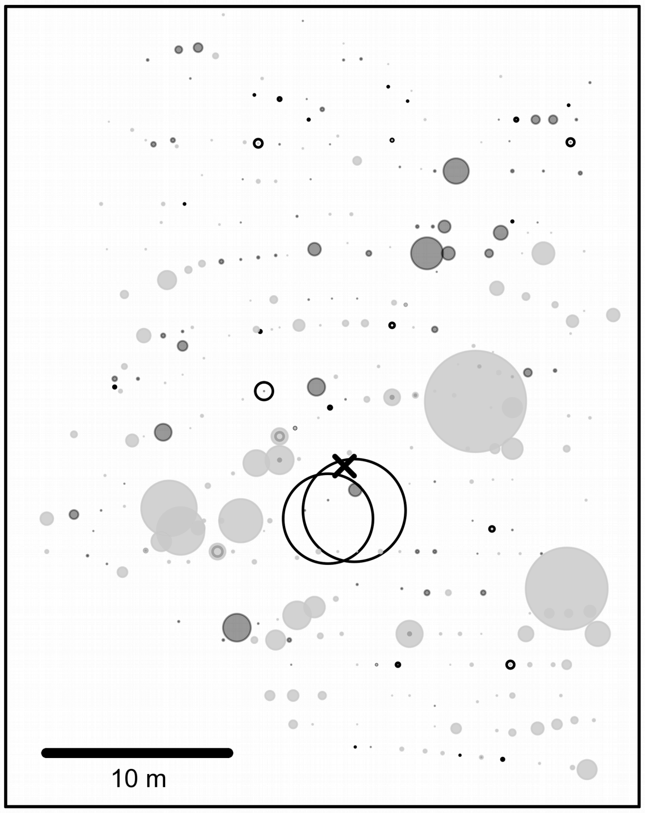


Figure IV.10. Abundance of *L. lecanii* in 2009 and change in prevalence of *H. vastatrix* from 2009 to 2010 in Site B. Diameters of open circles are proportional to the estimated number of infected *C. viridis* on coffee plants, with the largest circle corresponding to 468 infected scales. Dark gray circles are proportional to the increase in the number of leaves per coffee plant with lesions of *H. vastatrix*. Light gray circles are proportional to the decrease in the number of leaves with *H. vastatrix* lesions. Both the dark gray and light gray circles are scaled to a maximum change of 215 lesioned leaves. Crosses mark the centers of the *L. lecanii* concentrations.

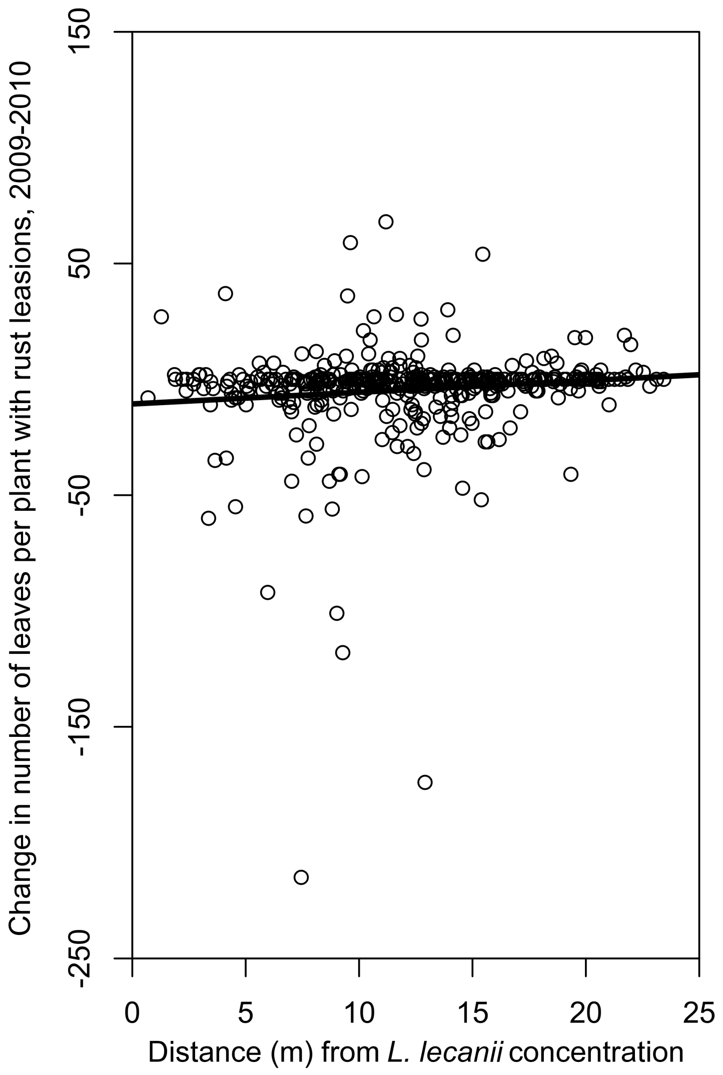
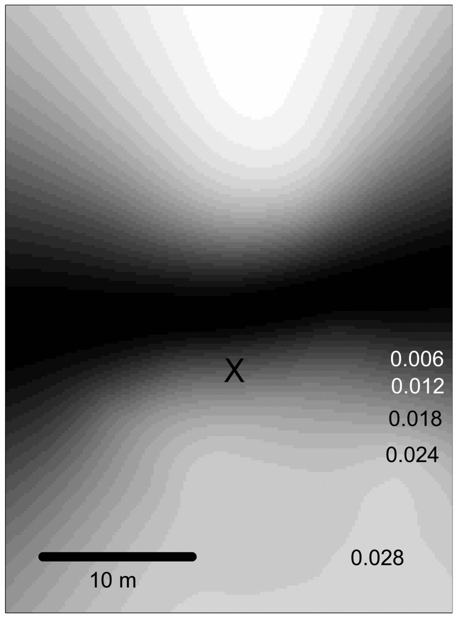


Figure IV.11. Change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to the center of mass of the *L. lecanii* concentration in 2009 in Site B [$R^2 = 0.011$, $P = 0.017$].

a)



b)

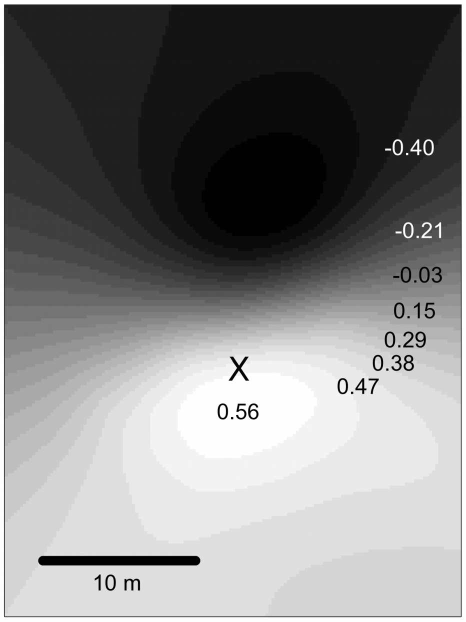


Figure IV.12. a) Coefficients of determination (R^2) and b) effect sizes (slopes) for linear regressions of the change in the number of leaves per plant with *H. vastatrix* lesions between 2009 and 2010 as a function of the distance to points within Site B. Crosses mark the center of mass of the *L. lecanii* epizootic in 2009.

Discussion

These results add to the accumulating evidence that *L. lecanii* can have an ecologically significant, controlling effect on *H. vastatrix* (Avelino et al. 2004, Vandermeer et al. 2009). And, importantly, in combination with the findings of Vandermeer et al. (2009), our results suggest that there is a time lag in the effect of *L. lecanii* on *H. vastatrix* that had not been previously recognized. This time lag, in which a large abundance of *L. lecanii* in one year suppresses *H. vastatrix* in the following year, is consistent with the known biology of the two fungi, and is also concordant with the observed variation in the within-year relationship between the fungi.

If the prevalence of *H. vastatrix* is more strongly affected by the abundance of *L. lecanii* in the previous year, the within-year relationship between *H. vastatrix* prevalence and *L. lecanii* abundance could be either negative (as in our 2010 data and the data reported by Vandermeer et al.) or positive (as in our 2009 data) depending on whether the location of the *L. lecanii* epizootic had remained relatively constant or had shifted from one year to the next. For example, given the very low abundance of *L. lecanii* in Site A in 2010 (Figure IV.3b), the apparent negative relationship between *H. vastatrix* and *L. lecanii* in 2010 (Figure IV.4b) is most likely not a result of the 2010 *L. lecanii* concentration. Rather, it is likely an artifact of the suppressive effect of the 2009 *L. lecanii* epizootic and the relative proximity of the center of masses of the 2009 and 2010 *L. lecanii* concentrations; had the center of the 2010 *L. lecanii* concentration shifted further to the upper right of the plot, the apparent negative relationship would have been positive instead.

This time-lag effect of *L. lecanii* was previously unrecognized because, to our knowledge, this is the first study to focus primarily on a comparison of *H. vastatrix* and *L. lecanii* abundances across multiple years. Although Vandermeer et al. (2009) did follow one site for two years, censuses in subsequent years revealed that the *A. instabilis* nest at the particular site that was available during their study period was in the process of dying, resulting in a decreased abundance of *C. viridis* and hence fewer scales infected by *L. lecanii* (Vandermeer, unpublished data). The weakened state of the ant nest, coupled with the less-extensive line transect survey method used in their study, likely account for the absence of a multi-year effect in their results.

As noted previously by Vandermeer et al. (2009), the controlling effect of *L. lecanii* on *H. vastatrix* under field conditions appears to be subtle. Proximity to the *L. lecanii* epizootic accounted for only a fraction of the variance in Site A ($R^2 = 0.315$). In Site B, the explanatory power of distance from the previous year's epizootic can be increased by assuming that the center of the epizootic was further downslope than the center of mass would indicate, which is a reasonable assumption considering the probable tendency for gravity to shift the dispersal of *L. lecanii* propagules downslope. However, even this assumption achieves only a small absolute improvement in the amount of variance explained, from $R^2 = 0.011$ to $R^2 = 0.028$.

In addition, as suggested by the large year-over-year decrease in the numbers of infected scales in both sites, as well as the relative differences in the effects in the two sites, the magnitude of the effect of *L. lecanii* on *H. vastatrix* will likely vary significantly over space and time. This variation is likely driven in part by the internal dynamics of the pathogen-host-mutualist system. The mutualist ant, *A. instabilis*, has been shown to significantly reduce its tending activity in response to experimentally-induced epizootics of *L. lecanii* (Andrew MacDonald, Doug Jackson, and Kate Zemenick, unpublished data). This suggests that epizootics of *L. lecanii* may decrease the amount of food available to an affected *A. instabilis* colony, which may weaken the colony and consequently diminish its effectiveness as a mutualist of *C. viridis*. This, in turn, could lead to a decrease in the size of the scale population the following year. Although this scenario could explain the decreases in *C. viridis* (both healthy and infected) observed in our study sites, further study would be necessary to demonstrate that this cascading effect in fact occurs.

Despite the apparent subtlety and variability of the controlling effect of *L. lecanii*, its regulatory effect on *H. vastatrix* may be substantial. The magnitude of the role the *L. lecanii* may play in preventing outbreaks of *H. vastatrix* depends on the details of the population dynamics of a complex web of interactions between multiple species, and these interactions themselves likely vary substantially over space and time. Therefore, while evidence is accumulating that *L. lecanii* does have a negative effect on the prevalence of *H. vastatrix* under field conditions, and that this effect can be detected, quantitatively assessing this effect will require further research into the dynamical interactions that characterize this complex system. Given its widespread distribution throughout this coffee farm, however, there is a high potential for *L. lecanii* to play an important regulatory role.

While the observations of Vandermeer et al. (2009) and the known mycoparasite-host relationship between *L. lecanii* and *H. vastatrix* strongly suggest that *L. lecanii* is a significant driver of the observed shift in *H. vastatrix* prevalence, the difficulty involved with directly quantifying the infection process of natural fungal populations under field conditions (e.g., see Eskes et al., 1991) leaves some equivocality. The development of coffee rust epidemics is known to be affected by a number of biotic and abiotic factors, including soil acidity, coffee yield, temperature, humidity, fertilization, and altitude (Avelino et al. 2006). *Lecanicillium lecanii* epizootics are also influenced by environmental factors, such as temperature and relative humidity (Reddy and Bhat 1989); and shade (Easwaramoorthy and Jayaraj 1977). However, none of these known influences seem likely to account for the observed shift in *H. vastatrix* prevalence relative to the concentrations of *L. lecanii*. Shade cover was not differentially altered within the sites, and there were no other known changes that would have affected the microclimate in a way that would have resulted in such a systematic shift. Likewise, the remainder of prominent factors are unlikely to have varied significantly on such a local scale, or to have varied at all. It is possible that there is some unknown force that affects both *L. lecanii* and *H. vastatrix*, thus leading to these results, but the most parsimonious explanation at present is that the observed pattern is a consequence of the pathogen-host relationship between the two fungi.

It is important to note that the hyperparasitic effect of *L. lecanii* occurs only because of the spatial structure of the ecosystem as a whole. That is, the mutualistic effect of the ant (*A. instabilis*) on the main host of *L. lecanii* is contained within distinct pockets of unusually high concentrations of that host (*C. viridis*), and those distinct pockets are so-called self-organized patches (Vandermeer et al. 2008). Thus, this conservation biological control includes other elements in the ecosystem as a whole acting in a spatially specific context (Liere and Perfecto 2008, Jackson et al. 2009, Vandermeer et al. 2010a). Consequently, the success of efforts to further enhance the control of *H. vastatrix* in similar coffee agroecosystems could depend on an understanding of that larger complex ecosystem, especially of what influences the spatial distribution of *A. instabilis* colonies. By studying the effects of naturally occurring concentrations of *L. lecanii*, maintained by virtue of a complex ecological network, we can learn to more effectively capitalize on the ecosystem services provided by this biological control agent; begin to predict how this autonomous biological control may respond to climate change; and suggest management strategies to maintain control. Restricting attention to the abiotic factors that are typically considered to affect coffee rust may thus not be wise.

CHAPTER V

The evolution of imperfect prudence

Survival of the fittest. The invisible hand. The wisdom of crowds. Self-organized criticality. The observation that unexpected – and often desirable – properties at the macro scale can arise spontaneously from endogenous interactions at smaller scales has captured the collective imagination to a degree that few other ideas within the last 300 years have. Modern *Homo sapiens*, enchanted by the insights symbolized by the likes of Charles Darwin and Adam Smith, conceive of a world filled with wonderfully complex organisms that have been crafted by the blind forces of natural selection; economies that propel themselves forward through the self-correcting push of market forces; and democracies that integrate the perspectives of individual citizens into a collective wisdom that exceeds that of any Solomon.

Inspired by the apparent power of these autogenous processes, scientists in fields as distinct as political science and biology continue to push the boundaries of our understanding of evolution and self regulation, demonstrating the potential of these processes to act in ways that far exceed, in breadth, diversity, and subtlety, what their original proponents could have possibly imagined. Prominent amongst these more recent elaborations of the basic principles is evidence that evolution, an indifferent actor unencumbered by human concepts of kindness or morality, can, counterintuitively, give rise to cooperative behaviors. That is, despite competition for limited resources being the putative force underlying evolution, evolution sometimes favors altruistic behaviors over those that are purely selfish.

Although the fact that cooperation can evolve is readily apparent from even a superficial acquaintance with nature, explaining how a process predicated on selfishness, i.e., evolution by natural selection, can give rise to cooperation is far from trivial. Any potential theory must explain how cooperators can resist invasion by cheaters: individuals that would exploit the benefits that cooperators provide while failing to reciprocate, thereby avoiding the costs of cooperation while enjoying the benefits. A number of processes that could lead to the evolution of cooperation have been identified, with one of the earliest being the concept of viscous populations, first proposed by Hamilton (1964b). Highly viscous populations are characterized by limited dispersal, which increases the frequency of repeated interactions between individuals and interactions between closely related individuals, both of which promote cooperation. Since Hamilton, the potential for the evolution of cooperation in viscous populations has been demonstrated by a large number of theoretical and computational studies (Lion and Baalen 2008). However, it was not until recently that experimental evidence in biological systems was obtained (Kerr et al. 2006, Boots and Mealor 2007, Szilágyi et al. 2009).

These experimental systems, and the majority of theoretical work done to date on the evolution of victim-exploiter systems, focus on the evolution of the exploiter. In the context of host-pathogen systems, where the host is the victim and the pathogen is the exploiter, this emphasis on the evolution of the exploiter would seem to be reasonable: pathogens typically have much shorter generation times, and hence are likely to evolve at much faster rates than their hosts. However, there are exceptions to this generalization. Resistance of the host plant *Lychnis alpina* to the anther smut fungus *Microbotryum violaceum* has been shown to be correlated with local characteristics of *L. alpina* spatial distribution, which suggests that host evolution in response to local changes in host-population connectivity is the dominant evolutionary process in this system (Carlsson-Graner and Thrall 2002). Duffy and Sivars-Becker (2007) showed that the termination of epidemics of the parasite *Metschnikowia bicuspidata* can be explained by rapid evolution of the susceptibility of its host, *Daphnia dentifera*. These exceptions suggest that by focusing exclusively on the evolution of the exploiter, we may be missing potentially important phenomena.

As with the nearly exclusive focus on the evolution of the exploiter, the tendency to compare the performance of evolved behaviors to purely selfish behaviors may also limit our understanding – in this case by leading us to subconsciously overestimate the effectiveness of evolution. For example, Kerr et al. (2006) showed that the evolution of pathogens in a spatially restricted (viscous) population resulted in competitive restraint that averted the "tragedy of the commons." When migration of bacteriophage in a metapopulation was restricted to a local neighborhood, prudent phages outcompeted rapacious phages, while the opposite was true when migration was unrestricted. Rapacious phages tended to over-exploit the common resource, thereby lowering overall productivity, whereas dominance by prudent phages resulted in higher productivity, thereby averting the tragedy of the commons. That is, the performance of the phage population, in terms of productivity, was improved by the evolution of cooperation. What was not considered, however, is how well the evolved population performed compared to the best possible performance. If we were able to prescribe a different level of prudence, could we increase the performance of the phage population even more? Did the phages evolve to the optimal level of prudence, or was the prudent phages' productivity good only in comparison to the poor performance of the purely selfish, highly rapacious phage?

The use of purely selfish behavior as the null expectation is, in some sense, a natural choice. In general, theoretical models predict that maximum selfishness will evolve in non-viscous populations (Hamilton 1964a). For example, the mean field expectation for the evolution of transmissibility in host-pathogen models (in the absence of other tradeoffs) is maximum transmissibility (Rand et al. 1995). So, it is interesting and surprising to show that cooperation can evolve in the form of decreased transmissibility relative to this mean field expectation. However, to properly gauge the performance of the evolved population, we would need to compare the performance of the evolved transmissibility to both the worst-case and best-case transmissibilities.

In the present study, I consider these two relatively unexplored aspects of the evolution of cooperation – the evolution of the victim and the performance of the evolved population relative to an optimal strategy – using a spatially-explicit host-pathogen model. In this model, a locally-dispersed host is subject to attack by a locally-dispersed pathogen. The spatial distribution of the host emerges as a consequence of reproduction of the hosts into empty sites in their local environment coupled with pathogen-induced mortality and a fixed background mortality rate. Depending on the hosts' reproduction rates, their spatial distribution will be characterized by either a well-connected network of large clusters (for high reproduction rates) or a poorly-connected landscape of smaller, isolated clumps (for low reproduction rates).

A well-connected landscape of large clusters will be more susceptible to large epidemics, as the pathogen will be able to percolate through the landscape of connected clusters, while a landscape of smaller, isolated clusters will be more resistant to the spread of the pathogen. This scenario creates a conflict between what is good for an individual host in the short term – rapid reproduction – and what is good for the host population as a whole in the long term – a poorly connected landscape generated as a consequence of slower reproduction rates. The question, then, is whether it is possible for cooperation, in the form of reduced host reproduction rates, to evolve. And, if this form of prudence on the part of the hosts is able to evolve, how will the cooperating host population perform, in terms of metrics such as population size and variability, compared to the extremes of pure selfishness and optimal cooperation?

The model

The model is a discrete time, probabilistic cellular automata on a square lattice with periodic boundary conditions (Appendix A). Each cell in the lattice can be in one of three states: empty, occupied by a susceptible host, or occupied by an infected host. In each time step, either the pathogens will execute their actions, if there are pathogens present, or the hosts will execute their actions. This results in the pathogen life cycle being effectively instantaneous compared to the host life cycle; the hosts reproduce and die of natural causes as long as there are no pathogens present, but once a pathogen infects a single host, host activity is frozen while the pathogens sweep through the host population. Host activity resumes only after the epidemic runs its course and the last pathogen dies.

Host activity includes reproduction, death by natural causes, and pathogen-induced mortality. Reproduction of a susceptible (healthy) host, *i*, into an empty cell in its von Neumann neighborhood (its four nearest neighbors) occurs with probability g_i. Each reproduction attempt is an independent event, meaning that a host surrounded by four empty cells can produce up to four offspring in a single time step. If multiple hosts attempt to reproduce into a single cell, the winner is chosen randomly. Infected hosts do not reproduce. Death by natural causes occurs with a fixed probability, *m*. Pathogen-induced morality is determined by the pathogen virulence*, v*. In the current study, virulence is fixed at a probability of 1, meaning that hosts only live for one time step after being infected.

Pathogen activity begins with an initial infection event that occurs with probability *l*. The initial infection targets a randomly chosen host. The pathogen subsequently spreads to neighboring hosts via reproduction, or transmission. Pathogens transmit to susceptible hosts in their von Neumann neighborhoods with probability τ. In the current implementation, τ is fixed at 1 for all pathogens. Collisions, in which multiple pathogens attempt to infect a single host, are resolved by choosing a winner at random. As with host reproduction, transmission is determined independently for all of an infected host's susceptible neighbors, so an infected host with *n* susceptible neighbors can infect between 0 and *n* individuals.

Evolution occurs during host reproduction. When host *i* reproduces, its offspring normally inherit its reproduction probability, g_i. However, mutations of $\pm\,\epsilon$ occur with probability μ. Therefore, the reproduction probability of offspring *j* of host *i* is defined as follows:

$$P(g_j = g_i) = 1 - \mu \tag{1}$$

$$P(g_j = g_i + \epsilon) = \mu/2 \tag{2}$$

$$P(g_j = g_i - \epsilon) = \mu/2 \tag{3}$$

The default parameter values used for all simulations, unless otherwise noted, are shown in Table V.1.

Parameter	Description	Default
g_i	reproduction probability of host *i*	variable
m	baseline (natural) mortality rate	0.2
l	probability of spontaneous infection	0.0016
v	virulence: mortality probability of infected host	1
τ	probability of transmission to susceptible neighbor	1
μ	probability of mutation of g_i	0.15
ϵ	magnitude of mutation of g_i	0.01
X	width of lattice (cells)	100
Y	height of lattice (cells)	100
N_0	initial host population size	500
P_0	initial number of pathogens	50

Table V.1. Default parameter values.

Under this framework, the host spatial distribution emerges due to the interaction between the hosts' reproduction probabilities, g_i, the background mortality rate, m, and the intermittent removal of hosts by epidemics. Epidemics occur at random and then spread through the host population. If the hosts are distributed in a well-connected landscape, the pathogen will sweep through a large portion of the host population. If the hosts are less well connected, the epidemic will be constrained to a smaller portion of the host population, and each epidemic will have less of an impact on the hosts' spatial distribution. Therefore, the hosts and pathogens simultaneously drive and are driven by the spatial structure of the system.

Results

With the parameters set to the default values shown in Table V.1 and all hosts initialized with the same reproduction probability, the host population consistently evolves to an intermediate average reproduction probability of approximately 0.2 (Figure V.1). This demonstrates that hosts exhibiting reproductive restraint, i.e., prudent hosts, can evolve. Furthermore, the system is driven to prudence whether the hosts are initialized with reproduction probabilities above or below the equilibrium value of 0.2, as demonstrated by the representative runs shown in Figure V.1. There is a basin of attraction that extends from an initial host reproduction probability of 0 to approximately 0.65; above this range, the hosts have a greater than 50% probability of evolving towards increasing reproduction probabilities, which causes the host population to form ever larger and more well-connected clusters that are inevitably subject to a catastrophic epidemic that extinguishes the entire host population, a phenomenon termed "evolutionary suicide" (Lion and Baalen 2008).

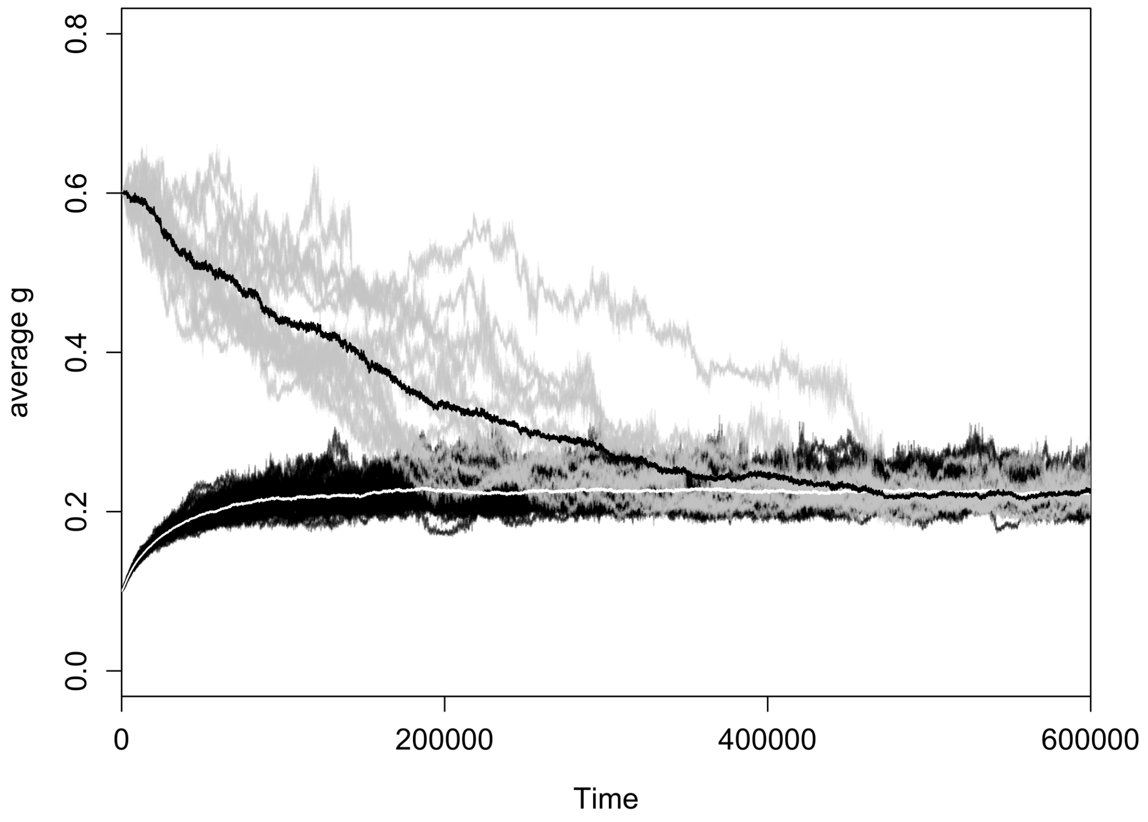


Figure V.1. Evolutionary dynamics of 100 representative realizations of the model initialized above and below the ESS host reproduction probability, *g*, of approximately 0.20. Gray lines show the average reproduction probabilities for the host populations of 50 representative runs with all hosts initialized with $g = 0.6$; the upper black line shows the average of these 50 realizations. The lower black lines show the results of 50 realizations with all hosts initialized with $g = 0.1$; the white line is the average of these 50 runs. To reduce the variance in *g*, μ was reduced to 0.02.

A pairwise invasibility plot (PIP) of the host reproduction probabilities reveals that $g \approx 0.2$ is an evolutionarily stable strategy (ESS), meaning that this strategy cannot be invaded by any competing strategy (Figure V.2). The PIP was generated by initializing the model with a fixed host reproduction probability, termed the resident strategy (g_R). The model was then run for 100 time steps with evolution disabled ($\mu = 0$), which was previously determined to be a sufficient amount of time for the model to reach an equilibrial state. At this time, 10 individual hosts with a different reproduction probability, termed the invader strategy (g_I), were placed randomly in the arena. After a total of 100,000 time steps, the strategy comprising the majority of the host population was designated as the winning strategy. A resident strategy of $g_R = 0.2$ cannot be invaded by any other strategy. Resident strategies below 0.2 are able to be invaded by some more rapidly reproducing hosts, while resident strategies above 0.2 can be invaded by a range of more prudent hosts.

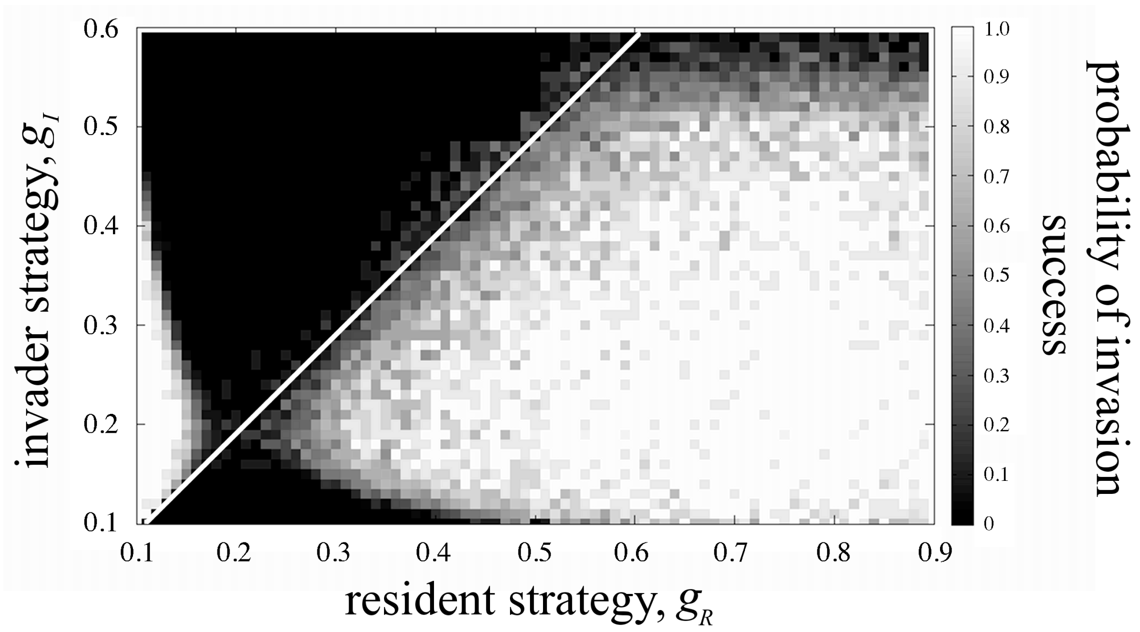


Figure V.2. Pairwise invasibility plot showing the probability that an invading strategy with a host reproduction probability g_I can beat a resident strategy g_R. For each run, all hosts were initialized with the resident strategy. After 100 time steps, 10 individuals with the invader strategy, g_I, were placed at random locations. After 100,000 time steps, the strategy represented by the majority of individuals was deemed the winning strategy. The white line is the 45 degree line, where the resident and invader strategies are equal. The probability of a successful invasion is shown by the grayscale spectrum, with lighter colors indicating a higher probability that the invading strategy will outcompete the resident strategy.

The mechanisms underlying the evolution of prudent hosts can be understood by examining the relationship between the host reproduction probability and the spatial structure of the host population (Figure V.3). In the presence of the pathogen, the cluster sizes and connectedness of the host population increases with g. Consequently, host mortality per epidemic also increases with g. Therefore, the hosts' life expectancy is negatively correlated with reproduction probability, giving individuals with a lower reproduction probability a longevity advantage (Figure V.4).

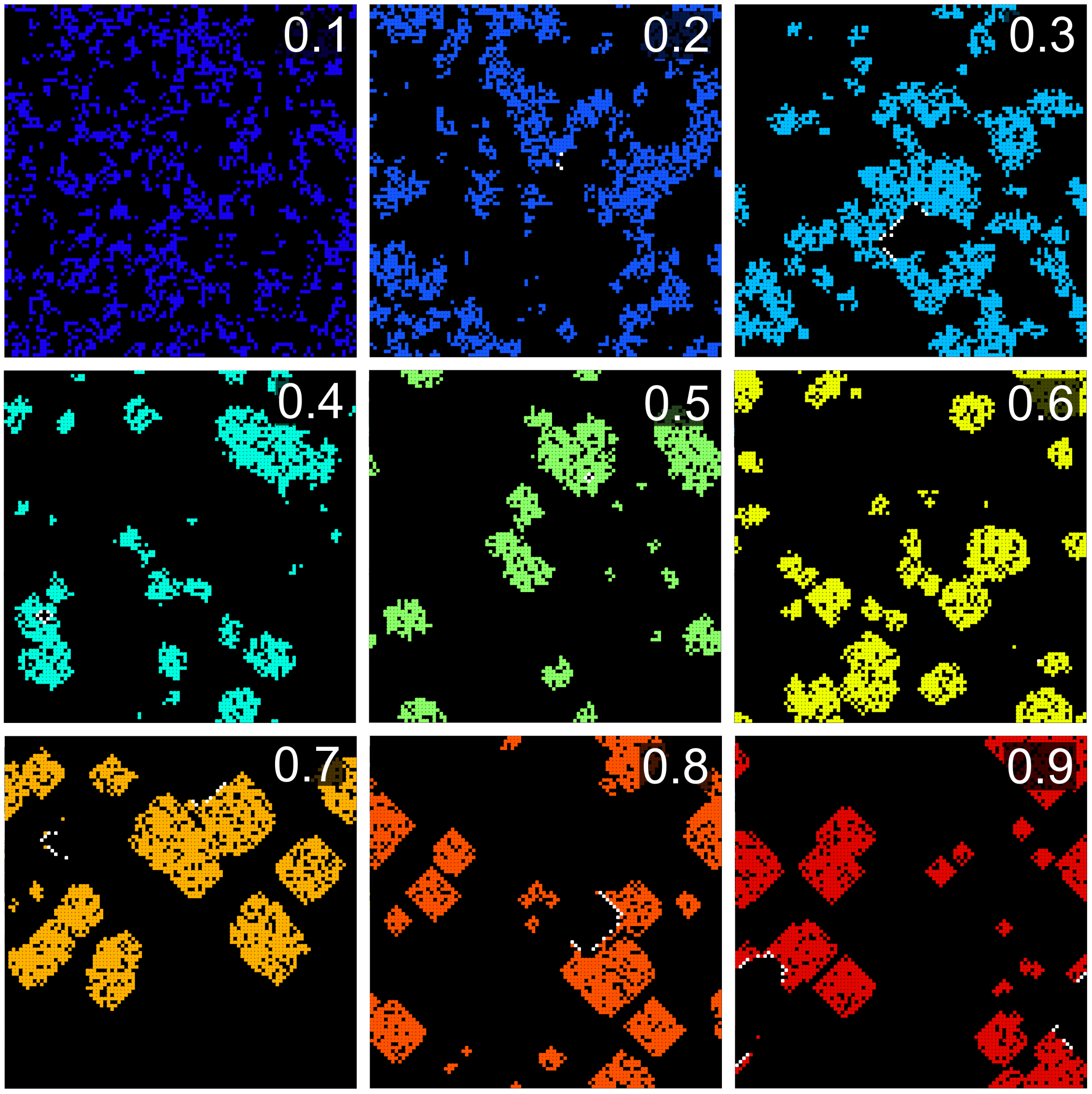


Figure V.3. The spatial structure of the host population for various host reproduction probabilities, *g* (shown in the upper righthand corner of each square). Black regions are empty cells. White cells are infected hosts. Colored cells are uninfected (susceptible) hosts, with the colors of the hosts indicating their reproduction probabilities.

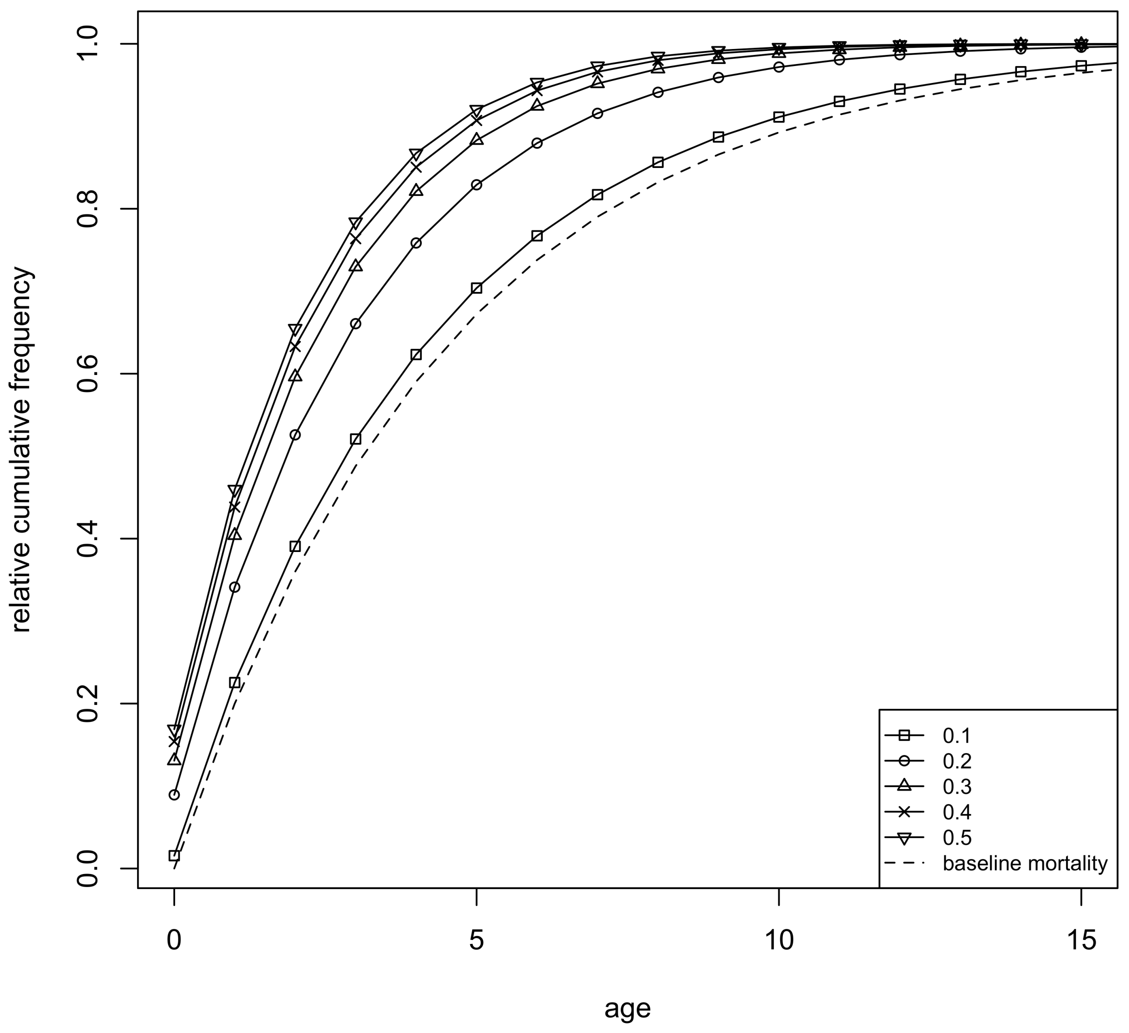


Figure V.4. Relative cumulative frequency of hosts of a given age at death for various host reproduction probabilities (*g*). Averages of 50 realizations for each value of *g* are shown. The dashed line is the expected distribution without the pathogen. As *g* increases, individuals tend to die younger.

Given that decreased reproduction probabilities confer a longevity advantage, why does the host population not simply evolve to the lowest possible reproduction probability, i.e., a reproduction probability that is simply sufficient to offset the background mortality rate? The answer lies in the other component of life history: fecundity. As would be expected, hosts with higher reproduction probabilities have a fecundity advantage (Figure V.5), suggesting that the ESS host reproduction probability is the result of a fecundity-longevity tradeoff.

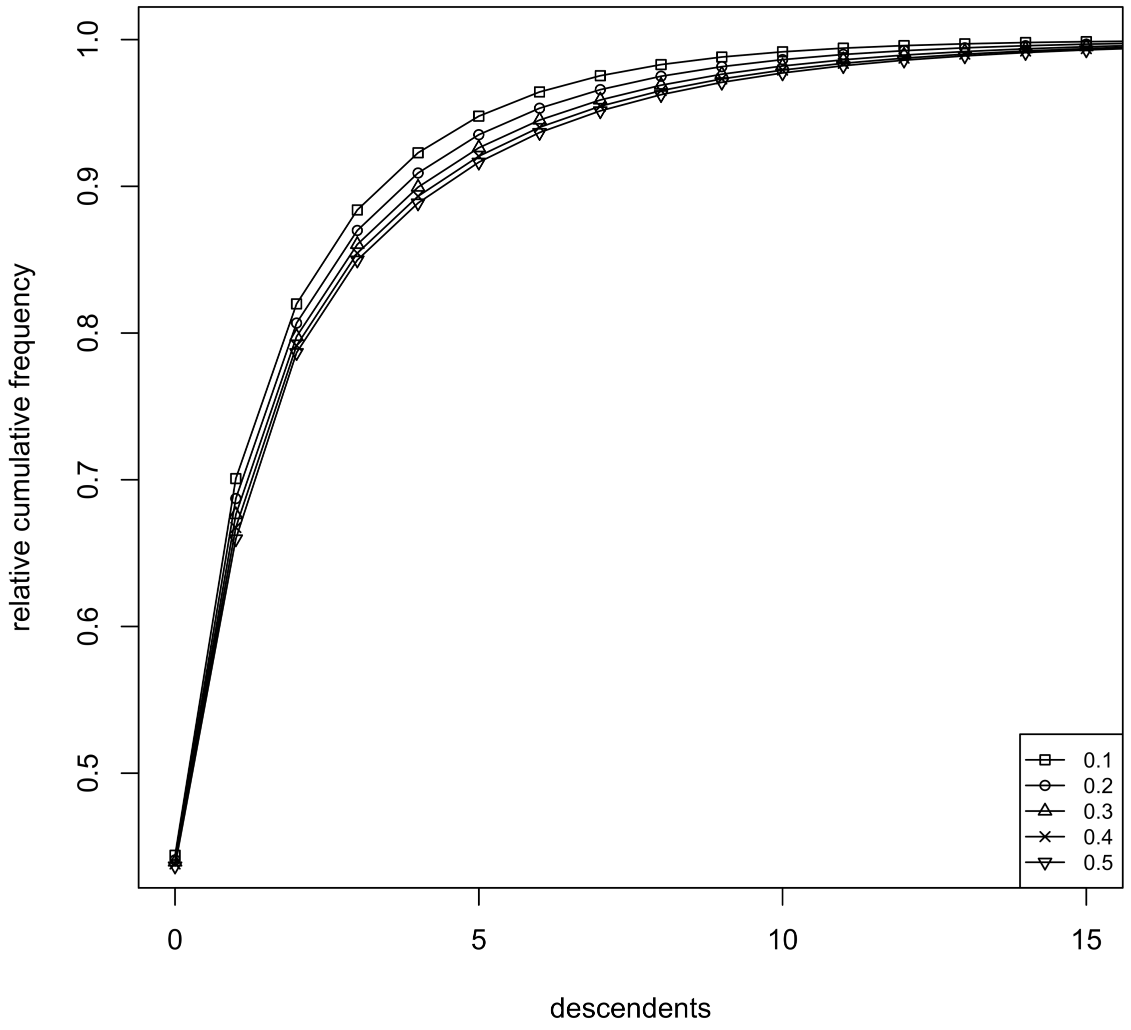


Figure V.5. Relative cumulative frequency of the total number of descendants per host upon death for various host reproduction probabilities (*g*). Averages of 50 realizations for each value of *g* are shown. As *g* increases, the number of descendants per individual increases.

When evolution is enabled, of course, the host population will not have a single, uniform reproduction probability, but rather the population will consist of a mosaic of different reproduction probabilities (Figure V.6). The fecundity and longevity characteristics calculated for the homogenous case will be modified by the interactions between hosts with different reproduction probabilities. For example, due to space competition, the growth of a cluster of rapidly reproducing hosts will be constrained by more slowly reproducing hosts (Figure V.7).

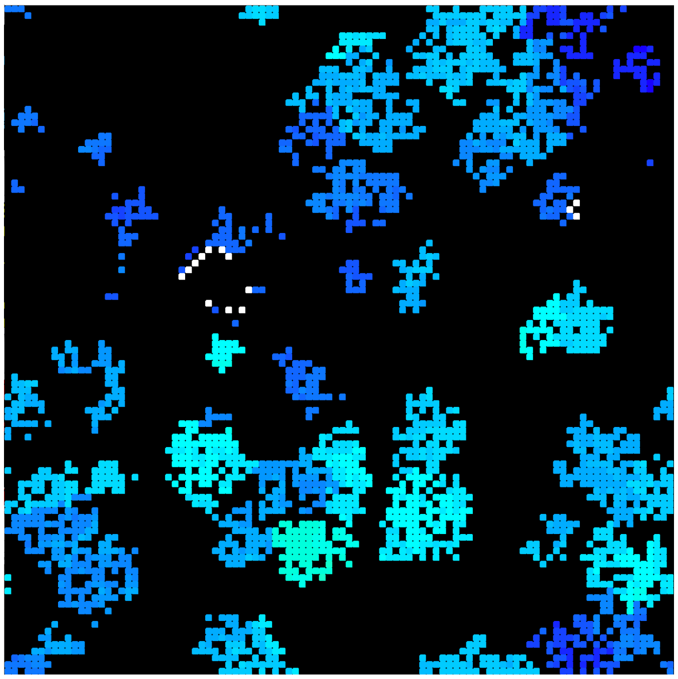


Figure V.6. Representative snapshot of the model after the evolutionary equilibrium has been achieved. Black regions are empty cells. White cells are infected hosts. Colored cells are uninfected (susceptible) hosts, with the colors of the hosts indicating their reproduction probabilities (see Figure V.3 for the growth rates that correspond to the colors).

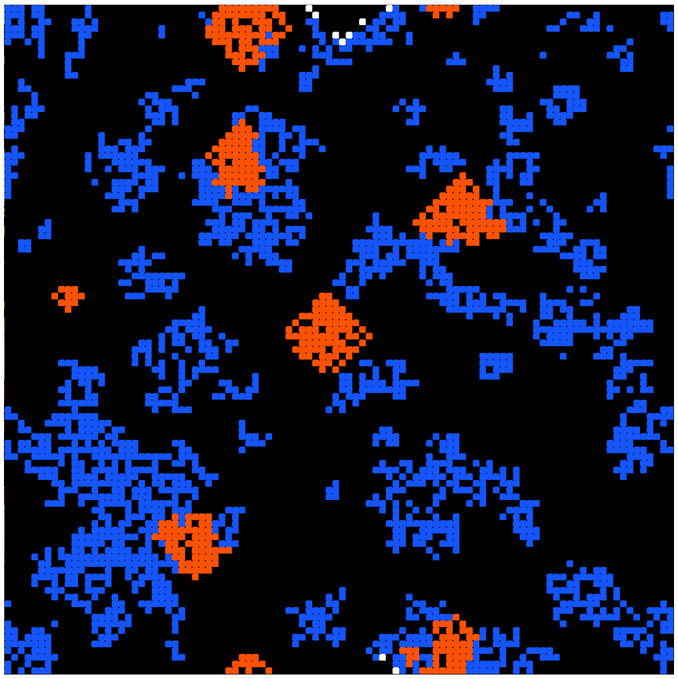


Figure V.7. Illustrative example of the growth of clusters of rapidly-reproducing hosts (orange cells, g = 0.8) being constrained by more slowly reproducing hosts (blue cells, g = 0.2). Black regions are empty cells. White cells are infected hosts.

Although a comprehensive analysis of how the baseline fecundity and longevity relationships shown in Figures V.4 and V.5 are changed in the context of the mosaic of different host strains is too complicated to cover here, a qualitative sense of these changes can be obtained by looking at how these properties are changed for challengers attempting to invade a resident population with $g = 0.2$ (Figures V.8 and V.9).

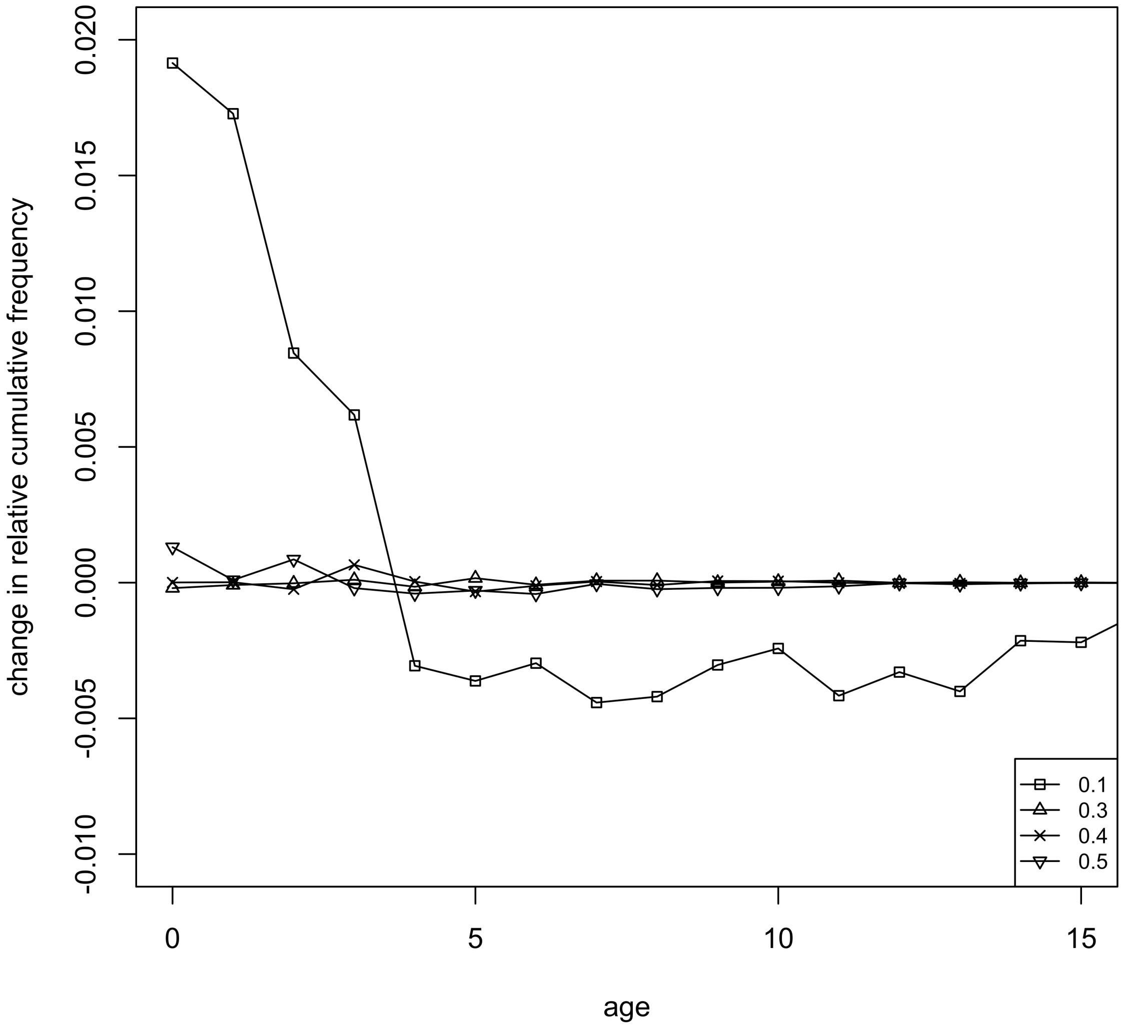


Figure V.8. Change in the relative cumulative frequency of hosts of a given age at death for various host reproduction probabilities (g) attempting to invade a resident host population with $g = 0.2$. Changes are relative to the distributions reported for the homogenous scenarios (Figure V.4). In the context of the resident population, invading individuals tend to die at a younger age than they would in a population of their own kind.

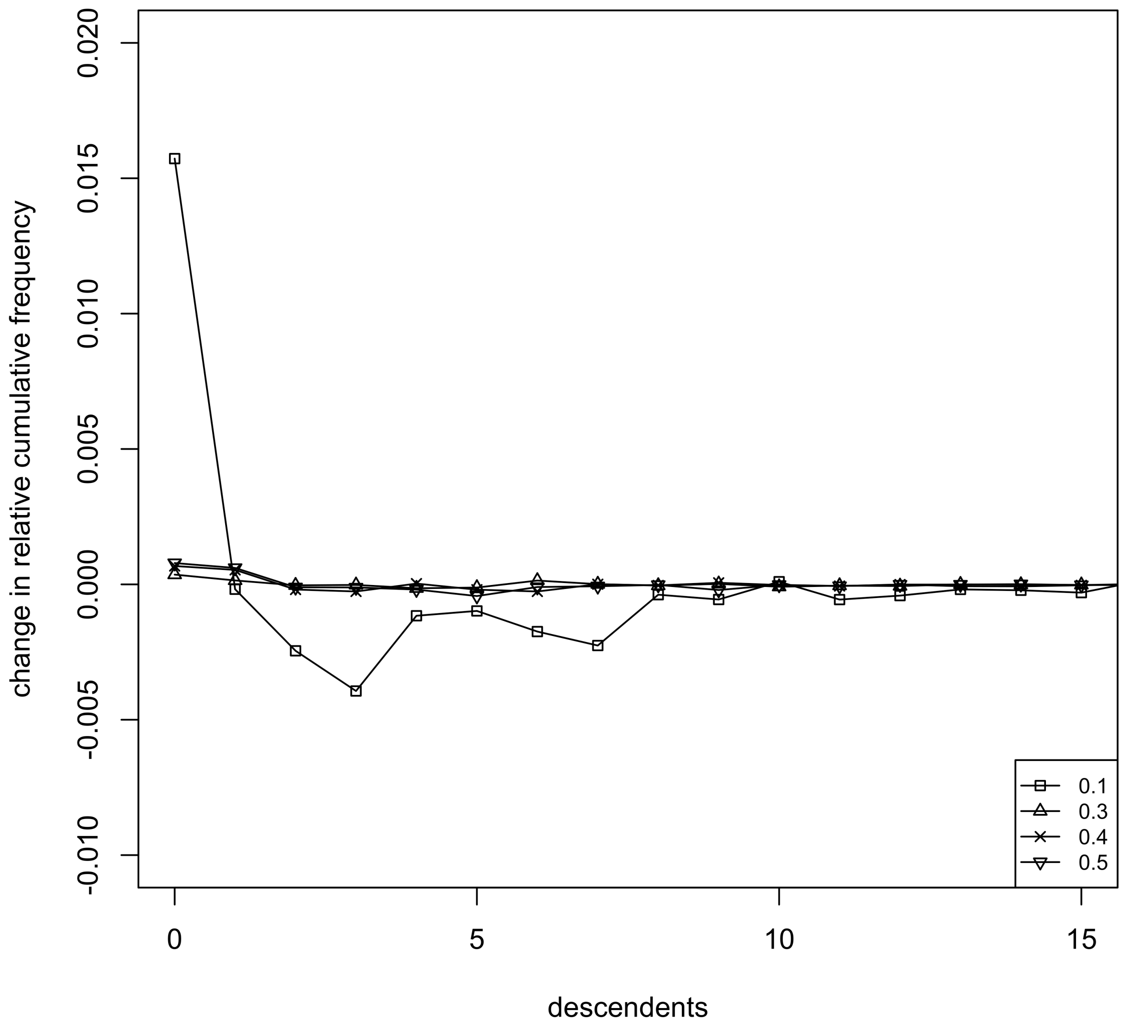


Figure V.9. Change in the total number of descendants per host upon death for various host reproduction probabilities (g) attempting to invade a resident host population with $g = 0.2$. Changes are relative to the distributions reported for the homogenous scenarios (Figure V.5). In the context of the resident population, invading individuals tend to produce fewer descendants than they would in a population of their own kind.

These results demonstrate that prudent hosts can evolve. Turning to the second question, how does this evolved population perform compared to the range of possible strategies? Clearly, the answer to this question depends on what is meant by "perform." In terms of resisting invasion by competing strategies, the PIP and the long-term dynamics of the model indicate that the evolved population reaches a global optimum. By other equally reasonable measures of performance, however, the population does not perform optimally. For example, the average population size, or standing crop, of the host population can be improved by decreasing the host reproduction probability below the

ESS (Figure V.10). The variability and minimum size of the population, which would both affect the likelihood that the population as a whole would be wiped out by a large, random, mortality-inducing event, are also not minimized at the ESS (Figure V.10). By reproducing more slowly than the ESS, the spatial structure of the host population is maintained in a patchwork of very small, isolated clusters, such that the host population is highly resistant to the spread of the pathogen; the host population persists steadily at a large size, perturbed neither by large epidemics nor rapid expansion of clusters.

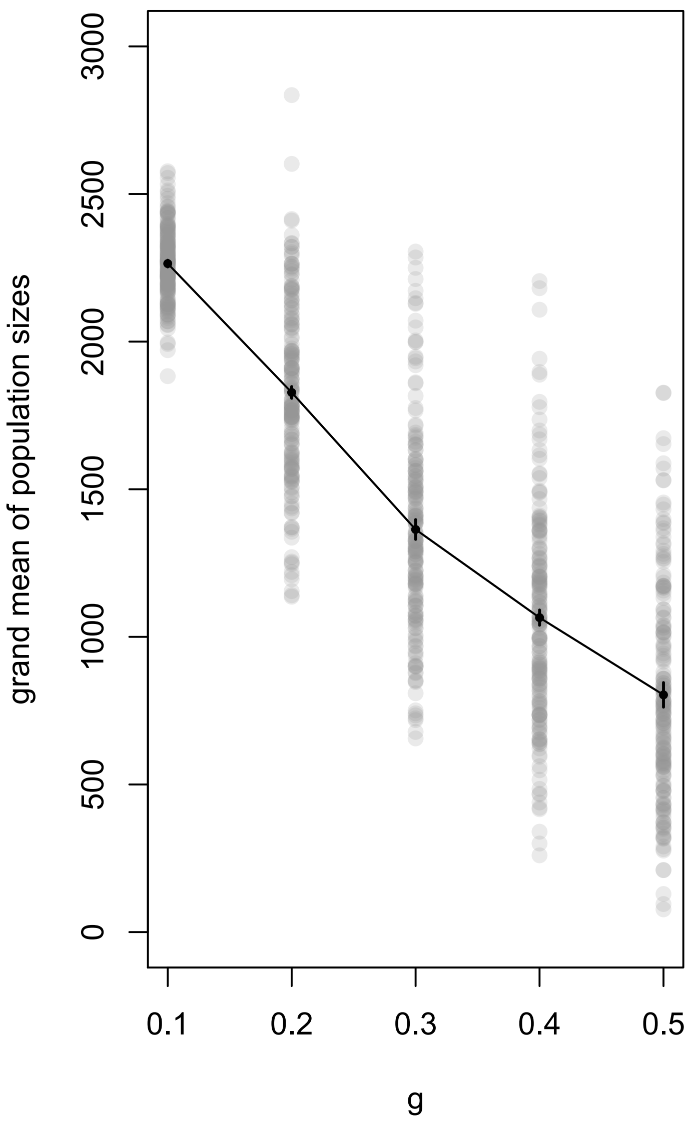


Figure V.10. Grand means of the host population size versus the host reproduction probability (± 1 s.e.m.). For each value of *g*, 50 realizations were performed. The populations were sampled every 1000 time steps for 18000 time steps, discarding the first 1000 time steps to avoid the initial transients. Gray circles show the average populations for the individual realizations.

Discussion

The significance of these results is twofold. First, they demonstrate the potential for an evolutionary phenomenon – reproductive restraint by victims as an anti-exploiter strategy in a spatially-explicit exploiter-victim system – that has potential relevance well beyond the host-pathogen system considered here. Second, they hint at the potential importance of more comprehensively exploring the relative performance of evolved behaviors. Although the latter point is primarily a philosophical one, and is more a matter of emphasis rather than the discovery of a particular new phenomenon, it is an important consideration given the ubiquity of autogenous processes in natural and human-designed systems.

The basic processes explored here could apply to any system in which a locally-dispersed victim that is subject to attack by a locally-dispersed exploiter is capable of evolving. Forests of trees subject to intermittent forest fires are an example that has received significant attention from modelers (Zinck and Grimm 2009). Spatially-explicit predator-prey systems, provided that both predator and prey are dispersal limited, are another biological example (Hassell et al. 1991).

Although, to my knowledge, this is the first demonstration of the evolution of reproductive restraint as an anti-pathogen phenotype, the evolution of other host characteristics in spatially-explicit systems has been explored. For example, Socolar et al. (2001) used a 2D lattice model to show that the non-disease-induced (natural) mortality rate of hosts that are subject to rare epidemics will evolve to an intermediate value. Furthermore, the system was shown to evolve to a state of self-organized criticality in which epidemic size distributions were characterized by a power law.

Moving from a theoretical demonstration of plausibility to detection of prudent hosts in real biological systems, as has been done for the evolution of prudent pathogens, will not be a trivial task. The evolution of prudent hosts demonstrated in the current study rests on a number of assumptions that are implicit in the structure of the model. First, it is assumed that there are no secondary factors influencing the relationship between transmission rate and host density, i.e., transmission is assumed to increase with the number of susceptible hosts in the neighborhood. Second, it is assumed that hosts are able to control (in an evolutionary sense) their reproduction rate.

As a review by Burdon and Chilvers (1982) of the effects of host density on plant disease ecology demonstrates, transmission does not necessarily increase with host density. The net effect of density can be counterintuitive in cases where indirect effects outweigh direct effects. Direct effects of increasing density include (1) an increase in the number of host plants that the inocula, transmitted through space and time, can impinge upon (the more plants there are per unit area, the more likely it is that inocula will land on a host) and (2) a decrease in the distance that spatially-dispersed inocula must travel to spread from plant to plant. These two effects are mutually reinforcing. Indirect effects can arise from interactions between: environment and host properties, e.g., changes in host size, shape, or nutritional status; environment and inoculum properties, e.g., changes in microclimate; environment and vector behavior; and environment and incidence of other plants.

Burdon and Chilvers cite a number of examples of systems in which these direct effects appear to dominate: of 69 studies (46 different host-pathogen combinations), they found that 39 (57%, or 62% of the host-pathogen combinations) exhibited a positive correlation between disease incidence and host density, which is what one would expect in cases where direct effects of density dominate. In 24 studies (35%, or 27% of host-pathogen combinations), however, there was a negative correlation between disease incidence and host density, which can only occur if there are one or more indirect effects that exceed the influence of direct effects. For example, one system with an inverse correlation was an aphid-plant system in which aphids responsible for spreading groundnut rosette disease were attracted to yellow light wavelengths reflected from soil, and were repelled by blue wavelengths reflected from dense crop covering. In another example, Ergot infections increased at lower densities due to increased tillering (formation of new shoots) of the host at lower densities, which led to the "development of more heads and an extended flowering period over which the plants remained susceptible to infection." As these examples show, the positive relationship between host density and disease transmission assumed in the current model may be less ubiquitous than one might expect.

Although populations of plants are perhaps the most obvious examples of systems characterized by relatively static spatial structures that might affect disease dynamics, other sessile organisms are also candidates for this sort of phenomenon, e.g., fungal infection of sponges (Galtsoff et al. 1939) and sea fan corals (Kim and Harvell 2004) and the isolation-modulated susceptibility of prairie dog colonies to plague (Lomolino et al. 2001). The susceptibility of colonies of eusocial insects to locally transmitted diseases would also seem likely to exhibit a dependence on host spatial distribution, but this does not seem to have been studied. However, results indicating that slave-making ants may evolve a “prudent predator” strategy in response to lower densities of host colonies in a manner that is analogous to the evolution of less virulent strains of diseases (Foitzik et al. 2001) suggest that spatial distribution may play an important role in spatial disease dynamics of eusocial insects as well.

The second assumption underlying the current model, i.e., that the spatial distribution of hosts is determined in a large part by a heritable reproduction rate, is particularly problematic. In nature, it is likely that there are a number of biotic and abiotic factors that influence the spatial distribution of hosts and the effect of this distribution on disease transmission. It seems likely that these influences will overwhelm the influence of the host's genotype per se. In addition to the likelihood for these environmental factors to disrupt any potential for the evolution of prudence, they would also tend to obscure evidence of prudence, making detection under natural conditions very challenging. As with the evolution of prudent pathogens, however, it may be possible to construct experimental systems that are simple enough to enable the evolution and detection of prudent hosts.

Much of the work on the emergence of cooperation has ranged from pessimistic to Panglossian, with Garret Hardin and Adam Smith (or at least the popular conception of Smith) representing the two competing worldviews. Adam Smith's work is commonly distilled into the belief that a man who "intends only his own gain" is naturally "led by an invisible hand to promote an end which was no part of his intention...By pursuing his own interest, he frequently promotes that of the society more effectually than when he really intends to promote it." (Smith 1776). Hardin, in contrast, argued that this individualistic pursuit of self interest leads inexorably to the now-classic "tragedy of the commons" (Hardin 1968). Hardin's rather repressive, militant conclusion was that only through "mutual coercion, mutually agreed upon," i.e., laws enforced such that individuals are prevented from overexploiting the commons, can this tragedy be averted. These two extremes – autogenous processes as threats and as saviors – have bookended the debate.

This dichotomy is manifested in the dominant narrative in the literature on the evolution of cooperation. Yes, the story goes, the null expectation is that tragedy will prevail. But, if at least one of multiple possible mechanisms is present, cooperation will emerge, and tragedy is averted. Nowak (2006) identified these processes as kin selection, direct reciprocity, indirect reciprocity, network reciprocity, and group selection. The literature is rich with examples demonstrating these phenomena: in a well-mixed population, an evolutionary Prisoner's Dilemma favors defectors, but repeated interactions give cooperators an advantage (Axelrod and Hamilton 1981); without spatial structure, rapacious bacteriophages outcompete prudent phages, leading to overexploitation of the bacterial prey, while local migration leads to competitive restraint and increased productivity (Kerr et al. 2006); spatial structure maintains castration virulence at an intermediate level in a mutualism between an ant-plant and its ant symbiont (Szilágyi et al. 2009); through alternating rounds of public goods and indirect reciprocity games, reputation helps solve the "tragedy of the commons" (Milinski et al. 2002); and many others. The conclusion is clear: a superficial acquaintance with the autogenous processes that pervade nature would lead one to predict a race to the bottom, but upon closer inspection we find that cooperation emerges unexpectedly from the melee.

What the results of the current model suggest, however, is that a certain degree of nuance is missing from this narrative, in tone if not in the actual results. Cooperation is typically framed as a binary characteristic; either a system exhibits cooperation, or it is characterized by selfishness. While in some model formulations this either/or dichotomy is appropriate, e.g., a Prisoner's Dilemma in which all players cooperate fully, typically there are degrees of cooperation that are achieved. What fraction of individuals cooperate? What percentage of the time do individuals display altruistic behavior? How much better could pathogens do if they reduced their virulence even more? How prudent is prudent, really? And, to what extent is performance, in terms of attributes that are peripheral to the main foci of evolution and the researchers, maximized?

Though here I am focusing on the evolution of cooperation, the observation that evolution is an imperfect and myopic device for optimization applies more broadly, and has been pointed out many times before (Gould and Lewontin 1979, Arnold 1992). Evolution is path dependent, meaning that the evolutionary outcome is contingent on history (Jacob 1977). In biological evolution, history influences the range of what can evolve – the "phenotype set" (Smith 1978). If this were not the case, perhaps we would have evolved bicycles instead of having to rely on our relatively inefficient legs for transport, and quadrupeds would not have to give up one pair of limbs in order to evolve wings. Even with an unrestricted phenotype set, as is usually assumed in computer models, the particular path traversed by the evolutionary process may lead to the system being stuck at a local optimum. Evidence of this is readily available from optimization theory. For example, genetic algorithms, an optimization technique that is modeled on the principles of natural selection, is known to fail to find the global optimum under some conditions (as do all optimization routines) (Schaffer et al. 1991).

The point of this discussion is not to say that the evolution of cooperation is not an important phenomenon, but rather to advocate for a more comprehensive framing of the possible results. First, the extremes represented by Hardin and Smith should been seen as, respectively, one end of a continuum of possible outcomes and another point lying at some intermediate position on this continuum, rather than as demarcating the range of possible outcomes. The other end of the continuum of cooperation, i.e., perfect cooperation, should also be identified as a point of reference. Second, we can obtain a more nuanced understanding of the outcomes of evolutionary processes by assessing the performance of the evolved population using multiple measures. For example, in the prudent host model presented here, the model is defined such that it evolves towards maximum individual fitness. However, by other measures, such as host population size and variability, the evolved population is not optimized. Although this point may seem obvious – the system evolves to optimize the optimization criterion, but not other criteria – it is one that tends to be forgotten when we speak of the "evolution of cooperation."

Although this discussion may seem like an unnecessary exercise in pedantry, the way we frame discussions of the evolution of cooperation can have important consequences for how these concepts are viewed in the popular imagination. (And, as the dark history of Social Darwinism makes clear, scientists have a responsibility to consider the potential for their work to be misconstrued and misapplied in the public sphere.) For example, when Nowak (2006), one of the leading researchers of the evolution of cooperation, states that

> "Humans are the champions of cooperation: From hunter-gatherer societies to nation-states, cooperation is the decisive organizing principle of human society. No other life form on Earth is engaged in the same complex games of cooperation and defection. The question of how natural selection can lead to cooperative behavior has fascinated evolutionary biologists for several decades."

the implication is that human cooperation has arisen autogenously via evolutionary processes. This raises two questions. First, what is the evidence that human cooperation arises directly from the actions of evolution, and not through a more rational process? Second, if human cooperation did evolve, what does that imply? What is the optimization criterion; to what extent has evolution achieved optimization; and how do the evolved behaviors perform in terms of other, non-optimized criteria? Without these details, we risk seeing the evolution of cooperation in a cartoonish manner, as a magical process from which spring forth desiderata, in much the same way as in the modern-day caricature of Adam Smith's invisible hand.

Chapter VI

Self-organization of background habitat determines the nature of population spatial structure

Understanding the distribution of organisms in space is essential to many areas of applied ecology, such as conservation (Hanski and Thomas 1994, Bulman et al. 2007), agroecosystem management (Thies and Tscharntke 1999, Bianchi et al. 2006, Perfecto and Vandermeer 2010), delivery of ecosystem services (Brosi et al. 2008), and epidemiology (Grenfell et al. 2001), among others (Kritzer and Sale 2004), and has also become a key element in the general theory of community structure of terrestrial, aquatic, and marine ecosystems (Tilman and Kareiva 1997, Werner et al. 2007). Key to this understanding has been the nature of the underlying habitat structure in which the population is embedded, islands conjuring the theory of island biogeography, isolated habitats suggesting metapopulations, and off-coast archipelagos envisioned as source/ sink populations (Levins 1969, Pulliam 1988, Rohani et al. 1996, Hanski and Gilpin 1997, Holt 1997, Hanski 1998, Moilanen and Hanski 1998, Hanski 1999, Amarasekare and Nisbet 2001, Vandermeer et al. 2010b). Yet a detailed analysis of the nature of that underlying habitat structure is lacking, despite its obvious importance for the structure of the occupying populations.

One way of examining underlying habitat structure is to examine its origin. While some habitats have obvious structural determinants (e.g., woodlots in eastern North America are largely a consequence of political boundaries), others derive from dynamic processes. Here we offer an approach based on the principle of self-organization. Typical self-organizing dynamics generally lead to a scale-free distribution of habitat patch sizes (Rohani et al. 1997, Bascompte and Solé 1998, Klausmeier 1999, Pascual et al. 2002, Rietkerk et al. 2002, Newman 2005, van de Koppel et al. 2005, Solé and Bascompte 2006, Alados et al. 2007, Scanlon et al. 2007, Rietkerk and van de Koppel 2008), the details of which determine to a great extent the nature of the population dynamics of any organism occupying those patches.

This framework is motivated by the concrete case of the spatial patterning of the arboreal ant *Azteca instabilis* F. Smith (Hymenoptera: Formicidae) and the use of that spatial pattern by its mutualist associate, the green coffee scale *Coccus viridis* Green (Hemiptera: Coccidae), in a coffee farm in southern Mexico (Perfecto and Vandermeer 2008b). The ant, in association with one or more natural enemies, generates a scale-free distribution of patches of nests in a uniform environment (Vandermeer et al. 2008, Jackson et al. 2009), and several other populations (beetles, spiders, fungi, in addition to the scale insect itself) become associated with those clusters in a complex fashion (Liere and Perfecto 2008, Livingston et al. 2008, Vandermeer et al. 2009). Each of these other populations uses the clusters of ant nests as basic habitat patches, and the question arises as to what is the structure of their populations as a function of the nature of the habitat patches, which have been constructed in an autonomous fashion through the principle of self-organization (Vandermeer et al. 2008). It is an example of a complex situation that evidently occurs throughout the natural world: habitat spatial distributions created by biological interactions into which independent populations are accommodated. Whether self-organization of habitat patches tends to promote or hinder the persistence of populations that inhabit these patches is thus a question of fundamental and general interest.

Although there are myriad ways of categorizing spatial population structure, two extreme cases emerge as particularly common, the metapopulation and the source/sink population (Figure VI.1). It is clearly possible to view these two canonical forms as extremes on a continuum. For many practical reasons it is useful, sometimes absolutely necessary, to know whether a population is a metapopulation or a source/sink population. For example, in the conservation context, a source/sink population commands attention to the location of the source population as the most important target for management activities. In contrast, a metapopulation structure suggests that the overall landscape would be the proper focus of management so as to maintain sufficiently high interhabitat migration (Perfecto and Vandermeer 2002). Many other examples could be cited. Here we consider the case of a population that is potentially either a metapopulation or a source/sink population and ask how its nature is fundamentally determined by the way in which the underlying habitat is structured through self-organization.

The metapopulation-source/sink continuum: theory

The general analytical model we propose is based on three key relations. First, the success of a source/sink population is determined largely by the size of the largest habitat patch (the source). The local extinction rate is a decreasing function of patch size and thus, all else equal, the average extinction rate will decline with the average size of the habitat patch. In turn, the average size of the habitat patch will be highly correlated with the average size of all habitat patches, which suggests the approximation,

$$e = f(c_m) \tag{1a}$$

where e is extinction rate and c_m is the size of the largest patch.

Second, the success of a metapopulation is determined by the ratio of the migration rate to the extinction rate. The migration rate, in turn, is determined principally by the distance between habitat patches, which we assume is determined in part by the number of patches. Thus,

$$m = g(n_T) \tag{1b}$$

where m is migration rate and n_T is the total number of habitat patches.

Third, although certainly the details will be more complicated, we make the assumption that the total number of habitat patches will normally be related in some fashion to the size of the largest patch, or,

$$n_T = h(c_m) \tag{1c}$$

For example, in a situation in which the overall biomass or population density of a population is constant, if almost all the individuals or biomass is contained in one particularly large patch, the overall number of patches will be limited to a very small number (since almost all the individuals are members of that largest patch).

In general, we can envision the possible population structures on a simple graph of *e* versus *m*, according to standard definitions of extinction and migration, as pictured in Figure VI.1. Furthermore, through the process of composition, we see that,

$$m = g\left\{h[f^{-1}(e)]\right\} = F(e) \tag{2}$$

presuming, of course, that *f* has an inverse. The process of composition and its resulting stipulation of the relationship between *m* and *e* can be easily viewed graphically (Figure VI.2). Particular patterns of habitat organization will thus produce particular patterns of *F*, leading to particular population structures. Those structures will obviously change as management decisions provoke changes in either the form of *F* or the parameter values it contains.

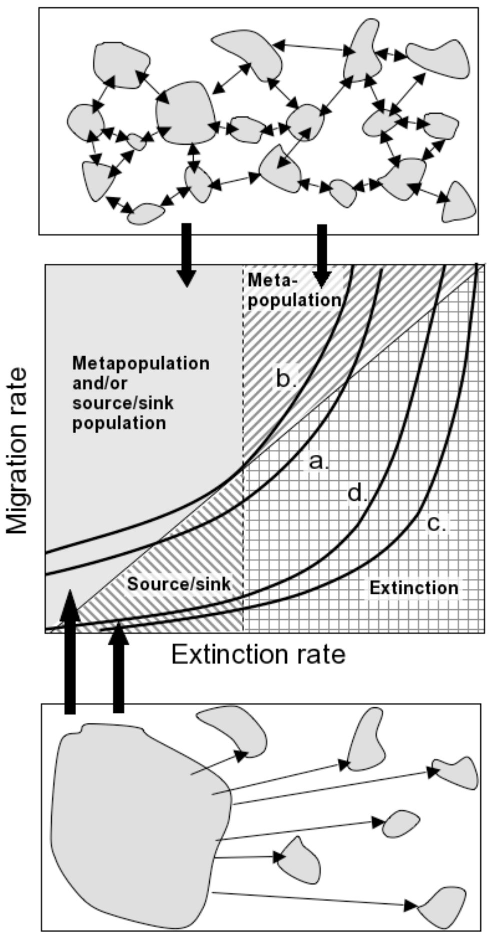


Figure VI.1. Diagrammatic illustration of the two extreme forms of population organization in a fragmented habitat. Top panel illustrates a metapopulation in which no given habitat patch can sustain a population in perpetuity, but the interhabitat migration is sufficiently large to offset extinctions from the small patches. Bottom panel illustrates a source/sink population in which one of the patches is large enough to sustain a population in perpetuity, the source population, while the others cannot. The smaller patches thus contain sinks in that any subpopulation existing in them will eventually become locally extinct. Middle panel illustrates the dynamics of each type of population with respect to the overall migration rate and the within-patch extinction rate. The upper triangle, in which a metapopulation is possible, is separated from the lower triangle, in which a metapopulation is not possible, by the standard metapopulation equilibrium, $p^* = 1 - (e/m)$ where p^* is the equilibrium fraction of the habitats occupied, e is the extinction rate and m is the migration rate. The dashed vertical line is the critical extinction rate above which the probability of having at least one patch capable of sustaining a viable population even in the absence of significant migration approaches 1.0. Lines a-d show the relationship between extinction and migration assuming the underlying habitat is self-organized and thus has a scale-free distribution that follows a power law. With increasing extinction rate (decreasing size of patch), we have an increasing migration rate (larger number of patches), assuming the overall habitat area is held constant. a: a_e=12; b_e=.25, a_m=1;p_T=100. b: a_e=12; b_e=.25, a_m=1; p_T=120. c: a_e=12; b_e=.1, a_m=.4; p_T=250; d: a_e=12; b_e=.2, a_m=.3; p_T=180. In scenario a, the population goes from a metapopulation/source/sink population, to a strictly source/sink population to population extinction to metapopulation, as extinction and migration increase. Changing the overall habitat area, we obtain scenario b, where the population exists first as a metapopulation/source/sink combination, but after the migration rate passes its critical point, becomes strictly a metapopulation. In scenario c the population begins as a source/sink population, then becomes extinct, but, with yet further increase in extinction rate, a metapopulation emerges. Scenario d illustrates the unusual case in which a source /sink population is driven to extinction, but then emerges at a much higher extinction/migration combination as a metapopulation.

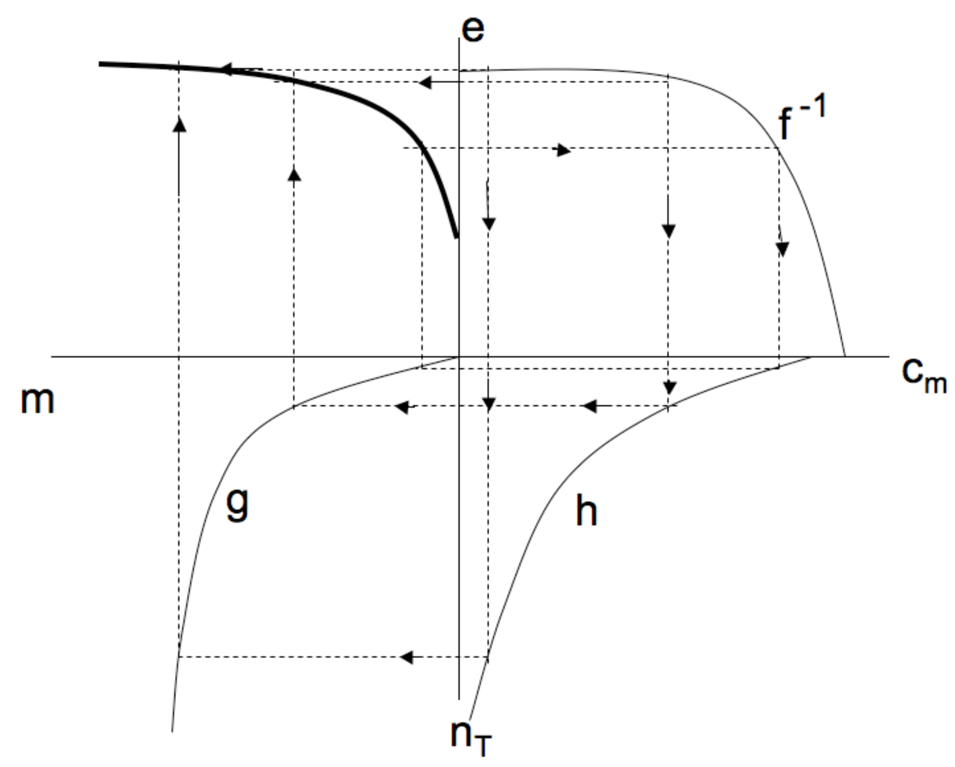


Figure VI.2. Graphical composition of the three essential functions to produce the relationship between migration rate (*m*) and extinction rate (*e*), based on the fundamental monotonic relationship between c_m and n_T. The final function gives a qualitative functional form to the relationship between migration (*m*) and extinction (*e*).

Basic rules of spatial dynamics frequently produce patterns in which clusters of individuals form habitat patches and those patches themselves are distributed according to a power law (Pascual et al. 2002, Newman 2005, Scanlon et al. 2007), at least in some likely situations (Kéfi et al. 2011). Thinking of this relationship as canonical, we ask, if patch sizes are distributed according to a power law, what will be the form of *h*? (and later, of *F*, making some reasonable assumptions about *f* and *g*).

We begin with the fundamental power law distribution,

$$v(c) = ac^{-b}\exp(-c/S) \tag{3}$$

where *c* = patch size, *v* is frequency, *b* and *a* are constants, and *S* is the so-called cutoff point where *v*(*c*) begins declining faster than the power law at lower values of *c*. Strictly speaking, as *S* approaches infinity, equation 3 becomes a pure power law. This is precisely what is expected if the population is at a critical state, whether driven there by some key parameter or evolving there by a self-organizing process (Bak 1996). Also, as discussed later, if the population in question exhibits robust scaling (Pascual et al. 2002, Kéfi et al. 2011), the assumption that *S* is very large, such that exp(-*c*/*S*) = 1, may be warranted for many systems. We proceed with that assumption.

Given that patches are made up of particles (individuals, biomass units, etc.), the total number of particles in the system is given as,

$$p_T(c) = \int_1^{c_m} cac^{-b}\mathrm{d}c = \int_1^{c_m} ac^{1-b}\mathrm{d}c = \frac{a}{2-b}c_m^{2-b} - \frac{a}{2-b} \tag{4}$$

which we assume is constant. The total number of patches is given as,

$$n_T(c) = \int_1^{c_m} ac^{-b}\mathrm{d}c = \frac{a}{1-b}c_m^{1-b} - \frac{a}{1-b} \tag{5}$$

From 1 we note that c_m (the largest patch size) occurs when *p(c)* = 1, giving,

$$a = c_m^b \tag{6}$$

which, when substituted into 5, gives us,

$$n_T = \frac{c_m - c_m^b}{1-b} \tag{7}$$

From equation 4, we write,

$$c_m^{2-b} = p_T\frac{2-b}{a} + 1 \tag{8}$$

If we restrict our analysis to p_T large, equation 8 becomes,

$$c_m^{2-b} = p_T\frac{2-b}{a}$$

which, after substituting from 6 and rearranging, becomes,

$$b = 2 - \frac{c_m^2}{p_T} \tag{9}$$

Substituting 6 and 9 into 5, we obtain,

$$n_T = \frac{c_m - c_m^{\left(2 - \frac{c_m^2}{p_T}\right)}}{\frac{c_m^2}{p_T} - 1} = h(c_m) \tag{10}$$

If we now assume linearity for the functions *f* and *g*, such that $e = a_e - b_e c_m$ and $m = a_m n_T$, and taking the inverse of *f*, we have $c_m = (a_e - e)/b_e$. Substituting these linear terms to compute the composed function (equation 4), we obtain,

$$m = a_m \frac{\psi(e) - \psi(e)^{\left(2 - \frac{\psi(e)^2}{p_T}\right)}}{\frac{\psi(e)^2}{p_T} - 1} \tag{11}$$

where,

$$\psi(e) = \frac{a_e - e}{b_e}$$

In Figure VI.1 we illustrate four scenarios of increasing extinction and migration rates (using equation 11), assuming that the overall area of habitat is constant and the frequency of patch sizes is distributed as a power function.

To further illustrate the dynamics, we used a discrete-time, lattice-based model (Appendix B) to simulate patch occupancy dynamics on an artificial landscape of habitat patches whose sizes were drawn from a power-law distribution. The landscape was modeled using a 200 X 200 cell, two-dimensional, square lattice, with each cell being designated as either habitat or non-habitat. Since we are interested in ecological systems, which are of finite size, non-periodic boundaries were used. The distribution of patch sizes was created using the method of approximating a discrete power-law distribution from a continuous distribution detailed in Clauset et al. (2007). Using a random real *r* drawn from a uniform distribution, $0 \leq r < 1$, an integer patch size *c* can be calculated:

$$c = \lfloor (\tfrac{1}{2})(1 - r)^{-1/(\alpha - 1)} + \tfrac{1}{2} \rfloor \tag{12}$$

where α is the power function exponent, or scaling parameter and the closing brackets are floor symbols. Patches were drawn repeatedly in this manner until the sum of the patch sizes was equal to or greater than 1200, i.e., the total habitat area was approximately equal for all runs.

Following generation of the patch size distribution, each patch was placed randomly in the lattice such that no two patches were touching using the following process. The first particle in each patch was placed in a randomly-chosen, empty location with no existing particles in the neighboring eight cells (the Moore neighborhood). For patches of size $c > 1$, a neighboring, empty cell with no other particles in the Moore neighborhood was then chosen for the next particle, and this process was repeated until a total of c cells had been designated.

All patches in the metapopulation were initially occupied. At each time step, local extinction in each patch was determined based on patch size. In each time step, the probability of extinction was calculated for each patch:

$$P(\text{extinction}) = e_0 \exp(-e_1 c) \tag{13}$$

where $e_0 = 0.9$, $e_1 = 0.03$, and c is the patch size. These values were chosen arbitrarily, with the goal simply being to make the probability of extinction a decreasing function of patch size. Rescue of extinct (unoccupied) patches was determined based on their proximity to other occupied patches. The probability of rescue was calculated as:

$$P(\text{rescue}) = 1 - \prod_{i=1}^{N} (1 - m_0 \exp(-m_1 d_i)) \tag{14}$$

where N is the number of occupied patches, $m_0 = 0.9$, $m_1 = 0.25$, and d_i is the shortest Euclidean distance from the focal patch to patch i. Again, these parameter values were chosen somewhat arbitrarily to achieve an increasing rescue probability with an increasing abundance and/or proximity of neighboring occupied patches.

The model was swept over a range of scaling parameters, from 1.5 to 2.8, with a step size of 0.025. For each value of the scaling parameter, 100 runs were performed. Each realization was run for 1000 time steps, and the average fraction of patches occupied was calculated for the final 100 steps; trial runs had demonstrated that steady state was reached after approximately 200 time steps, so it is reasonable to assume that this average excludes transient behavior. For runs in which the metapopulation went extinct, the average fraction of patches occupied was defined to be zero.

The results of the simulation of the theoretical species are displayed in Figure VI. 3. It is evident that the population lives as a metapopulation for a very low power function scaling parameter, basically because the largest patch size is too small to form a source population, but there are a large number of patches insuring a high overall migration rate. As the scaling parameter increases, the population moves toward extinction, largely due to the low number of patches resulting in a lowered migration rate but without the concomitant emergence of at least one large patch to accommodate a source population. At very high values of the scaling parameter, the population is maintained as a source/sink population due to the existence of at least one large habitat patch that houses a source population. The simulation is reminiscent of cases a and d of Figure VI.1.

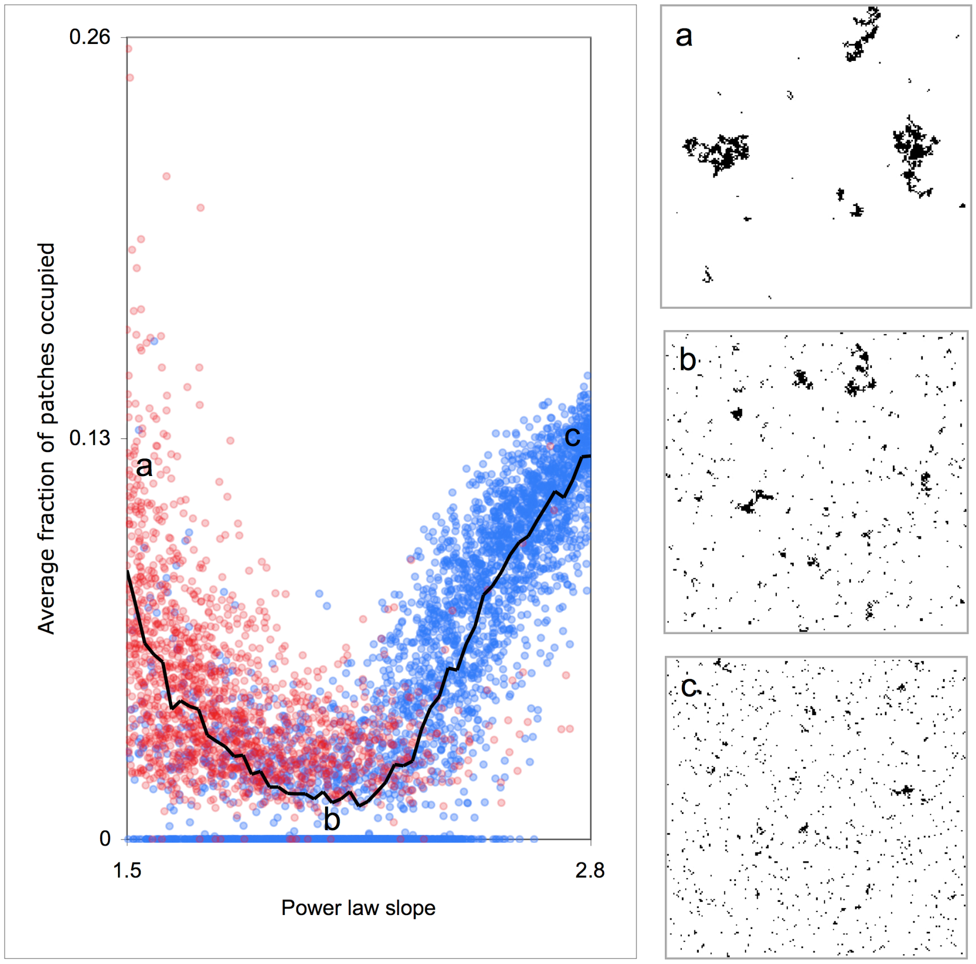


Figure VI.3. Average fraction of patches occupied as a function of the scaling parameter of the original "self-organized" habitat distribution, from simulation experiments. Red points, clustered on the left, signify a population maintained as a source/sink population. Blue points, clustered on the right, signify a population maintained as a metapopulation. The black line is the average of 100 realizations at each scaling parameter setpoint.

Self-organization of habitat patches and consequences for equilibrium patch occupancy

The above results illustrate the importance of the parameter of the power law of the underlying habitat in determining the fundamental structure of the population in question, that is, whether it will exist as a metapopulation or a source/sink population (or, for that matter, go extinct). However, the assertion of an underlying power law is based on the assumption that the habitats are themselves self-organized. Thus the question naturally arises as to what might be the difference between a population that exists in a set of habitat patches that themselves are self-organized compared to a population existing in habitat patches not so organized, especially ones in which habitat sizes themselves have a frequency distribution that is a power function, but the patches themselves are randomly allocated in space (as in the calculations in the previous section).

To explore this question, we ran the patch occupancy simulation model described previously on a landscape of patches generated by the discrete-time, lattice-based CA model of Vandermeer et al. (2008). Recall that in this framework the habitat to be "constructed" is a shade tree occupied by an ant nest, while the population to be affected (i.e, the organisms that live in these patches) responds to the distribution of the patches of those ant nests. The generation of patches in the CA model occurs as follows: each site in the lattice can adopt one of two states, either occupied by an ant nest or empty. Unoccupied sites are colonized through local expansion of ant nest clusters, while occupied sites become unoccupied with some mortality rate that is an increasing linear function of the number of occupied sites in the Moore neighborhood (the 8 sites surrounding the cell). By varying the intercept of this linear mortality function, the equilibrial number of ant nests (habitat points) can be varied. In this manner, self-organized landscapes with an arbitrary number of habitat points can be generated. Following the generation of the habitat landscape, habitat patches, defined as contiguous clusters of points touching on an edge or corner, were identified.

The self-organized landscapes generated by the CA model were compared to two other scenarios. First, to separate the effects of the frequency distribution of patch sizes from the spatial distribution of patches, the locations of the habitat patches in the landscapes generated by the CA were randomly perturbed to create landscapes with the same cluster (habitat patch) size distributions but different spatial arrangements of the clusters, i.e., the CA-generated patches were dispersed randomly throughout the lattice. Second, landscapes of randomly distributed points were generated to represent the full null expectation without self organization.

Populations inhabiting self-organized landscapes consistently achieve a higher equilibrial fraction of habitat patches occupied than either the null model or the dispersed

CA model (Figure IV.1). The dispersed CA landscapes, despite having the same cluster size distributions as the CA landscapes, was substantially worse than the self-organized landscape.

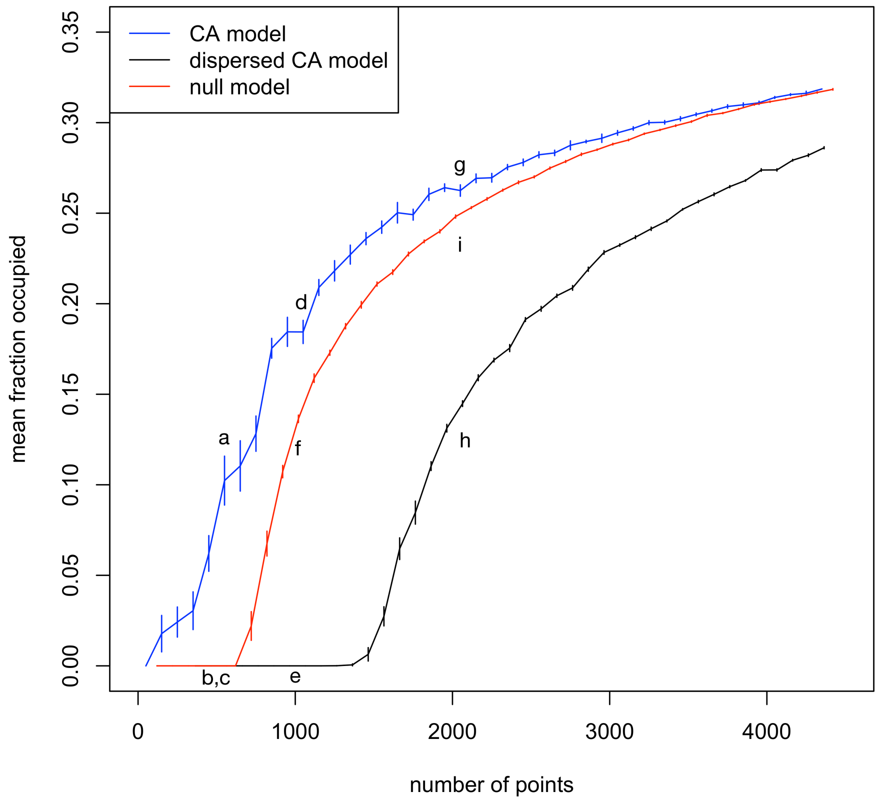


Figure VI.4. The mean fraction of habitat patches occupied for landscapes with different amounts of habitat. The blue line is for habitat landscapes generated by the CA model. The red line is for landscapes generated by randomly placing habitat points in the lattice. The black line is for landscapes generated by randomly dispersing the habitat patches generated by the CA model. Vertical lines show the standard errors of 10 realizations of the model. The mean fraction occupied was calculated from 100 time steps taken after the model had reached steady state (after 900 time steps). The letters correspond to the habitat distributions shown in Figure VI.5.

The qualitative characteristics of the landscapes generated in the three scenarios are markedly different (Figure VI.5). The self-organized habitat displays non-random spatial structure at multiple scales, both at the patch scale and across patches; habitat points cluster to form patches, and the patches themselves are clustered. The dispersed CA landscapes only retain the former (clusters of points), while the latter (clusters of clusters) is, by design, absent. The null landscapes are characterized by greater dispersion, i.e., less clustering, at all spatial scales compared to the self-organized landscapes. Quantitatively, a Ripley's K analysis, which is a measure of clustering of point patterns (Goreaud and Pélissier 1999), corroborates the qualitative picture, with the self-organized landscapes being significantly more clustered at all spatial scales than the dispersed CA and null landscapes (Figure VI.6). The dispersed CA landscapes are significantly clustered at smaller spatial scales due to the clustering of points that form habitat patches, but become less clustered at larger spatial scales due to the random dispersal of the habitat patches. The null landscapes are not significantly clustered at any spatial scale. The null model landscapes are characterized by cluster size distributions with much steeper slopes than the CA and dispersed CA landscapes (which by definition both have the same cluster size distributions) (Figure VI.7). This indicates that the null landscapes have many more small clusters and fewer large clusters than the CA and dispersed CA landscapes.

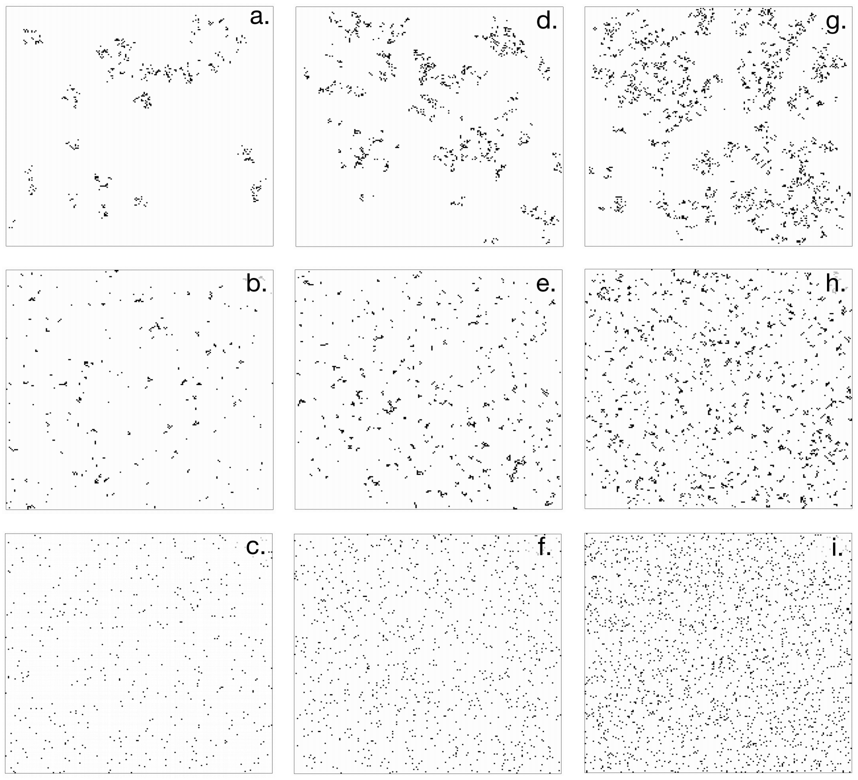


Figure VI.5. Representative landscapes corresponding to the data shown in Figure VI.4. Black points are habitat and white areas are uninhabitable. Row 1 shows self-organized landscapes generated by the CA model with 500 points (a), 1000 points (d), and 2000 points (g). Row 2 shows the corresponding "dispersed CA" landscapes that were generated by randomly dispersing the habitat patches in the CA-generated landscapes. Landscape b was generated by randomly dispersing the patches in landscape a; e corresponds to d, and h corresponds to g. Row three shows landscapes that were generated by randomly placing 500 points (c), 1000 points (f), and 2000 points (i).

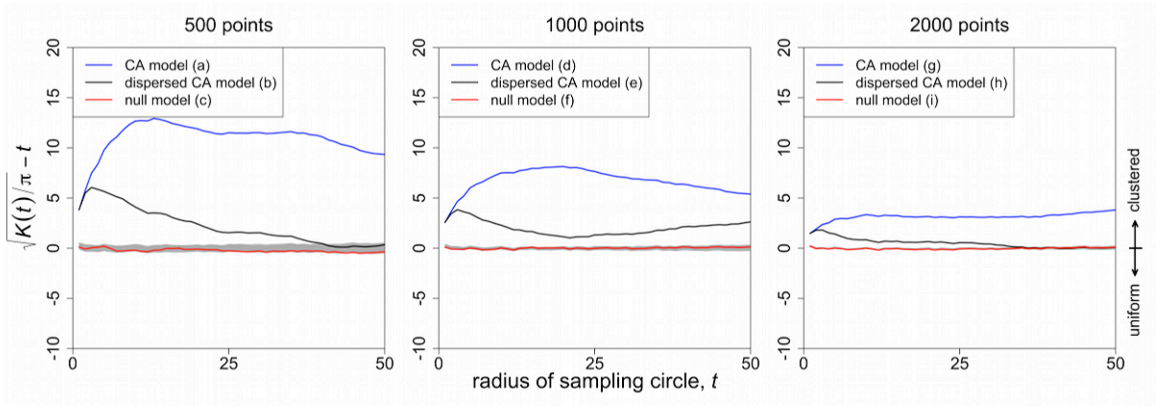


Figure VI.6. Ripley's K, transformed such that the expectation for all sample sizes is zero for a random spatial pattern and greater than zero for clustered patterns, for the scenarios identified in Figures VI.4-VI.5. Blue lines are for the self-organized landscapes generated by the CA model; black lines are for the dispersed CA landscapes; red lines are for the null landscapes; and the gray region shows the 95% confidence intervals for the random expectation based on 200 randomly-generated landscapes. Lines above these gray regions are significantly clustered.

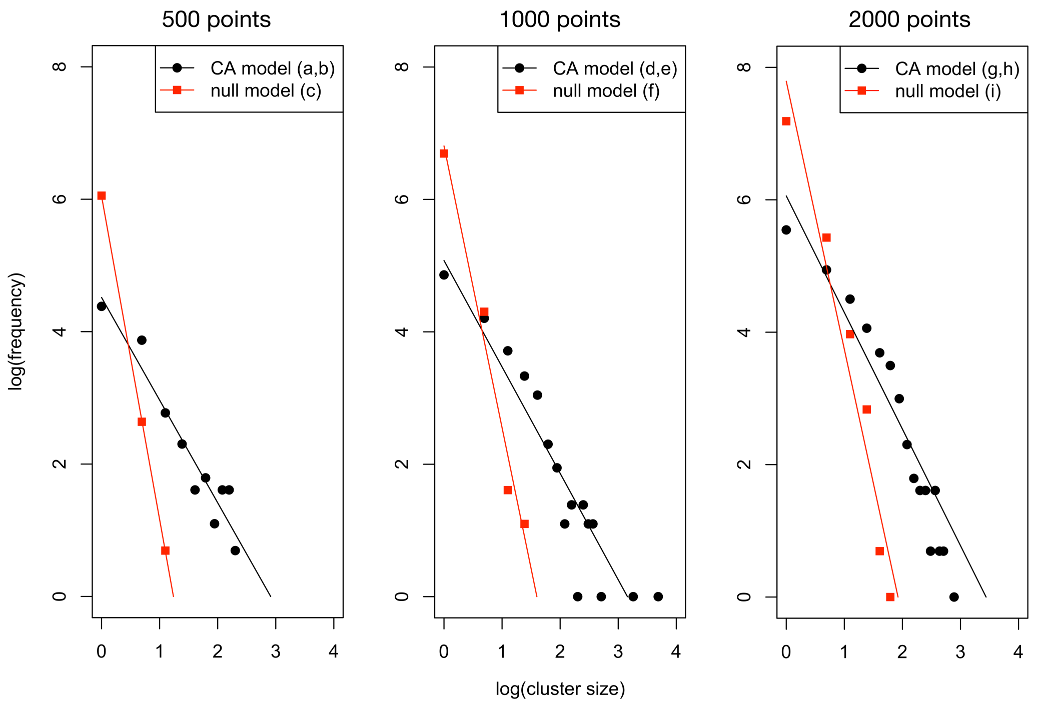


Figure VI.7. Cluster size distributions for the scenarios identified in Figures VI.4-VI.6.

These results suggest that the self-organization of habitat patches, by promoting clustering across a range of spatial scales, creates landscapes that promote the persistence of populations, either as metapopulations or source/sink populations. The scale-free structure of self-organized habitat results in clusters of clusters, thereby providing both the large patches and the short distances between patches that, respectively, avert extinction of occupied patches and foster rescue of unoccupied patches.

An empirical example

In general, if a particular biological system forms the habitat background, the spatial scale of that system may not correspond to the effective spatial scale of the population that occupies it. In our exemplary system, for example, the arboreal ant forms clusters of nests as it responds to a variety of ecological forces (Perfecto and Vandermeer 2008b, Vandermeer et al. 2008, Jackson et al. 2009), whereas the organisms that utilize those nest clusters as habitat patches may be responsive to some spatial scale that is different from the spatial scale that is meaningful to the ants themselves. So, for example, it may be that the dispersal stage of the green coffee scale insect is on the order of 50 meters, while the ants only forage over a distance of 10 meters. The question of spatial scale thus becomes a relative question.

Furthermore, in all cases in which the habitat-forming organism is registered as occupying individual particles of habitat in space, there needs to be some spatial scale over which particles can be regarded as members of the same cluster (patch), i.e., a "cluster scale." This implies that there are actually two parameters that inevitably determine the final clustering of habitat particles (and, therefore, the distribution of patch sizes): the total number of particles and the cluster scale. In theory, one may assume discrete lattices, in which adjacent particles are considered members of the same patch. In practice, however, a self-organizing process will frequently be conceptualized as particles in continuous space, which means that some cluster scale has to be chosen in order to determine patch membership.

Given a particular number of particles in space, the relationship between the number of isolated particles (singletons) and the largest patch (the critical issues with regard to population structure, as discussed in the theory section above) is obviously a negative relationship: as the clustering scale increases, the number of singletons declines, and the size of the largest patch increases. For example, we illustrate this relationship for a random allocation of 761 particles in Figure VI.8 (bold lines). It is evident that where these two functions cross, the slope of the power function that describes the scale-free distribution of particles should be approximately -1.0 (the intercept on the y axis is the total number of singletons and the intercept on the x axis is the number of particles in the largest patch; when both are equal, the line connecting the two will have a slope of -1 on a logarithmic scale).

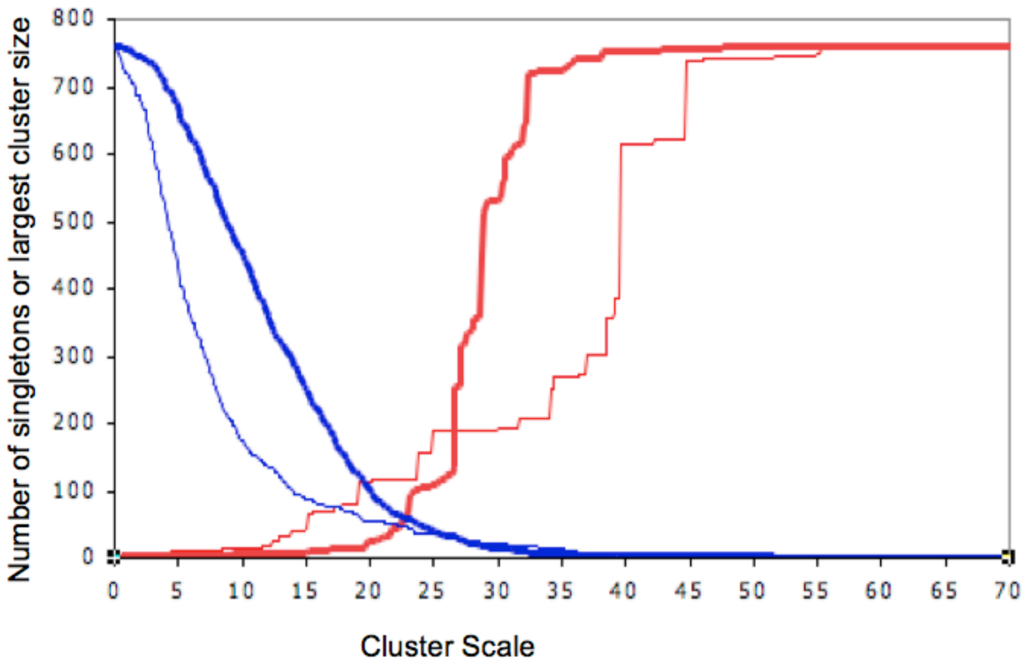


Figure VI.8. Relationship between cluster scale and number of singletons (blue curves descending) and number of points in the largest patch (red curves ascending). Bold lines are from a random allocation of 761 points. Fine lines are from the actual position of 761 nests in a 45 ha plot in a coffee farm in southern Mexico.

In Figure VI.8 we also show the distribution of the numbers of singletons and largest patch size, for the same range of cluster scales, for the actual distribution of nests of the ant *A. instabilis* in our 45 ha study plot in May of 2010. We note that the two functions cross at a cluster scale of approximately 17 (i.e., the ant nests are responding to one another over a range of 17 meters at this point).

The population of concern, which is to say, the population that "occupies" the habitat patches created by the process of clustering of ant nests, is the green coffee scale insect, the hemipteran *C. viridis*, that forms a mutualistic association with the ants. That is, the ant (which tends *C. viridis*) creates the background habitat into which the scale insect must fit. Every dry season the hemipteran populations drop to very low levels except in some of the ant nest clusters where residual populations persist (Figure VI.9). In surveys of the entire 45 hectares performed from January to April of 2009 and again from March to May of 2010, before the beginning of the rainy season, we recorded the presence/absence of an ant nest in each of the approximately 8,000 shade trees in the 45 hectare plot and noted whether there were hemipterans in nearby coffee bushes. Using a cluster scale of 17 m (choosing the point where the two functions cross in Figure VI.8) we present the results of these surveys in Figure VI.10. Note that at this scale, the insects are concentrated in the larger clusters of ant nests and that the concentrations of scale insects generally persist in the same nest clusters from year to year, precisely as would be expected for a source/sink population. Thus, it would appear that the self-organizing attributes of the arboreal ants create the patch structure that generates a source/sink dynamic for the green coffee scale insect.

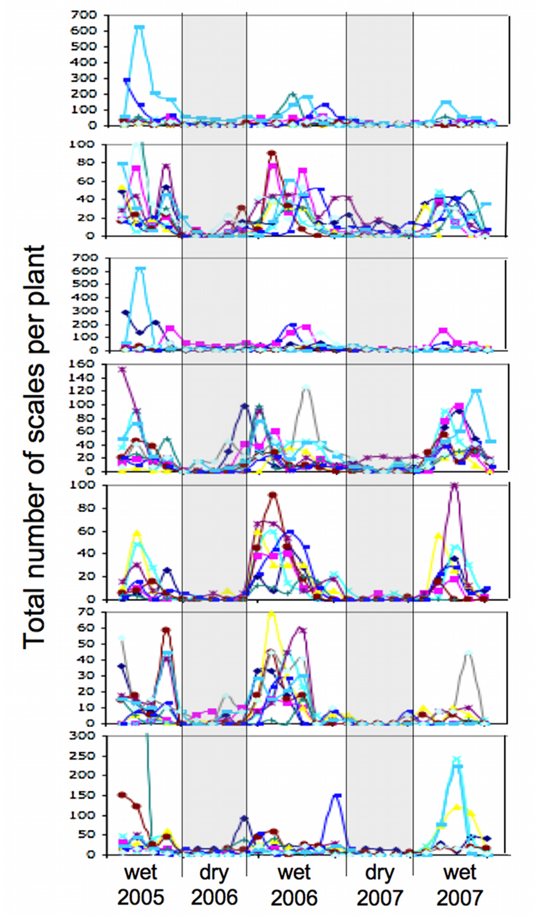


Figure VI.9. Time series of 35 populations of *Coccus viridis* at 7 distinct locations in a 45 ha plot on an organic coffee farm in Chiapas, Mexico. Note the distinct decline to almost zero during each dry season for all populations.

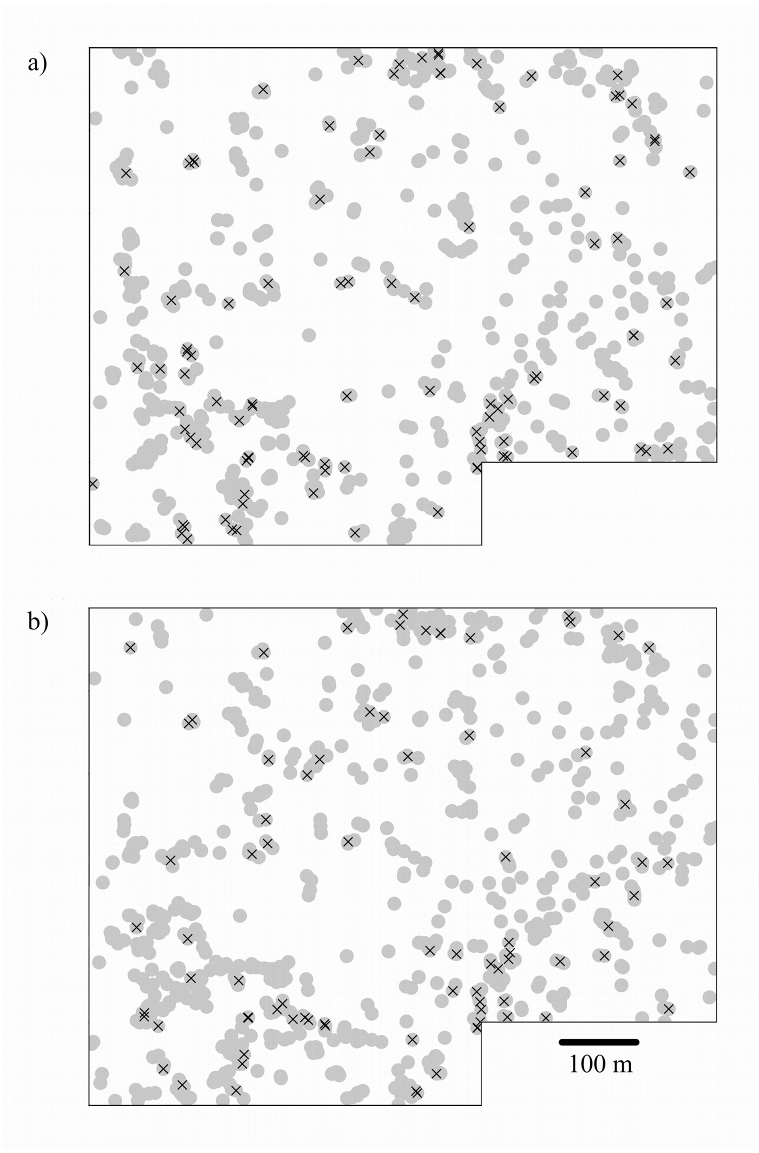


Figure VI.10. Distributions of ant nests in a 45 ha plot, represented as 17 m diameter gray circles, along with X's marking the locations where at least one neighboring coffee bush contained green coffee scale insects in a) 2009 and b) 2010. Note that the concentrations of scale insects tend to occur in the same nest clusters (defined by gray circles that touch or overlap one another) from year to year. The distances between the locations of scales in 2010 and the nearest scales in 2009 are significantly less than would be expected by chance ($p<0.0001$ using a Monte Carlo method with 10,000 repeats wherein the 2010 scales were randomly allocated to ant nests and the average distance to the nearest 2009 neighbor was calculated), indicating that the clusters of scales in 2010 generally occur near 2009 clusters, which is consistent with the persistence of the scale insects as a source/sink population.

Conclusions

Under a wide variety of scenarios, the self-organization of biological habitats may result in a distribution of habitat patch sizes that lacks a central tendency, and may thus be approximated by a power law, or some similar function. Given this habitat construction, the resulting populations that live in those habitat patches may exist as either a source/sink population or a metapopulation, conditioned not only on the migration and persistence qualities of the population itself, but also on the underlying distribution of the self-organized habitat patches. This framing of population spatial structure redirects the typical focus from one of migration/extinction dynamics only, to one that asks how the structure of the underlying habitats codetermines (along with the migration/extinction characteristics) the nature of population structure (whether the population exists as a source/sink population or a metapopulation).

Using this framework it seems to be the case that the ant, *Azteca instabilis*, which nests in shade trees in coffee plantations in southern Mexico, forms the underlying habitat structure that determines the fact that the associated green coffee scale, *Coccus viridis*, exists as a source/sink population.

In addition to the obvious implications for theoretical ecology, these results command attention from ecosystem managers of various persuasions. For example, in conservation planning, political exigencies frequently determine the size distributions of natural habitat preserves within a hostile matrix. Species of conservation interest living in these habitat fragments may exist as either source/sink populations or metapopulations, depending not only on the migration/extinction potential inherent in the species, but also on the underlying distribution of the habitat sizes, with concomitant management challenges for conservation planners. For instance, improving the conditions of migration might lead to a loss of a source/sink structure and extinction of the population, before the metapopulational structure can be realized (see curve a or d in Fig. VI.1), a counterintuitive result that can be readily understood in the framework of habitat patch size distributions and the metapopulation-source/sink continuum. This also casts the classical Single Large or Several Small (SLOSS) debate in a new light, suggesting that under some conditions intermediate states between these two extremes may maximize risk of extinction of the population.

Chapter VII

Detection of imminent, non-catastrophic regime shifts

The concept of a regime shift, in which an ecosystem changes rapidly from one state to a qualitatively different state, has gained a certain prominence in the context of anthropogenic climate change. Climate scientists hypothesize that there exist thresholds of atmospheric greenhouse gas levels at which very rapid changes in large-scale environmental conditions will occur. Examples of such scenarios include the collapse of the Atlantic thermohaline circulation and the disappearance of the Greenland Ice Sheet (Lenton et al. 2008).

Regime shifts also occur at much smaller scales, and are of particular concern when they involve a transition from a desirable to an undesirable state, such as the cultural eutrophication of lakes (Carpenter 2005, Scheffer and Nes 2007), desertification (Kéfi et al. 2007), and the collapse of fish stocks (Daskalov et al. 2007). Such shifts pose a challenge for the management of ecosystems, as the rapidity of the transitions makes it difficult – if not impossible – to arrest them once they have begun. Consequently, there has been much interest recently in developing early warning signals, or leading indicators, to detect imminent regime shifts far enough in advance to enable prevention (Scheffer et al. 2009). The proposed leading indicators typically depend on a phenomenon termed "critical slowing down," in which the dynamics of a system on the verge of a transition are predicted to slow down in a characteristic way (Strogatz 1994), leading to detectable statistical signals in the temporal and/or spatial dynamics of the system.

Critical slowing down is typically conceptualized using the metaphor of a ball in a cup. The cup represents a basin of attraction that tends to drive the state of the system, represented by the ball, to a particular equilibrial condition. When the system is far from a regime shift, the sides of the cup are very steep, and the ball will rapidly return to the center of the basin of attraction following any perturbation. In contrast, during an incipient regime shift, the system can be conceived of as a ball in a cup with very shallow sides. When the ball is displaced by perturbations, the shallow sides only weakly draw the ball back to the center of the cup, leading to a much slower rate of recovery – a slowing down of the dynamics. A regime shift occurs when the sides of the cup are shallow enough, or the perturbation is large enough, to knock the ball into a neighboring basin of attraction.

In this scenario, the slowing down of dynamics leads to both temporal and spatial autocorrelations. When dynamics are slow, the state of the system at any point in time is likely to be similar to what it was a short time before (temporal autocorrelation). Likewise, critical slowing down will tend to cause points in close spatial proximity to be in similar states, provided that there is sufficient dispersal between sites. With coupling via dispersal, a site that has been displaced from equilibrium by a perturbation will influence its neighboring sites by sending (or failing to send) propagules, thereby displacing the neighboring sites in the same direction. When there is only a weak basin of attraction, i.e., during critical slowing down, the influence of dispersal from neighboring sites will dominate over a site's own internal dynamics, causing neighboring sites to be significantly more similar than distant sites (spatial autocorrelation). Both temporal and spatial autocorrelation have been proposed as leading indicators of regime change (Wissel 1984, Dakos et al. 2009).

A second class of proposed leading indicators relies on changes in the spatial variance and skew of a system property of interest, e.g., in the spatial variance and skew of the abundance of a particular organism of interest. When a system is firmly embedded in a basin of attraction, all of the sites' states will tend to be tightly centered on the attractor. Therefore, a histogram of the sites will exhibit low variance and low skew. As the system moves towards a regime change, sites will tend to be less strongly drawn to the original attractor; at the same time, the influence of the alternative basin of attraction will begin to significantly affect some sites. Together, these two tendencies result in an increase in both the spatial variance and skew prior to a regime shift. Specifically, a peak in the skewness coupled with a continued increase in variance is proposed as a general indicator of regime change (Guttal and Jayaprakash 2008, Guttal and Jayaprakash 2009).

These indicators have generally been developed for systems that exhibit catastrophic thresholds, most commonly associated with fold bifurcations. However, whether these indicators will also prove effective for non-catastrophic thresholds is an open question (Scheffer et al. 2009). Near a catastrophic, fold-bifurcation threshold, there exist multiple equilibria, and the equilibrium that the system resides at depends on the path that the system took to reach the current state, i.e., there is hysteresis. This type of bifurcation with hysteresis is exemplified by the collapse of semi-arid vegetation as a result of a drying climate (Rietkerk et al. 2004). As dryness increases, the system crosses a threshold at which the vegetation suddenly collapses and a barren desert is formed; however, recovery of the vegetated state requires that dryness decrease well below the collapse threshold. Near the collapse threshold, whether the system resides at the vegetated equilibrium or the barren equilibrium depends on whether the system is approaching the threshold from the direction of increasing or decreasing dryness. A non-catastrophic threshold, in contrast, is characterized by a sudden change in the system state in response to a small change in a forcing parameter, but without the discontinuity and hysteresis of a fold bifurcation.

Hastings and Wysham (2010) offer a counter to the view that general leading indicators can be developed. The ball and cup metaphor and the proposed leading indicators that follow from this conceptualization depend on the system having a smooth potential, without underlying complex dynamics such as period doubling cascades to chaos. Therefore, they argue, for a large set of real ecological systems without smooth potentials, we should not expect to observe the proposed leading indicators prior to a regime shift. Agroecosystems, in particular, are likely to undergo regime shifts without detectable advance warning (Vandermeer 2011).

In the present study, the potential for regime change, in the form of a collapse of the population of a fungal biocontrol in a coffee agroecosystem, is tested using a spatially-explicit, stochastic simulation model. The fungus, *Lecanicillium lecanii*

(Zimmerman) Zare and Gams, is an entomopathogen and mycoparasite that provides an essential ecosystem service of pest control in coffee farms. The goals of this study are twofold: first, to use this model, which incorporates essential components of the known natural history of the fungus, to predict whether sudden regime change could occur in this system in response to small changes in the primary environmental parameters of the model; and second, to determine if the proposed leading indicators can be used to detect imminent regime shifts in this model.

These proximate goals are motivated by the immense practical utility that reliable detection of imminent regime shifts in agroecosystems could provide. For example, if a regime shift involves moving from a regime in which autonomous pest control is maintained to a regime in which that control is lost, detection of a regime shift could mean the prevention of a catastrophic pest outbreak or development of a chronic pest problem. More concretely, an early warning signal predicting the loss of *L. lecanii* from the coffee agroecosystem could allow managers to adjust their management activities before the population collapses, thereby maintaining *C. viridis* or *H. vastatrix* below pest status. Because of the centrality of ecosystem services to an agroecological management approach, the ability to predict major changes in the ecosystem far enough in advance to take ameliorative action would be invaluable.

A second motivation is to add to the existing models that have been used to test leading indicators. As is appropriate during the initial stages of developing a body of theory, the models used to date have tended to be formulated to favor the detection of regime shifts. Therefore, there is a strong need to continue accumulating a catalog of case studies using biologically realistic models and data from real systems in order to determine the practical potential of these indicators.

Methods

The study system

The study system upon which the model is based is comprised of *L. lecanii* and its primary host, the green coffee scale *Coccus viridis* Green (Hemiptera: Coccidae)

(González et al. 1995). The study site is in Finca Irlanda, an organic coffee agroecosystem located in the southeast of the state of Chiapas, Mexico. In addition to its potential role as a biological control of *C. viridis*, *L. lecanii* is known to attack coffee rust, *Hemileia vastatrix*, a potentially devastating disease (Moricca and Ragazzi 2008, Vandermeer et al. 2009, Jackson et al. 2012).

The ant *Azteca instabilis*, which tends *C. viridis* in a classic ant-hemipteran mutualism, is a keystone species that structures many of the relevant ecological interactions in this system (Vandermeer et al. 2010a). In exchange for a carbohydrate-rich excretion generated by the scales, *A. instabilis* protects *C. viridis* from its predators and parasitoids. In the presence of *A. instabilis*, *C. viridis* populations can grow to hundreds of individuals per coffee plant. These large populations of scale insects provide resources for a number of associated organisms, including *L. lecanii*. Local epizootics of *L. lecanii* frequently result in nearly 100% mortality of the scale insects tended by a given ant colony (Jackson et al. 2009).

The prevalence and distribution of *L. lecanii*, and hence its potential to maintain control of *C. viridis* and *H. vastatrix* within the coffee farm, may depend on a number of factors related to management practices. First, the ants nest in trees that are planted by farmers to shade the coffee plants below, and therefore the locations of concentrations of scale insects – the hosts of the fungus – are determined in part by the locations of the shade trees. The density of the shade tree canopy also influences the intensity of ultraviolet radiation that reaches the soil, which may be a determinant of fungal spore survival (Paul and Gwynn-Jones 2003); the shade trees are periodically pruned by the farmers, so management practices likely have a direct effect on the mortality rate of spores in the soil, which has been shown to be an important environmental reservoir of *L. lecanii* (Jackson et al. In press).

Environmental factors may also play an important role in the epizootiology of *L. lecanii*. Pronounced wet and dry seasons are a key climatic feature of the Soconusco region of Chiapas. During the wet season, which lasts for approximately 6 months, there is rain virtually every day that typically lasts from mid-afternoon through the night. In the dry season, in contrast, rain is very infrequent, with most days being sunny, warm, and dry. The intensity of the rainy season is potentially important because rain splash has been shown to be a mechanism for translocation of spores from the soil onto susceptible scale insects (Jackson et al. In press). Therefore, the dispersal of *L. lecanii* is heavily dependent on rainfall.

The spatially explicit, stochastic model

The core system of equations upon which the model (Appendix C) is based is equivalent to Hochberg's reservoir model (Hochberg 1989), with two changes. First, the host population dynamics are density dependent, i.e., the growth rate of the host population decreases as it approaches a carrying capacity, *K*, as a result of influences unrelated to the pathogen. Second, infected individuals do not reproduce. This latter assumption is appropriate for the specific insect host-fungal entomopathogen system that I consider in the present paper, and is also appropriate for a number of other insect diseases (Fuxa and Tanada 1987).

To add explicit spatial structure to the model, multiple instantiations of the system of differential equations shown below are embedded in a two-dimensional, continuous-space arena (Figure VII.1). A specified number of ant nest sites are distributed randomly within the arena, with an instance of the system of equations placed at each site.

$$\frac{dS_i}{dt} = rS_i\left(1 - \frac{S_i}{K}\right) - \beta S_i\left(W_i + \sum_{j=1}^{M} \frac{\alpha\mu W_j}{\exp^{\delta d_{i,j}}}\right) \tag{1}$$

$$\frac{dI_i}{dt} = \beta S_i\left(W_i + \sum_{j=1}^{M} \frac{\alpha\mu W_j}{\exp^{\delta d_{i,j}}}\right) - \sigma I_i \tag{2}$$

$$\frac{dW_i}{dt} = \sigma\theta_1 I_i - (\mu + \lambda)W_i + \nu Q_i \tag{3}$$

$$\frac{dQ_i}{dt} = \sigma\theta_2 I_i - (\rho + \nu)Q_i + \lambda W_i \tag{4}$$

where S_i, I_i, W_i, and Q_i are the susceptible hosts, infected hosts, infectious pathogens, and latent pathogens at site i, respectively; r is the intrinsic growth rate of the host population; K is the carrying capacity; and β is the transmission rate. Infected individuals, I_i, are removed at rate σ, converted into infectious biomass at rate $\sigma\theta_1$ and converted into latent pathogen biomass at rate $\sigma\theta_2$. The latent pathogen biomass, Q_i, represents the environmental reservoir.

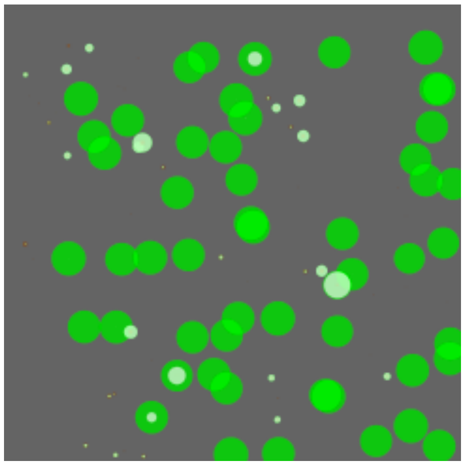


Figure VII.1. Snapshot of the stochastic, spatially-explicit model. Circles are locations of *A. instabilis* nests (sites). Green circles are proportional to the number of healthy *C. viridis* (S_i). White circles are proportional to the number of infected individuals (I_i).

The infectious and latent pathogens undergo two processes: translocation between the latent and infectious classes, and mortality. Translocation from the latent class to the infectious class and vice versa occur at rates v and λ, respectively. Mortality (removal) rates are μ for the infectious class and ρ for the latent class.

The individual sites are linked with the other sites by dispersal of infectious pathogens. Infectious pathogens can disperse from any of the M other sites in the arena. The rate of dispersing pathogens is a fraction, α, of the infectious pathogens removed at rate μ by either death or dispersal. The rate of dispersal from site j to site i is assumed to fall off exponentially as a function of the distance between sites, $d_{i,j}$, with a decay constant δ.

Seasonality is implemented based on the following assumptions: First, there are two distinct seasons, one in which the host is actively reproducing, and another in which it is quiescent. Second, at the beginning of the active season (following the dormant season) the host population is reset to an initial value that is independent of the size of the host population in the previous active season. In the study system, migration of the scale insects from areas outside of the location of an epizootic ensures that there will be an initial, small population of scales at the beginning of the active season even if the scales in a location were completely exterminated by an epizootic in the previous active season. Third, a portion θ_2 of infected individuals present at the end of the active season is converted into latent pathogens at the end of the active season. Fourth, there are no infectious individuals or infectious pathogens at the beginning of the active season, i.e., I and W are reset to zero at the beginning of each active season. Finally, the mortality rate of the latent pathogens, ρ, is the same during the active and dormant seasons.

Given these assumptions, it is possible to model the dormant season using a simple discrete-time map from the end of one active season to the beginning of the next active season. At the beginning of each active season, $S = S_0$, $I = 0$, $W = 0$, and $Q = (Q_a + \theta_2 I_a)\exp(-\rho T_d)$, where Q_a and I_a are the latent pathogens and infected individuals at the end of the previous active season, respectively; T_d is the length of the dormant season; and θ_2 and ρ are the conversion and mortality rate parameters described previously.

The active-season dynamics of the model described above were implemented using a modified version of Gillespie's stochastic simulation algorithm known as the optimized τ-leap method (Cao et al. 2006, Pineda-Krch 2008). In brief, the original formulation of Gillespie's algorithm, the direct method, involves first defining discrete events that can occur in the model, such as an infection event or a death event. The rates of all possible events, as defined by the model parameters and the current state of the system, are then used to calculate a distribution of times between events. At each step, the time to the next event is drawn from this distribution. The identity of the next event is determined probabilistically based on the relative rates of the different events. In this manner, the model state is advanced through time by repeatedly drawing time steps and event identities from distributions based on the rates of the various events.

Gillespie's direct method becomes very computationally expensive as population sizes increase and the time between events consequently becomes very small. The τ-leap method was developed to address this issue. It involves defining the time between events, τ, a priori and then calculating the number of occurrences of all of the different events during each time step. The number of occurrences of each event is based on the current firing rates of the events, as determined by the parameter values and the current state of the system. The challenge with this method is choosing a τ that is large enough to provide significant computational savings while not causing any computational anomalies. For example, if τ is too large, populations could fall below zero due to the number of death events exceeding the size of the population at the beginning of the time step; a smaller time step would allow the death rate to be adjusted as the population decreases, causing the population to smoothly approach zero without overshooting, but this accuracy comes at the expense of speed.

The difficulty of choosing a time step a priori that would strike a good balance between speed and accuracy under all conditions led to the development of the optimized τ-leap method. This method uses a combination of Gillespie's direct method; the explicit τ-leap method with a variable τ that is modified in real time based on the state of the system; and judicious execution of "critical" events, i.e., events that run the risk of driving any state to a negative value. The particular method or combination of methods used in a given cycle and the length of the time step (for cycles in which the τ-leap method is employed) are determined adaptively to optimize efficiency while preventing any processes from being driven below zero (Cao et al. 2006, Pineda-Krch 2008).

Parameter values

Default parameter values were derived using a combination of field data from the *L. lecanii-C. viridis* system and biologically reasonable estimates (Table VII.1). Data on the abundance of scale insects and the prevalence of *L. lecanii* on individual coffee plants over time are available, so it was possible to calculate estimates of the initial scale insect population size, growth rate, and carrying capacity. Data were unavailable for most of the parameters related to the fungus, including spore dispersal, translocation, and survival.

Only a very rough estimate of the transmission rate was possible. However, given that the goal of this investigation was to reveal possible qualitative outcomes, and not to provide accurate prediction, the emphasis was on choosing parameters that could generate the qualitative dynamics that are observed in the system, and not on precise parameter estimation.

Parameter	**Description**	**Value**
S_0	Initial number of susceptible scales	50*
I_0	Initial number of infected scales	10†
W_0	Initial biomass of infectious spores	10†
Q_0	Initial biomass of latent spores	300†
r	Intrinsic growth rate of scale population	0.0668*
K	Carrying capacity of scale population	1100*
β	Transmission rate	0.01*
σ	Removal rate of infected scales	0.07†
θ_1	Conversion rate of infected scales to infectious spores	0.5†
θ_2	Conversion rate of infected scales to latent spores	0.05†
μ	Rate of removal of infectious spores	0.1†
λ	Infectious to latent translocation rate	0.05†
v	Latent to infectious translocation rate	0.01†
ρ	Latent spore mortality rate	0.012†
α	Fraction of removed infectious spores that disperse	0.1†
δ	Dispersal kernel decay constant	0.2†
T_a	Length of active season (days)	183*
T_d	Length of dormant season (days)	182*
X	Width of arena (m)	243‡
Y	Height of arena (m)	243‡
sites	Number of sites	100‡

* Estimated from field data

† No field data available; based on biological plausibility and model behavior

‡ Chosen to match the ant nest density observed in the field and for computational tractability

Table VII.1. Model parameters and default values.

Calculation of leading indicators

Three of the leading indicators proposed in the literature were used to detect the onset of regime change: spatial autocorrelation, skewness, and variance. All of these indicators were calculated using the number of infected individuals, I_i, as these are the data that would be most readily obtainable from field surveys.

Spatial autocorrelation was calculated using Moran's coefficient (Legendre and Fortin 1989):

$$C(d) = \frac{n \sum_{i=1}^{n} \sum_{j=1}^{n} \omega_{i,j} (I_i - \overline{I})(I_j - \overline{I})}{\Omega \sum_{j=1}^{n} (I_{i,j} - \overline{I})^2} \tag{5}$$

where d is a distance class; $\omega_{i,j}$ is a weight that is 1 if the distance between sites i and j lies in distance class d and 0 otherwise; Ω is the total number of pairs of sites that fall into distance class d; and n is the number of neighboring sites. Moran's coefficient was calculated for all distance classes ranging from 5 to 250 m, with a bin width of 5 m. The correlation length, Ψ, which is an estimate of the distance over which sites are correlated, was then estimated by an exponential fit, $\exp(-d/\Psi)$ to $C(d)$ (Solé and Bascompte 2006). The correlation length was calculated for each day during the wet season, as well as for the average value of C during the entire wet season, or $\overline{C}(d)$.

Spatial variance, *Var(I)*, and skewness, *Skew(I)*, were calculated based on the distribution of I_i for all of the sites in the arena (Guttal and Jayaprakash 2009).

Results

Three scenarios leading to apparent regime shifts in which the population of *L. lecanii* collapsed throughout the arena were identified. All of these regime shifts are non-catastrophic, without any apparent hysteresis, i.e., they are not associated with a fold bifurcation. The first two scenarios are triggered by changes to the translocation rate, *v* (Figure VII.2). If the translocation rate is decreased from the default value of 0.01, the median fraction of sites with at least one infected individual increases. Beyond a threshold value of approximately 0.0025, however, the median fraction of infected sites rapidly falls, eventually leading to the complete extinction of the fungus from the system as v is decreased further. In the second scenario, when v is increased above the default value there is also a rapid decrease in the prevalence of *L. lecanii* in the plot, although the decrease is not as precipitous (Figure VII.2).

The third scenario under which a sudden regime shift is observed involves an increase in the latent spore mortality rate, ρ. As ρ is increased beyond the default value of 0.012, there is a threshold at which the site-wide prevalence of *L. lecanii* falls rapidly (Figure VII.3). If ρ is increased even further above this threshold, the fungus eventually dies out completely.

The dynamics of the system as the translocation rate is slowly decreased across the regime shift threshold are shown in Figure VII.4a. As v nears the threshold, the total population of infected scales decreases noticeably. Both the variance and the skewness also change markedly as the threshold is approached. There is no discernible trend in the correlation length, Ψ. In response to an increasing translocation rate, there are no apparent signals of a regime shift in any of the leading indicators (Figure VII.4b); the only obvious change in the data is the gradual reduction in the number of infected individuals.

For a scenario in which the mortality rate of latent spores, ρ, is increased across the regime shift threshold, there is a general trend towards a lower number of infected sites. There is also a decrease in the spatial skew, and an increase in the spatial variance (Figure VII.5); however, neither of these two leading indicators exhibits a noticeable, rapid change until after the collapse has already taken place. The correlation length also shows no apparent signal of the imminent regime shift.

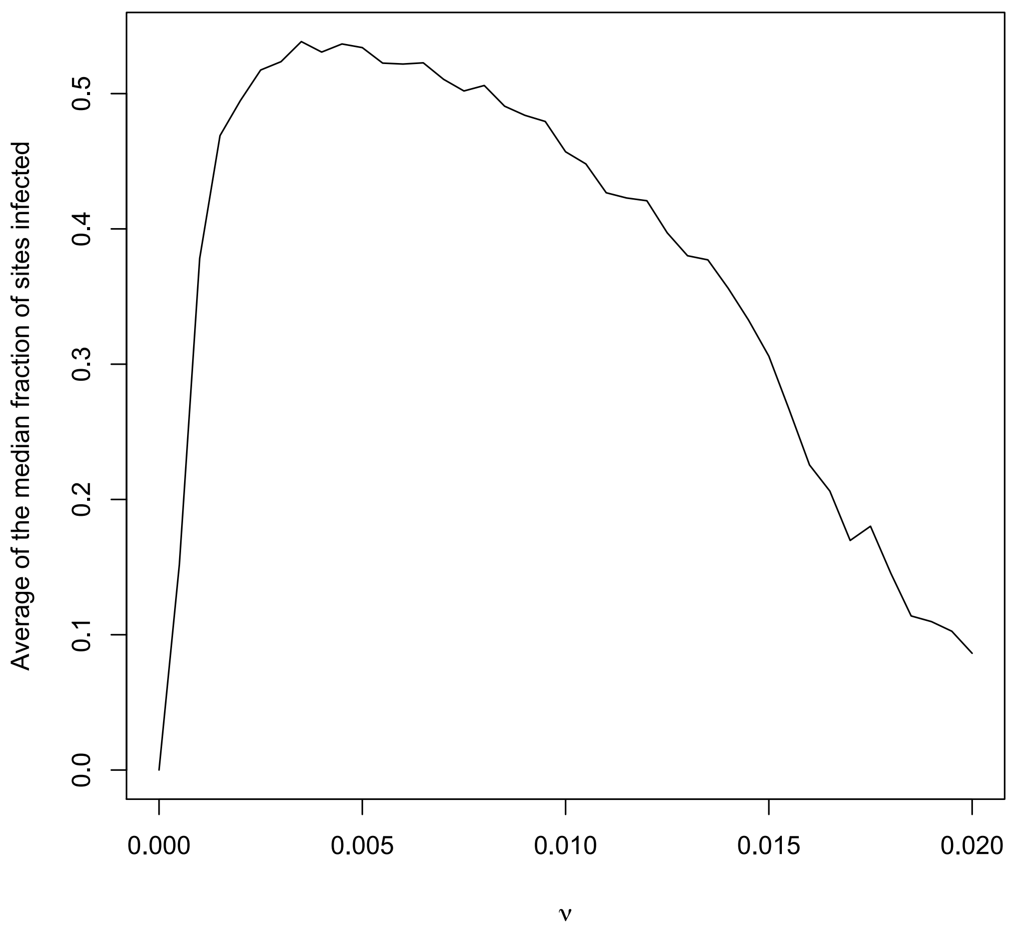


Figure VII.2. Median fraction of sites with infected individuals, averaged over 100 realizations of the model, as a function of the rate of translocation from the environmental reservoir to the infectious class (*ν*). The model was run for 10,000 active season days, and the final 5,000 active season days of each run were extracted for analysis.

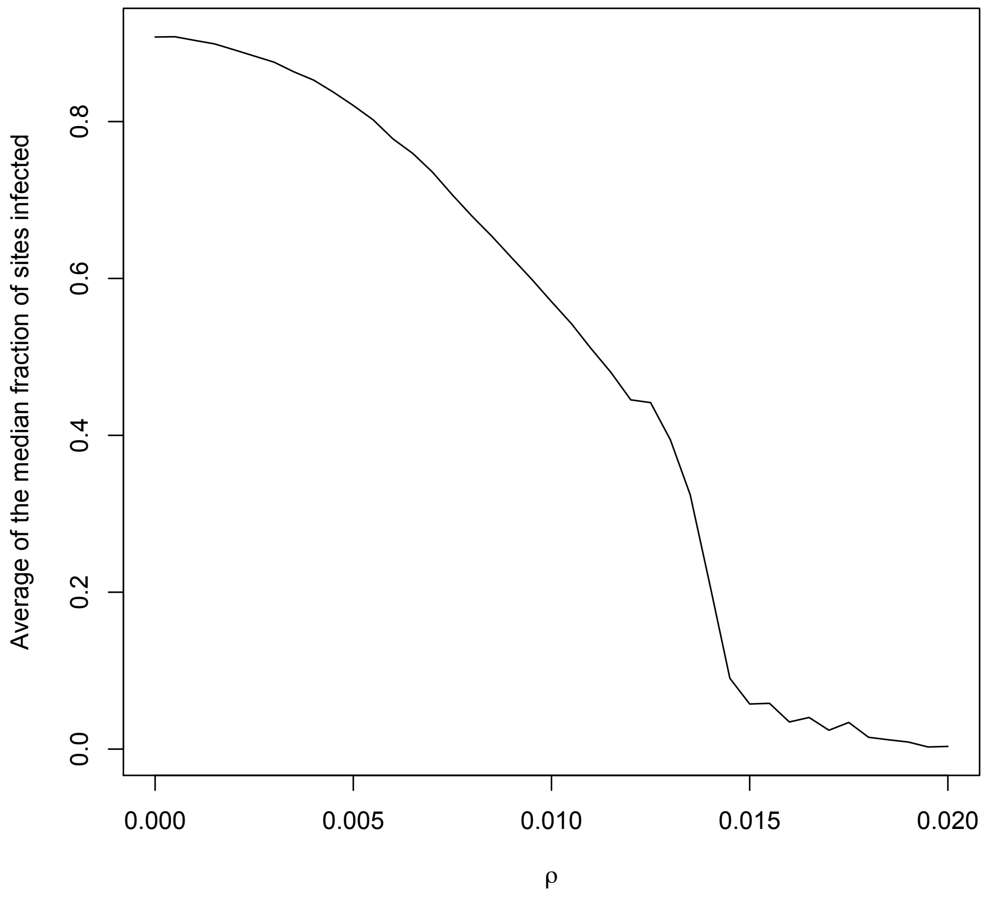


Figure VII.3. Median number of sites with infected individuals, averaged over 100 realizations of the model, as a function of the mortality rate of latent pathogens, ρ. Dashed line is without dispersal ($\alpha = 0$) and solid line is with dispersal ($\alpha = 0.1$). The model was run for 10,000 active season days, and the final 5,000 active season days of each run were extracted for analysis.

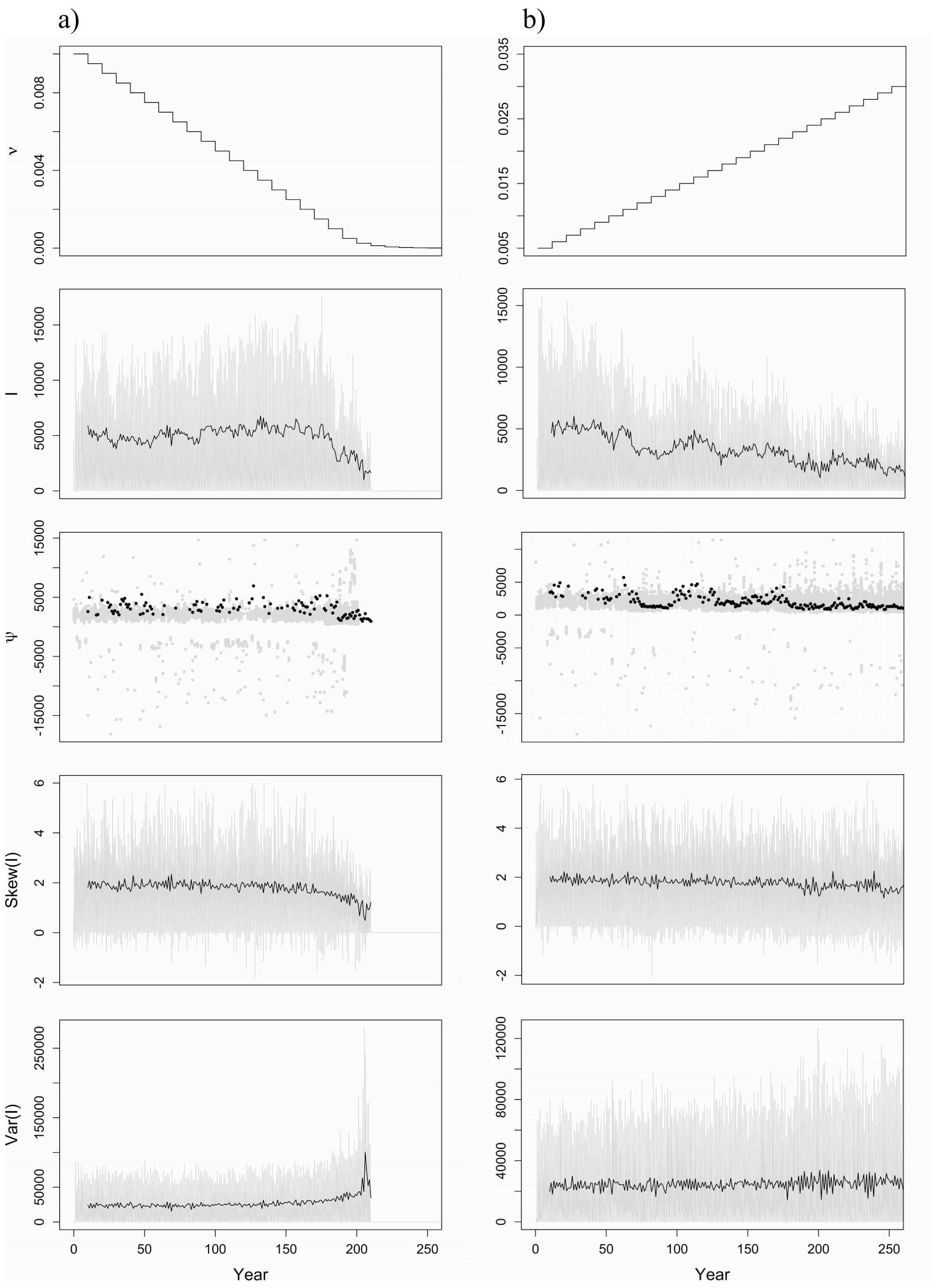


Figure VII.4. Translocation rate (*v*), total number of infecteds, correlation length (*Ψ*), skewness, and variance for two regime shift scenarios. a) Decrease of *v* across the lower regime shift threshold and b) increase of *v* across the upper regime shift threshold. Gray lines are daily values; black lines are averages across a single wet season. Gray points are for fits to Moran's coefficients for daily snapshots of I_i; black points are fits to Moran's coefficients averaged across a single wet season. Correlation lengths are only shown when the exponential fit, exp(-*d*/*Ψ*), was significant at $P < 0$ .05.

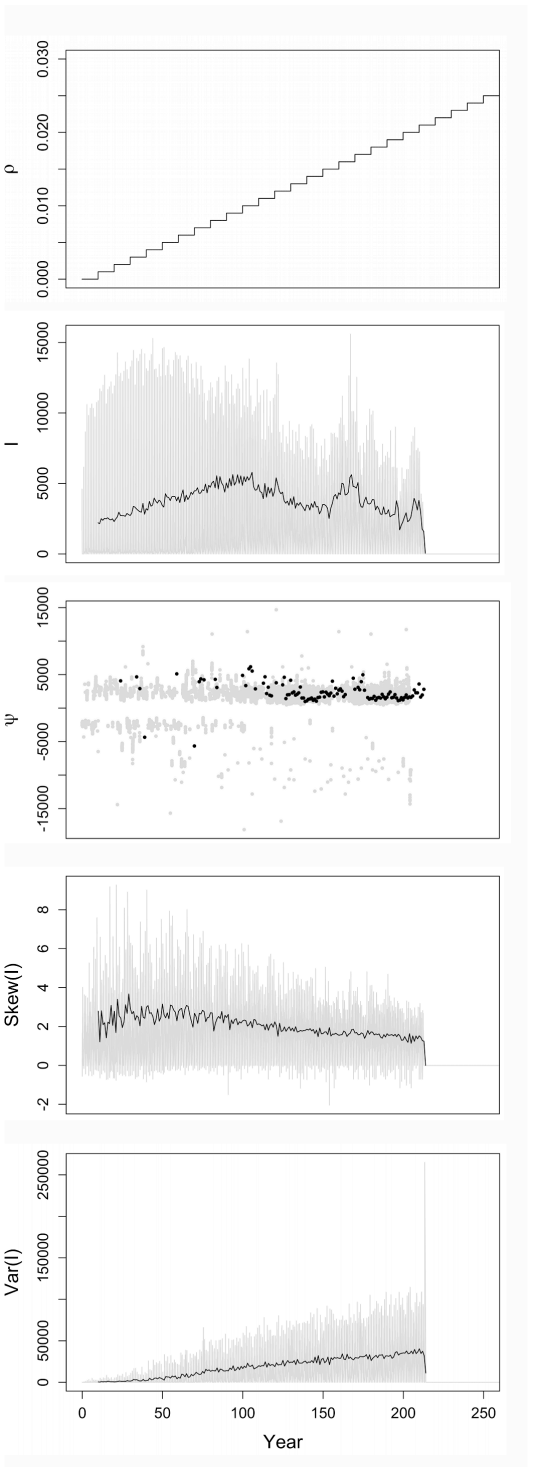


Figure VII.5. Mortality rate of latent spores (*ρ*), total number of infecteds, correlation length (*Ψ*), skewness, and variance for a scenario in which *ρ* was increased across a regime shift threshold. Gray lines are daily values; black lines are averages across a single wet season. Gray points are for fits to Moran's coefficients for daily snapshots of I_i; black points are fits to Moran's coefficients averaged across a single wet season. Correlation lengths are only shown when the exponential fit, exp(-*d*/*Ψ*), was significant at *P* <0 .05.

Discussion

The existence of regime shifts in this agroecosystem model underscores the potential importance of nonlinearities in this particular agroecosystem, and in managed systems in general. It also highlights the need to develop leading indicators that can reliably predict the onset of a regime shift. However, the general failure of the proposed leading indicators – spatial autocorrelation, spatial variance, and spatial skewness – to unambiguously signal the onset of regime shifts observed in this model suggests that the leading indicators developed to date for catastrophic regime shifts may be less effective for non-catastrophic regime shifts, and emphasizes that much work still needs to be done to develop signals of impending regime change that are practicable under real-world management scenarios for systems that exhibit this class of regime shift.

In the context of the example study system, the humped relationship between prevalence and the latent-to-infectious translocation rate illustrates how nonlinearities can have important implications for the maintenance of biological control. In the *L. lecanii*-*C. viridis* system, an environmental reservoir is in the soil, and latent-to-infectious translocation occurs by rain splash (Jackson et al. In press), so the latent-to-infectious translocation rate will be a function of rainfall intensity. Therefore, intermediate rainfall intensity could be expected to maximize the effectiveness of biological control of the scale insect (*C. viridis*) by the fungus. The possibility that intermediate rainfall intensity could maximize biological control has not been considered in the literature related to this system, which focuses exclusively on the positive effects that elevated relative humidity has on germination of *L. lecanii* and the initiation of epizootics (Reddy and Bhat 1989). Ignoring the effects of rainfall on the environmental reservoir could lead to qualitative predictions that are counter to what may actually occur if rainfall were to change.

These results also point to the potential for management decisions to have unexpected effects on biological control. Pruning shade trees in order to increase photosynthetic activity is a common technique for increasing coffee yield (Staver et al. 2001). However, this practice is also likely to significantly increase the intensity of ultraviolet radiation reaching the soil. Ultraviolet radiation is known to have a strong negative effect on the survival of fungal conidia (Paul and Gwynn-Jones 2003), and could therefore increase the mortality rate of spores in the environmental reservoir, i.e., ρ. This could potentially lead to a drastic reduction in the prevalence of *L. lecanii* (Figure VII.3) and a subsequent loss of the ecosystem service of pest control. Although further research would be necessary to determine whether the real system resides in a region of parameter space where small changes in ρ would lead to major changes in prevalence, these results suggest that it would be prudent for managers to monitor the response of the fungus as shade levels are manipulated.

There are a number of factors that contributed to the failure of the leading indicators to unequivocally signal the onset of the observed regime shifts. The primary factor may be the relative gradualness of the non-catastrophic transitions observed in this system compared to the catastrophic transitions that the leading indicators have generally been applied to. Intuitively, critical slowing down should be less severe for non-catastrophic thresholds, and therefore the associated signals should be less pronounced, making them more difficult to distinguish from noise. In comparing the tendency for the leading indicators to change as the system approaches the three regime shifts considered here (Figures VII.4 and VII.5), it appears that this intuition is borne out, with the change in the indicators seemingly correlated with the rapidity of the transition (Figures VII.2 and VII.3). Although none of the scenarios exhibited the diagnostic peak in skewness coupled with a continued increase in variance (Guttal and Jayaprakash 2009), the scenario with the most abrupt threshold exhibited a noticeable change in these parameters prior to the collapse (Figure VII.4a) while the scenario with the most gradual shift did not (Figure VII.4b).

In addition to the inherent difficulty of detecting the onset of non-catastrophic regime shifts, the lack of a signal of increased spatial autocorrelation is also likely due to the weak coupling of sites, which is a consequence of the shape and magnitude of the dispersal kernel as well as the relatively large and varied distances between sites. Both of these characteristics (low rates of dispersal and large, varied distances between sites) are likely to be common in real systems, which calls into question the potential utility of spatial autocorrelation as a leading indicator. Use of spatial autocorrelation as a leading indicator relies on dispersal becoming dominant as the internal dynamics of the sites slow down, but this may be unlikely to occur in systems that do not have the strong spatial coupling that is characteristic of the lattice-based models with which this leading indicator was originally developed (Dakos et al. 2009).

An important consideration if these indicators are to be successfully applied to real systems is how feasible it will be to collect the required data. The spatial and temporal resolutions of the sample data available in simulation models such as the one used in this and other theoretical studies on leading indicators far exceeds what could reasonably be collected in the field. For example, a more realistic scenario for the *L. lecanii-C. viridis* study system, given current technology and resources, would be for plot-wide surveys to be completed once per month. Given the amount of variation in the simulated daily values of the leading indicators (Figures VII.4 and VII.5), it is clear that substantial development of statistical methods to cope with noisy, incomplete, and infrequent data would be required before these leading indicators could be employed by managers.

Despite the challenges that remain, these results provide some hope that leading indicators could be used to augment the tools that are currently available to managers. Although an unequivocal signal of impending regime change was not detectable for any of the scenarios in this study, the rapid changes in variance and skewness associated with the most severe threshold (Figure VII.4a) could conceivably be used as a warning signal for this particular regime shift, albeit one without any precise information regarding the proximity of the threshold. The magnitude of the changes in variance and skewness also provide some hope that these signals would be detectable even in more realistic, data poor scenarios.

These results also provide another example of what is coming to be seen as a widespread and common phenomenon: rapid changes in the state of an ecosystem in response to small changes in environmental forcers. As ecosystem managers continue to confront the rapid and profound change that is the defining feature of the Anthropocene, effective methods to detect impending regime shifts could prove to be an invaluable tool for allocating limited management resources to the most urgent management concerns. However, it is important to note that even if leading indicators can be developed to detect the onset of certain regime shifts, it may be impossible to detect others (Hastings and Wysham 2010, Vandermeer 2011). Therefore, leading indicators may be useful for preventing some detectable regime shifts, but they cannot substitute for applying the precautionary principle in agroecosystem management. For while not all surprises are inevitable, surprise itself is.

Chapter VIII
Conclusion

The body of work represented by this dissertation should be seen as both encouraging and sobering. Encouraging, in that there is ample reason for hope that an agroecological approach can help to move agriculture beyond the production-versus-health trade-offs that inhere within the industrial agriculture paradigm. Sobering, in that moving from the preliminary, basic research contained herein to concrete, actionable management recommendations will require many, many dissertations worth of work. That modern agroecology is in many ways in its infancy compared to the relatively sophisticated, if misguided, state of conventional agronomy is as much a reflection of the immense resources that have been expended on the industrial agriculture agenda as of the intrinsic difficulty of agroecology. But with only 0.81% of agricultural land worldwide managed using certified organic methods (Willer and Kilcher 2010), and a smaller fraction still qualifying as agroecological, it is clear that proponents of agroecology have much work ahead of them to overcome the inertia of the food system (though it is important to note that this percentage does not include non-certified organic agriculture, which is difficult to quantify because much of it is grown, distributed, and consumed locally without passing through monitored market systems (Scialabba and Hattam 2002)).

In truth, there have already been great strides made in agroecology. For instance, a substantial amount of evidence showing the agricultural benefits of biodiversity has been accumulated (Jarvis et al. 2007), and enough evidence has been collected showing the biodiversity benefits of agroecology to make a strong case for it on conservation grounds (Bengtsson et al. 2005, Perfecto and Vandermeer 2008a). In terms of production, even without advancements beyond the current state of the art of sustainable farming, indications are that it would be possible to meet the food and fiber needs of a growing world population (Badgley et al. 2007, Badgley and Perfecto 2007). At the same time, the case against conventional agriculture becomes more damning by the day. As just a sampling of the damaging side effects associated with conventional agriculture that have come to light in the first few months of this year alone: a potential link to colony collapse disorder in bees (Henry et al. 2012, Whitehorn et al. 2012), long-term effects of prenatal pesticide exposure on reproductive function in boys (Wohlfahrt-Veje et al. 2012), and collapse of fisheries (Scholz et al. 2012). In short, with what we already know about agroecology and the status quo, there is ample reason to shift away from conventional agriculture.

So, what is preventing a wholesale switch to a less damaging approach to agriculture? Part of the answer can be found in the present research. In this dissertation, I have shown that 1) *L. lecanii* exerts a subtle, but statistically significant negative influence on *H. vastatrix*, 2) propagules of *L. lecanii* appear to exist in low, but detectable, levels throughout the coffee farm, 3) the initiation and subsequent spread of *L. lecanii* epizootics is accomplished through the joint actions of rain splash, the ant *A. instabilis*, and possibly other as-yet-unknown mechanisms, 4) it is likely that the *L. lecanii* population could collapse rather suddenly in response to changes in management and/or climatic conditions, and that a collapse could occur without any detectable advance warning, and 5) there is a strong potential for a cross fertilization of ideas and theory between agroecology and the rest of the science of ecology. Clearly, an appreciation for these results and the body of similarly complex results that has been generated by agroecologists over the years requires a level of tolerance for uncertainty, contingency, and complexity that a farmer accustomed to a "spray this to kill that" approach will be unlikely to possess. As stated in the National Research Council in their 1989 report on alternative agriculture, "Alternative farming practices typically require more information, trained labor, time, and management skills per unit of production than conventional farming" (1989). With farmers the world over facing an onslaught of economic challenges, this is not an easy program to sell, regardless of the potential advantages.

Although the challenges are daunting, it is heartening to remember that history is full of examples of seemingly insurmountable inertia being overcome. After all, at various times and places in human history, it was inconceivable that people with dark skin would cease to be treated like beasts of burden; that God would no longer personally appoint a despot to subjugate the people; that women would be considered to be full citizens; that love between two men or two women would be valued as much as love between a man and a woman; that the East would dominate the West, or the West the East; and that innumerable other apparently immutable conditions might change. With luck, and much hard work, an agriculture based on prudent cooperation with nature instead of physical and chemical violence against it will become the norm.

APPENDICES

Appendix A

Computer code for Chapter V: The evolution of imperfect prudence

Software specifications

Recursive Porous Agent Simulation Toolkit (Repast) 3.0

Class list

HostPathogen.java	146
Host.java	172
Pathogen.java	182
StatsDisplayObject.java	187
GUIModel.java	188
BatchModel.java	194
ModelParameters.java	197

Class details

HostPathogen.java

```
package hostPathogen_v7;

// A host-pathogen, probabilistic cellular automata model to investigate
// the evolution of prudence in hosts
import hostPathogen_v7.Host;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Vector;
import java.util.Collections;

import uchicago.src.sim.analysis.Plot;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.space.Object2DTorus;

public class HostPathogen extends ModelParameters
{

    // class variables
```

```
public static Plot transmissibilityGraph;
public static Plot virulenceGraph;
public static Plot growthGraph;
public static Plot numHostsGraph;
public static Plot numPathogensGraph;

// control variables
public boolean newMethod = true; // true = disable infectionTries method of pathogen spread

 // instance variables
 public ArrayList<Host> hostList = new ArrayList<Host>();
 public ArrayList<Pathogen> pathogenList = new ArrayList<Pathogen>();
 public ArrayList<Host> hostBirthList = new ArrayList<Host>();
 public ArrayList<Pathogen> pathogenBirthList = new ArrayList<Pathogen>();
 public ArrayList<Host> hostDeathList = new ArrayList<Host>();
 public ArrayList<Pathogen> pathogenDeathList = new ArrayList<Pathogen>();
 public int[][] occupancyMatrix;
 public int[] numHGP;
 public int[] numDescendents;
 public int[] ageDist;
 public int[] numDescendents2;
 public int[] ageDist2;

 public Schedule schedule;

 public int sizeX;
 public int sizeY;
 public int startNumHosts;
 public int numHosts;
 public int startNumPathogens;
 public int changeNumPathogens;
 public int numPathogens;
 public int graphUpdatePeriod;
 public int hostExecutionPeriod;
 public double probLongDistance;

 // PIP parameters
 public double challengerGrowthProb;
 public int challengerStartNum;
 public int challengeStartTime;
 public int challengeFreq;

 // defaults for the hosts and pathogens
 public double defaultGrowthProb;
 public double probGrowthMutate;
 public double growthMutation;
 public double defaultNaturalDeathZero;
 public double defaultNaturalDeathSlope;
 public double defaultTransmissibility;
 public double defaultVirulence;
 public double probTransMutate;
 public double transMutation;
 public double probVirulenceMutate;
 public double virulenceMutation;

 // stats
 public double minTransmissibility;
 public double avgTransmissibility;
public double maxTransmissibility;
public double minVirulence;
public double avgVirulence;
public double maxVirulence;
public double minGrowthProb;
public double avgGrowthProb;
public double maxGrowthProb;

public int numNaturalDeaths;
public int numDiseaseDeaths;
public int hostTime;

public int writeTime;
public int enableStepReport;

// environs densities
public double P_pp;
public double P_mm;
public double P_pm;
public double P_mp;
public double P_p0;
public double P_0p;
public double P_m0;
public double P_0m;
public double P_00;
public int sum_P;

public Object2DTorus world; // 2D class from Repast
public Object2DTorus offspringWorld;
public Object2DTorus descendentsWorld;

// ///////////////////////////////////////////////////////////////////////
// addModelSpecificParameters
// add alias and long name for Model parameters you want to set at run time
// the long name should be same as instance variable
//
// Note: the generic parameters from ModelParameters are already available.

@Override
public void addModelSpecificParameters()
{
    parametersMap.put("X", "sizeX");
    parametersMap.put("Y", "sizeY");
    parametersMap.put("sNH", "startNumHosts");
    parametersMap.put("sNP", "startNumPathogens");
    parametersMap.put("gUP", "graphUpdatePeriod");
    parametersMap.put("pLD", "probLongDistance");
    parametersMap.put("hEP", "hostExecutionPeriod");
    parametersMap.put("dGP", "defaultGrowthProb");
    parametersMap.put("dNDZ", "defaultNaturalDeathZero");
    parametersMap.put("dNDS", "defaultNaturalDeathSlope");
    parametersMap.put("dTau", "defaultTransmissibility");
    parametersMap.put("dV", "defaultVirulence");
    parametersMap.put("pGM", "probGrowthMutate");
    parametersMap.put("gM", "growthMutation");
    parametersMap.put("pTM", "probTransMutate");
    parametersMap.put("tM", "transMutation");
    parametersMap.put("pVM", "probVirulenceMutate");
    parametersMap.put("vM", "virulenceMutation");
    parametersMap.put("cGP", "challengerGrowthProb");
    parametersMap.put("cSN", "challengerStartNum");
    parametersMap.put("cST", "challengeStartTime");
    parametersMap.put("cFR", "challengeFreq");
    parametersMap.put("wT", "writeTime");
    parametersMap.put("eSR", "enableStepReport");
}

// control what appears in the repast parameter panel
@Override
public String[] getInitParam()
{
    String[] params =
    { "sizeX","sizeY", "startNumHosts", "startNumPathogens", "graphUpdatePeriod",
            "probLongDistance", "hostExecutionPeriod", "defaultGrowthProb", defaultNaturalDeathZero",
            "defaultNaturalDeathSlope", "defaultTransmissibility", "defaultVirulence",
            "probGrowthMutate", "growthMutation", "probTransMutate",
            "transMutation", "probVirulenceMutate", "virulenceMutation", "challengerGrowthProb",
            "challengerStartNum", "challengeStartTime", "challengeFreq", "writeTime",
            "enableStepReport"};
    return params;
}

// ////////////////////////////////////////////////////////////////////////////
// constructor, if needed.
public HostPathogen()
{

}

// ////////////////////////////////////////////////////////////////////////////
// setup
// set defaults after a run start or restart

@Override
public void setup()
{
    if (rDebug > 0)
        System.out.printf("==> setup...\n");
    schedule = null;
    System.gc();

    hostList = new ArrayList<Host>();
    pathogenList = new ArrayList<Pathogen>();

    sizeX = 100;
    sizeY = 100;

    occupancyMatrix = new int[sizeX][sizeY];
    numHGP = new int[101];
    numDescendents = new int[100];
    ageDist = new int[100];
    numDescendents2 = new int[100];
    ageDist2 = new int[100];

    startNumHosts = 500;
    startNumPathogens = 50;
    numPathogens = startNumPathogens;
    changeNumPathogens = 0;

    graphUpdatePeriod = 100000;

    hostExecutionPeriod = -999;

    probLongDistance = 0.0016;

    // PIP parameter defaults
    challengerGrowthProb = 0.2;
    challengerStartNum = 0;
    challengeStartTime = 100;
    challengeFreq = -999;

    // defaults for hosts and pathogens
    defaultGrowthProb = 0.6;
    probGrowthMutate = 0.15;
    defaultNaturalDeathZero= 0.2;
    defaultNaturalDeathSlope = 0;
    growthMutation = 0.01;
    defaultTransmissibility = 1;
    defaultVirulence = 1;
    probTransMutate = 0;
    transMutation = 0.005;
    probVirulenceMutate = 0;
    virulenceMutation = 0.01;

    numNaturalDeaths = 0;
    numDiseaseDeaths = 0;
    hostTime = 0;

    writeTime = 1000;
    enableStepReport = 0;

    super.setup(); // THIS SHOULD BE CALLED after setting defaults in
    // setup().
    schedule = new Schedule(1); // create AFTER calling super.setup()

    if (rDebug > 0)
        System.out.printf("\n<=== setup() done.\n");

}

// ////////////////////////////////////////////////////////////////////////
// buildModel
// We build the "conceptual" parts of the model.
// (vs the display parts, and the schedule)
//
// Create a 2D world, tell the organisms about it.
// Create organisms and add them to the lists.

public void buildModel()
{
    if (rDebug > 0)
        System.out.printf("==> buildModel...\n");

    // CALL FIRST -- defined in super class -- it starts RNG, etc
```

```
        buildModelStart();

        // tell the hosts and pathogens about "this"
        Host.setModel(this);
        Pathogen.setModel(this);

        // create the 2D world, tell the Host and Pathogen classes about it.
        world = new Object2DTorus(sizeX, sizeY);
        Host.setHostsWorld(world);
        Pathogen.setPathogensWorld(world);

        offspringWorld = new Object2DTorus(sizeX, sizeY);
        Host.setOffspringWorld(offspringWorld);

        descendentsWorld = new Object2DTorus(sizeX, sizeY);
        Host.setDescendentsWorld(descendentsWorld);

        // set the default parameters of the hosts and pathogens
        Host.setDefaultGrowthProb(defaultGrowthProb);
        Host.setDefaultNaturalDeathZero(defaultNaturalDeathZero);
        Host.setDefaultNaturalDeathSlope(defaultNaturalDeathSlope);
        Host.setProbGrowthMutate(probGrowthMutate);
        Host.setGrowthMutation(growthMutation);
        Pathogen.setDefaultTransmissibility(defaultTransmissibility);
        Pathogen.setDefaultVirulence(defaultVirulence);
        Pathogen.setProbTransMutate(probTransMutate);
        Pathogen.setTransMutation(transMutation);
        Pathogen.setProbVirulenceMutate(probVirulenceMutate);
        Pathogen.setVirulenceMutation(virulenceMutation);

        // create and scatter the initial hosts
        scatterRandomHosts();

        // create and scatter the initial pathogens
        infectHostsRandomly(startNumPathogens);

        // some post-load finishing touches
        startReportFile();

        // for the initial state, calculate these numbers, store in instance
        // variables
        // record some stats every step
        calcStatistics();

        // calls to process parameter changes and write the
        // initial state to the report file.
        applyAnyStoredChanges();
        stepReport();
        getReportFile().flush();
        getPlaintextReportFile().flush();

        if (rDebug > 0)
            System.out.printf("<== buildModel done.\n");
    }

    // Create a new Host with growthProb=gP and put it at x, y
    public void createNewHost(int x, int y, double gP)
    {
```

```
    Host aHost = new Host(x, y, gP);
    world.putObjectAt(x, y, aHost);
    setOccupancyMatrix(x,y,1);
    hostList.add(aHost);
}

// Create a new Pathogen and put it at x, y
public Pathogen createNewPathogen(int x, int y)
{
    Pathogen aPathogen = new Pathogen(x, y);
    pathogenList.add(aPathogen);
    return aPathogen;
}

// Add random hosts
public void scatterRandomHosts()
{

    int randomX, randomY;

    if (rDebug > 0)
        System.out.printf("==> scattering hosts...\n");

    // Check to see if there are too many hosts to fit on the grid
    if (startNumHosts>(sizeX*sizeY))
    {
        // fill entire grid with hosts and output warning message
        startNumHosts = sizeX*sizeY;
        System.out.println("Warning: startNumHosts is too big; I'll put a host in every cell");
    }

    for (int i = 0; i<startNumHosts; i++)
    {

        // lets find a random place that is unoccupied
        // This method of selecting random cells will get really slow
        // as the number of hosts approaches the number of cells in the arena
        do
        {
            randomX = getUniformIntFromTo(0, world.getSizeX()-1);
            randomY = getUniformIntFromTo(0, world.getSizeY()-1);
        } while (((Host)world.getObjectAt(randomX, randomY)) != null);

        createNewHost(randomX, randomY, defaultGrowthProb);

    }

    if (rDebug > 0)
        System.out.printf("==> ...done scattering hosts\n");
}

// Add random challengers
public void scatterChallengers()
{

    int randomX, randomY;
    int tempChallengerStartNum = challengerStartNum;
```

```
        if (rDebug > 0)
            System.out.printf("==> scattering challengers...\n");

        // Check to see if there are too many hosts to fit on the grid
        if (challengerStartNum>((sizeX-1)*(sizeY-1)-numHosts))
        {
            // fill entire grid with hosts and output warning message
            challengerStartNum = (sizeX-1)*(sizeY-1)-numHosts;
            System.out.println("Warning: challengerStartNum is too big; I'll put a host in every empty cell");
        }

        for (int i = 0; i<challengerStartNum; i++)
        {

            // let's find a random place that is unoccupied
            // This method of selecting random cells will get really slow
            // as the number of hosts approaches the number of free cells in the arena
            do
            {
                randomX = getUniformIntFromTo(1, world.getSizeX() - 1);
                randomY = getUniformIntFromTo(1, world.getSizeY() - 1);
            } while (((Host)world.getObjectAt(randomX, randomY)) != null);

            createNewHost(randomX, randomY, challengerGrowthProb);

        }

        // restore the specified challengerStartNum
        challengerStartNum = tempChallengerStartNum;

        if (rDebug > 0)
            System.out.printf("==> ...done scattering challengers\n");
    }

    // Randomly infect the hosts
    public void infectHostsRandomly(int numPathogens)
    {

        // Shuffle the list of hosts
        Collections.shuffle(hostList);

        // check to see if there are too many pathogens
        if (numPathogens > hostList.size())
        {
            System.out.println("Warning: number of new pathogens is too big; every host will be infected");
            numPathogens = hostList.size();
        }

        // infect the first numPathogens hosts in hostList
        for (int i = 0; i<numPathogens; i++)
        {
            hostList.get(i).setInfected(true);
            hostList.get(i).setPathogen(createNewPathogen(hostList.get(i).getX(), hostList.get(i).getY()));
        }
    }

    // ///////////////////////////////////////////////////////////////////////
    // step
```

```
// The top of the "conceptual" model's main dynamics
public void step()
{
    Vector<Host> neighbors;

    if (rDebug > 0)
        System.out.printf("==> CML step %.0f:\n", getTickCount());

    if((hostTime % writeTime) == 0)
    {
        writeState();
    }
    if(executeHost())
    {
        incrementHostTime();
        // loop through all the hosts to see if they get randomly infected.  For now,
        // the new pathogens will have parameter values equal to those of existing pathagens
        // chosen at random
        for (Host aHost : hostList)
        {
            if (getUniformDoubleFromTo(0, 1) < probLongDistance)
            {
                if(!pathogenList.isEmpty())
                {
                    // Shuffle the list of pathogens; the new pathogen will have parameter values
                    // from the first pathogen in this randomly ordered list
                    Collections.shuffle(pathogenList);

                    // add a new pathogen here with parameter values from the randomly-chosen pathogen
                    Pathogen aPathogen = new Pathogen(aHost.getX(), aHost.getY(),
                            pathogenList.get(0).getTransmissibility(), pathogenList.get(0).getVirulence());
                    addPathogenBirth(aPathogen);
                }
                else
                {
                    // add a new pathogen with the default parameter values
                    Pathogen aPathogen = new Pathogen(aHost.getX(), aHost.getY(),
                            defaultTransmissibility, defaultVirulence);
                    addPathogenBirth(aPathogen);
                }

            }
        }
    }

    // loop through all of the cells to see if organisms reproduce into them
    for (int i=0; i<sizeX; i++)
    {
        for (int j=0; j<sizeY; j++)
        {
            // get the neighbors
            neighbors = world.getVonNeumannNeighbors(i, j, false);
            for (Host aHost : neighbors)
            {
                aHost.reproduce(i, j);
            }
        }
    }
```

```
        // perform the organisms' steps
        for (Host aHost : hostList )
        {
            aHost.step();
        }

        for (Pathogen aPathogen : pathogenList)
        {
            aPathogen.step();
        }

        // process birth and death lists
        processHostBirthList();
        processPathogenBirthList();
        processHostDeathList();
        processPathogenDeathList();

        // inject invading/challenging hosts if hostTime>challengeStartTime
        if(executeHost() && challengerStartNum > 0 && hostTime>=challengeStartTime)
        {
            if(hostTime==challengeStartTime || (challengeFreq>0 && ((hostTime-challengeStartTime) %
                challengeFreq)==0))
            {
                scatterChallengers();
            }
        }

        // call method to update graphs
        updateGraphs();

        if (rDebug > 0)
        {
            System.out.printf("<== main step done.\n");
        }

    }

    // ///////////////////////////////////////////////////////////////////////
    // stepReport
    // each step write out:
    // Note: update the writeHeaderCommentsToReportFile() to print
    // lines of text describing the data written to the report file.
    public void stepReport()
    {

        if(enableStepReport == 1)
        {
            //if(executeHost())
            //{
                // set up a string with the values to write
                String s = String.format( "%5.0f", getTickCount() );
                s += String.format("  %d", numHosts);
                s += String.format("  %d", numPathogens);
                s += String.format("  %6.3f", minTransmissibility);
                s += String.format("  %6.3f", avgTransmissibility);
                s += String.format("  %6.3f", maxTransmissibility);
                s += String.format("  %6.3f", minVirulence);
                s += String.format("  %6.3f", avgVirulence);
                s += String.format("  %6.3f", maxVirulence);
                s += String.format("  %6.3f", minGrowthProb);
                s += String.format("  %6.3f", avgGrowthProb);
                s += String.format("  %6.3f", maxGrowthProb);
                s += String.format(" %d", numNaturalDeaths);
                s += String.format(" %d", numDiseaseDeaths);
                s += String.format("  %d", hostTime);
                s += String.format("  %6.3f", P_pp);
                s += String.format("  %6.3f", P_mm);
                s += String.format("  %6.3f", P_pm);
                s += String.format("  %6.3f", P_mp);
                s += String.format("  %6.3f", P_p0);
                s += String.format("  %6.3f", P_0p);
                s += String.format("  %6.3f", P_m0);
                s += String.format("  %6.3f", P_0m);
                s += String.format("  %6.3f", P_00);
                s += String.format("  %d", sum_P);

                for(int i=0; i<101; i++)
                {
                    s += String.format("  %d", numHGP[i]);
                }

                // write it to the xml and plain text report files
                //writeLineToReportFile("<stepreport>" + s + "</stepreport>");
                writeLineToPlaintextReportFile(s);

                // flush the buffers so the data is not lost in a "crash"
                getReportFile().flush();
                getPlaintextReportFile().flush();
            //}
        }

    }

    public void writeState()
    {
        try
        {
            BufferedWriter out = new BufferedWriter(new FileWriter("state_" + Double.toString(hostTime) +
                ".csv"));

            out.write("x, y, infected");
            out.newLine();
            // loop through all of the hosts and write their info
            for (Host aHost : hostList)
            {
                out.write(Integer.toString(aHost.getX()));
                out.write(",");
                out.write(Integer.toString(aHost.getY()));
                if(aHost.getInfected())
                {
                    out.write(", 1");
                }
                else
                {
                    out.write(", 0");
                }
                out.newLine();
            }
            out.close();
        } catch (IOException e)
        {

        }

        // write the frequency of descendents to a file
        try
        {
            BufferedWriter out = new BufferedWriter(new FileWriter("descendents_" + Double.toString
                hostTime) + ".csv"));

            out.write("descendents, frequency, frequency2");
            out.newLine();
            // loop through all of the hosts and write their info
            for (int i = 0; i<numDescendents.length; i++)
            {
                out.write(Integer.toString(i));
                out.write(",");
                out.write(Integer.toString(numDescendents[i]));
                out.write(",");
                out.write(Integer.toString(numDescendents2[i]));
                out.newLine();
            }
            out.close();
        } catch (IOException e)
        {

        }

        // write the frequency of ages to a file
        try
        {
            BufferedWriter out = new BufferedWriter(new FileWriter("ages_" + Double.toString(hostTime) +
                ".csv"));

            out.write("ages, frequency, frequency2");
            out.newLine();
            // loop through all of the hosts and write their info
            for (int i = 0; i<ageDist.length; i++)
            {
                out.write(Integer.toString(i));
                out.write(",");
                out.write(Integer.toString(ageDist[i]));
                out.write(",");
                out.write(Integer.toString(ageDist2[i]));
                out.newLine();
            }
            out.close();
        } catch (IOException e)
        {

        }
```

```
}
// ////////////////////////////////////////////////////////////////////////////

// writeHeaderCommentsToReportFile
// customize to match what you are writing to the report files in
// stepReport.

@Override
public void writeHeaderCommentsToReportFile()
{
    writeLineToReportFile("<comment>");
    writeLineToReportFile("      ");
    writeLineToReportFile("  time numHosts  numPathogens  minTrans avgTrans maxTrans minVirul
        avgVirul maxVirul minGrwth avgGrwth maxGrwth natDth disDth hostTime P_pp P_mm P_pm
        P_mp P_p0 P_0p P_m0 P_0m P_00 sum_P hGP=0.0 0.01    0.02 0.03 0.04 0.05 0.06 0.07 0.08
        0.09 0.1  0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28
        0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41 0.42 0.43 0.44 0.45 0.46 0.47 0.48
        0.49 0.5  0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68
        0.69 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88
        0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1");
    writeLineToReportFile("</comment>");

    writeLineToPlaintextReportFile("#      ");
    writeLineToPlaintextReportFile("# time  numHosts  numPathogens  minTrans avgTrans maxTrans
        minVirul avgVirul maxVirul minGrwth avgGrwth maxGrwth natDth disDth hostTime P_pp P_mm
        P_pm P_mp P_p0 P_0p P_m0 P_0m P_00 sum_P hGP=0.0 0.01    0.02 0.03 0.04 0.05 0.06 0.07
        0.08 0.09 0.1  0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
        0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41 0.42 0.43 0.44 0.45 0.46 0.47
        0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67
        0.68 0.69 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83 0.84 0.85 0.86 0.87
        0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1");
}

// ////////////////////////////////////////////////////////////////////////////
// printProjectHelp
// this could be filled in with some help to get from running with -help
// parameter
@Override
public void printProjectHelp()
{
    // print project help

    System.out.printf("\n%s -- \n", getName());

    System.out.printf("\n **** Add more info here!! **** \n");

    System.out.printf("\nactivationOrder           value\n");
    System.out.printf("\nfixed                     0\n");
    System.out.printf("\nrandomWithReplacement       1\n");
    System.out.printf("\nrandomWithoutReplacement     2\n");

    System.out.printf("\n");

    printParametersMap();

    System.exit(0);

}
```

```
void processHostBirthList()
{
    // shuffle the list so a host is chosen at random when more than one tries to
    // reproduce into the same cell
    Collections.shuffle(hostBirthList);

    // loop through the hostBirthList, adding hosts to empty spots
    for (Host aHost : hostBirthList)
    {
        // check to see if the spot is empty
        if (getOccupancyMatrix(aHost.getX(), aHost.getY()) == 0)
        {
            aHost.getParent().incrementOffspring();
            world.putObjectAt(aHost.getX(), aHost.getY(), aHost);
            setOccupancyMatrix(aHost.getX(), aHost.getY(), 1);
            hostList.add(aHost);
        }

    }

    // note that the next line will abort hosts if the spot they were allocated
    // to wasn't empty
    hostBirthList.clear();

}

void processPathogenBirthList()
{
    // shuffle the list so one of the pathogens at a given location is chosen at random
    Collections.shuffle(pathogenBirthList);

    // loop through the pathogenBirthList, infecting susceptible hosts
    for (Pathogen aPathogen : pathogenBirthList)
    {
        // check to see if the host is already infected or marked for death
        // If not, infect the host with the pathogen and add the pathogen to pathogenList
        if (!((Host)world.getObjectAt(aPathogen.getX(), aPathogen.getY())).getInfected() &&
                !((Host)world.getObjectAt(aPathogen.getX(), aPathogen.getY())).getDoomed())
        {
            ((Host)world.getObjectAt(aPathogen.getX(), aPathogen.getY())).setInfected(true);
            ((Host)world.getObjectAt(aPathogen.getX(), aPathogen.getY())).setPathogen(aPathogen);
            pathogenList.add(aPathogen);
        }

    }

    // note that the next line will abort pathogens whose host was already infected
    pathogenBirthList.clear();
}

public void processHostDeathList()
{
    numNaturalDeaths = 0;
    numDiseaseDeaths = 0;

    for (Host aHost : hostDeathList)
    {
        if(aHost.getNaturalDeath() == true)
        {
            numNaturalDeaths++;
        }
        else
        {
            numDiseaseDeaths++;
        }
        world.putObjectAt(aHost.getX(), aHost.getY(), null);
        offspringWorld.putObjectAt(aHost.getX(), aHost.getY(), null);
        descendentsWorld.putObjectAt(aHost.getX(), aHost.getY(), null);
        setOccupancyMatrix(aHost.getX(), aHost.getY(), 0);

        // record the number of descendents the host had when it died
        if(challengerStartNum>0 & aHost.getGrowthProb()==challengerGrowthProb)
        {
            if(aHost.getDescendents()<numDescendents2.length)
            {
                numDescendents2[aHost.getDescendents()]++;
            }
            if(aHost.getAge()<ageDist2.length)
            {
                ageDist2[aHost.getAge()]++;
            }
        }
        else
        {
            if(aHost.getDescendents()<numDescendents.length)
            {
                numDescendents[aHost.getDescendents()]++;
            }
            if(aHost.getAge()<ageDist.length)
            {
                ageDist[aHost.getAge()]++;
            }
        }
        hostList.remove(aHost);
    }
    hostDeathList.clear();

}

public void processPathogenDeathList()
{
    // All it should take to kill the pathogens is to remove them
    // from pathogenList -- simply destroying their host
    // (which will remove all links to the pathogens and let Java's garbage collection
    // take care of them) should be enough
    for (Pathogen aPathogen : pathogenDeathList)
    {
        pathogenList.remove(aPathogen);
    }
    pathogenDeathList.clear();
}

public void updateGraphs()
{
    //check one of the graphs to see if we are in GUI mode
    if ( transmissibilityGraph != null )
    {
        transmissibilityGraph.plotPoint(this.getTickCount(), minTransmissibility, 1);
        transmissibilityGraph.plotPoint(this.getTickCount(), avgTransmissibility, 2);
        transmissibilityGraph.plotPoint(this.getTickCount(), maxTransmissibility, 3);

        virulenceGraph.plotPoint(this.getTickCount(), minVirulence, 1);
        virulenceGraph.plotPoint(this.getTickCount(), avgVirulence, 2);
        virulenceGraph.plotPoint(this.getTickCount(), maxVirulence, 3);

        growthGraph.plotPoint(this.getTickCount(), minGrowthProb, 1);
        growthGraph.plotPoint(this.getTickCount(), avgGrowthProb, 2);
        growthGraph.plotPoint(this.getTickCount(), maxGrowthProb, 3);

        numHostsGraph.plotPoint(this.getTickCount(), numHosts, 1);

        numPathogensGraph.plotPoint(this.getTickCount(), numPathogens, 1);

    }
}

// calculate the average transmissibility of all pathogens
public void calcStatistics()
{
    double cumulTransmissibility = 0;
    double cumulVirulence = 0;
    double cumulGrowthProb = 0;
    int oldNumPathogens = numPathogens;

    // environs density variables
    boolean occupied=false;
    boolean infected=false;
    Vector<Host> neighbors;
    Host thisHost;
    int sum_P_pp = 0;
    int sum_P_mm = 0;
    int sum_P_pm = 0;
    int sum_P_mp = 0;
    int sum_P_p0 = 0;
    int sum_P_0p = 0;
    int sum_P_m0 = 0;
    int sum_P_0m = 0;
    int sum_P_00 = 0;

    numHosts = hostList.size();
    numPathogens = pathogenList.size();
    changeNumPathogens = numPathogens - oldNumPathogens;

    if(enableStepReport==1)
    {
        // reset max and min transmissibilities
        minTransmissibility = 1;
        maxTransmissibility = 0;

        // reset max and min virulences
        minVirulence = 1;
        maxVirulence = 0;
```

```
// Calculate average values for pathogens
if(pathogenList.size()==0)
{
    minTransmissibility = defaultTransmissibility;
    avgTransmissibility = defaultTransmissibility;
    maxTransmissibility = defaultTransmissibility;

    minVirulence = defaultVirulence;
    avgVirulence = defaultVirulence;
    maxVirulence = defaultVirulence;
}
else
{
    for (Pathogen aPathogen : pathogenList)
    {
        cumulTransmissibility += aPathogen.getTransmissibility();

        if(aPathogen.getTransmissibility() > maxTransmissibility)
        {
            maxTransmissibility = aPathogen.getTransmissibility();
        }

        if(aPathogen.getTransmissibility() < minTransmissibility)
        {
            minTransmissibility = aPathogen.getTransmissibility();
        }

        cumulVirulence += aPathogen.getVirulence();

        if(aPathogen.getVirulence() > maxVirulence)
        {
            maxVirulence = aPathogen.getVirulence();
        }

        if(aPathogen.getVirulence() < minVirulence)
        {
            minVirulence = aPathogen.getVirulence();
        }

    }

    avgTransmissibility = cumulTransmissibility/pathogenList.size();
    avgVirulence = cumulVirulence/pathogenList.size();
}

// reset max and min growth probabilities
maxGrowthProb = 0;
minGrowthProb = 1;

if(hostList.size()==0)
{
    minGrowthProb = defaultGrowthProb;
    avgGrowthProb = defaultGrowthProb;
    maxGrowthProb = defaultGrowthProb;
}
else
{
    for(int i=0; i<101; i++)
    {
        numHGP[i]=0;
    }
    for (Host aHost : hostList)
    {
        cumulGrowthProb += aHost.getGrowthProb();

        if(aHost.getGrowthProb() > maxGrowthProb)
        {
            maxGrowthProb = aHost.getGrowthProb();
        }

        if(aHost.getGrowthProb() < minGrowthProb)
        {
            minGrowthProb = aHost.getGrowthProb();
        }

        numHGP[(int) Math.floor((float)(aHost.getGrowthProb()*100))]++;

    }
    avgGrowthProb = cumulGrowthProb/hostList.size();
}

// Calculate the environs densities
sum_P = 0;
for (int i=0; i<sizeX; i++)
{
    for (int j=0; j<sizeY; j++)
    {

        // see if there's a host at this location, then see if it's infected
        occupied = false;
        infected = false;
        if(world.getObjectAt(i, j) != null)
        {
            occupied = true;
            if(((Host) world.getObjectAt(i, j)).getInfected())
            {
                infected = true;
            }
            else
            {
                infected = false;
            }
        }

        neighbors = world.getVonNeumannNeighbors(i, j, true);
        for (Host anotherHost : neighbors)
        {
            sum_P++;

            if(anotherHost != null)
            {
                boolean test = anotherHost.getInfected();
                if(occupied & !infected & !anotherHost.getInfected())
                {
                    sum_P_pp++;
                    }
                    else if(occupied & infected & anotherHost.getInfected())
                    {
                        sum_P_mm++;
                    }
                    else if(occupied & !infected & anotherHost.getInfected())
                    {
                        sum_P_pm++;
                    }
                    else if(occupied & infected & !anotherHost.getInfected())
                    {
                        sum_P_mp++;
                    }
                    else if(!occupied & !anotherHost.getInfected())
                    {
                        sum_P_0p++;
                    }
                    else if(!occupied & anotherHost.getInfected())
                    {
                        sum_P_0m++;
                    }
                }
                else
                {
                    if(occupied & !infected)
                    {
                        sum_P_p0++;
                    }
                    if(occupied & infected)
                    {
                        sum_P_m0++;
                    }
                    else if(!occupied)
                    {
                        sum_P_00++;
                    }
                }

            }
        }
    }
    P_pp = (double)sum_P_pp/sum_P;
    P_mm = (double)sum_P_mm/sum_P;
    P_pm = (double)sum_P_pm/sum_P;
    P_mp = (double)sum_P_mp/sum_P;
    P_p0 = (double)sum_P_p0/sum_P;
    P_0p = (double)sum_P_0p/sum_P;
    P_m0 = (double)sum_P_m0/sum_P;
    P_0m = (double)sum_P_0m/sum_P;
    P_00 = (double)sum_P_00/sum_P;
  }

}

public Boolean executeHost()
{
  if(getHostExecutionPeriod()==-999)
  {
```

```
            if(newMethod)
            {
                if(numPathogens==0)
                {
                    return Boolean.TRUE;
                }
                else
                {
                    return Boolean.FALSE;
                }
            }
            else
            {
                if(changeNumPathogens==0)
                {
                    return Boolean.TRUE;
                }
                else
                {
                    return Boolean.FALSE;
                }
            }

        }
        else
        {
            if(getTickCount() % getHostExecutionPeriod() == 0)
            {
                return Boolean.TRUE;
            }
            else
            {
                return Boolean.FALSE;
            }
        }
    }

    public void incrementHostTime()
    {
        hostTime++;
    }
    @Override
    public Schedule getSchedule()
    {
        return schedule;
    }

    @Override
    public String getName()
    {
        return "HostPathogen";
    }

    // setters and getters
    // notes:
    // - we use the schedule != null to indicated model has been initialized
    // - some things can't be changed after model initialization
    // (which things just depends on how the model is implemented)
```

```
// - if we set something after model initialization,
// we need to write an change entry to the report file.
// - some things need to send messages to update class variables.
//
// NOTE: if you want changes a user makes to parameter like numBugs
// to be used after a restart (vs going back to defaults),
// you probably have to change setup() to not reinitialize IVs.

public static void setTransmissibilityGraph (Plot graph) { transmissibilityGraph = graph; };
public static void setVirulenceGraph (Plot graph) { virulenceGraph = graph; };
public static void setGrowthGraph (Plot graph) { growthGraph = graph; };
public static void setNumHostsGraph (Plot graph) { numHostsGraph = graph; };
public static void setNumPathogensGraph (Plot graph) { numPathogensGraph = graph; };

public boolean getNewMethod()
{
    return newMethod;
}

public int getSizeX()
{
    return sizeX;
}

public void setSizeX(int sizeX)
{
    this.sizeX = sizeX;
}

public int getSizeY()
{
    return sizeY;
}

public void setSizeY(int sizeY)
{
    this.sizeY = sizeY;
}

public void setOccupancyMatrix(int x, int y, int occupied)
{
    this.occupancyMatrix[x][y] = occupied;
}

public int getOccupancyMatrix(int x, int y)
{
    return occupancyMatrix[x][y];
}

public int getStartNumHosts()
{
    return startNumHosts;
}

public int getNumHosts()
{
    return numHosts;
}
```

```
public void setStartNumHosts(int startNumHosts)
{
    this.startNumHosts = startNumHosts;
}

public int getStartNumPathogens()
{
    return startNumPathogens;
}

public int getNumPathogens()
{
    return numPathogens;
}

public void setStartNumPathogens(int startNumPathogens)
{
    this.startNumPathogens = startNumPathogens;
}

public int getGraphUpdatePeriod()
{
    return graphUpdatePeriod;
}
public void setGraphUpdatePeriod(int gUP)
{
    graphUpdatePeriod = gUP;
}

public int getHostExecutionPeriod()
{
    return hostExecutionPeriod;
}
public void setHostExecutionPeriod(int hEP)
{
    hostExecutionPeriod = hEP;
}

public double getProbLongDistance()
{
    return probLongDistance;
}
public void setProbLongDistance(double pLD)
{
    probLongDistance = pLD;
}

public double getDefaultGrowthProb()
{
    return defaultGrowthProb;
}
public void setDefaultGrowthProb(double dGP)
{
    defaultGrowthProb = dGP;
}

public double getDefaultNaturalDeathZero()
```

```
{
    return defaultNaturalDeathZero;
}
public void setDefaultNaturalDeathZero(double dNDZ)
{
    defaultNaturalDeathZero = dNDZ;
}

public double getDefaultNaturalDeathSlope()
{
    return defaultNaturalDeathSlope;
}
public void setDefaultNaturalDeathSlope(double dNDS)
{
    defaultNaturalDeathSlope = dNDS;
}

public double getProbGrowthMutate()
{
    return probGrowthMutate;
}
public void setProbGrowthMutate(double pGM)
{
    probGrowthMutate = pGM;
}

public double getGrowthMutation()
{
    return growthMutation;
}
public void setGrowthMutation(double gM)
{
    growthMutation = gM;
}

public double getDefaultTransmissibility()
{
    return defaultTransmissibility;
}
public void setDefaultTransmissibility(double tau)
{
    defaultTransmissibility = tau;
}

public double getDefaultVirulence()
{
    return defaultVirulence;
}
public void setDefaultVirulence(double v)
{
    defaultVirulence = v;
}

public double getProbTransMutate()
{
    return probTransMutate;
}
public void setProbTransMutate(double pTM)
```

```
{
    probTransMutate = pTM;
}

public double getTransMutation()
{
    return transMutation;
}
public void setTransMutation(double tM)
{
    transMutation = tM;
}

public double getProbVirulenceMutate()
{
    return probVirulenceMutate;
}
public void setProbVirulenceMutate(double pVM)
{
    probVirulenceMutate = pVM;
}

public double getVirulenceMutation()
{
    return virulenceMutation;
}
public void setVirulenceMutation(double vM)
{
    virulenceMutation = vM;
}

public double getChallengerGrowthProb()
{
    return challengerGrowthProb;
}
public void setChallengerGrowthProb(double cGP)
{
    challengerGrowthProb = cGP;
}

public int getChallengerStartNum()
{
    return challengerStartNum;
}
public void setChallengerStartNum(int cSN)
{
    challengerStartNum = cSN;
}

public int getChallengeStartTime()
{
    return challengeStartTime;
}
public void setChallengeStartTime(int cST)
{
    challengeStartTime = cST;
}
```

```
public int getChallengeFreq()
{
    return challengeFreq;
}
public void setChallengeFreq(int cFR)
{
    challengeFreq = cFR;
}

public int getWriteTime()
{
    return writeTime;
}
public void setWriteTime(int wT)
{
    writeTime = wT;
}

public int getEnableStepReport()
{
    return enableStepReport;
}
public void setEnableStepReport(int eSR)
{
    enableStepReport = eSR;
}

public void addHostBirth (Host h)
{
    hostBirthList.add(h);
}

public void addPathogenBirth (Pathogen p)
{
    pathogenBirthList.add(p);
}

public void addHostDeath (Host h)
{
    hostDeathList.add(h);
}

public void addPathogenDeath (Pathogen p)
{
    pathogenDeathList.add(p);
}

// ////////////////////////////////////////////////////////////////////////
// processEndOfRun
// called once, at end of run.
// writes some final info, closes report files, etc.
public void processEndOfRun()
{
    if (rDebug > 0)
        System.out.printf("\n\n===== processEndOfRun =====\n\n");
    applyAnyStoredChanges();
    endReportFile();
    this.fireStopSim();
    }

}
```

Host.java

```
/**
 * Host.java
 */

package hostPathogen_v7;

import hostPathogen_v7.GUIModel;

import java.awt.BasicStroke;
import java.lang.Boolean;
import java.util.Vector;

import uchicago.src.sim.gui.*;
import uchicago.src.sim.space.Object2DTorus;
import uchicago.src.sim.gui.ColorMap;
import java.awt.Color;

public class Host implements Drawable
{
    // class variables
    public static int nextID = 0; // to give each an ID
    public static Object2DTorus hostsWorld; // where the hosts live
    public static Object2DTorus offspringWorld;
    public static Object2DTorus descendentsWorld;
    public static HostPathogen model; // the model "in charge"
    public static GUIModel guiModel = null; // the gui model "in charge"
    public static ColorMap colorMap;
    public static double maxOffspring = 10;
    public static double maxDescendents = 20;

    public static double defaultGrowthProb;
    public static double defaultNaturalDeathZero;
    public static double defaultNaturalDeathSlope;
    public static double probGrowthMutate;
    public static double growthMutation;

    // instance variables
    public  int         ID;
    public boolean infected;
    public int x, y;
    public Color myColor;
    public Pathogen myPathogen;
    public double growthProb;
    public double naturalDeathZero;
    public double naturalDeathSlope;
    public boolean naturalDeath;
    public boolean doomed;
    public int age;
    public Host myParent;
    public int offspring;
    public int descendents;
    public StatsDisplayObject offspringDisplayObject;
    public StatsDisplayObject descendentsDisplayObject;

    // the Host constructors
    public Host(int X, int Y)
```

```
{
    ID = nextID++;
    x = X;
    y = Y;
    infected = false;
    doomed = false;
    growthProb = defaultGrowthProb;
    setColorMap();
    myColor = colorMap.getColor(Math.round((float)(63*growthProb)));
    naturalDeathZero= defaultNaturalDeathZero;
    naturalDeathSlope = defaultNaturalDeathSlope;
    age = 0;
    myParent = null;
    offspring = 0;
    descendents = 0;
    offspringDisplayObject = new StatsDisplayObject(Color.WHITE);
    offspringWorld.putObjectAt(getX(), getY(), offspringDisplayObject);
    descendentsDisplayObject = new StatsDisplayObject(Color.WHITE);
    descendentsWorld.putObjectAt(getX(), getY(), descendentsDisplayObject);
}

public Host(int X, int Y, double g)
{
    ID = nextID++;
    x = X;
    y = Y;
    infected = false;
    doomed = false;
    growthProb = g;
    setColorMap();
    myColor = colorMap.getColor(Math.round((float)(63*growthProb)));
    naturalDeathZero= defaultNaturalDeathZero;
    naturalDeathSlope = defaultNaturalDeathSlope;
    age = 0;
    myParent = null;
    offspring = 0;
    descendents = 0;
    offspringDisplayObject = new StatsDisplayObject(Color.WHITE);
    offspringWorld.putObjectAt(getX(), getY(), offspringDisplayObject);
    descendentsDisplayObject = new StatsDisplayObject(Color.WHITE);
    descendentsWorld.putObjectAt(getX(), getY(), descendentsDisplayObject);
}

public Host(int X, int Y, double g, Host mP)
{
    ID = nextID++;
    x = X;
    y = Y;
    infected = false;
    doomed = false;
    growthProb = g;
    setColorMap();
    myColor = colorMap.getColor(Math.round((float)(63*growthProb)));
    naturalDeathZero= defaultNaturalDeathZero;
    naturalDeathSlope = defaultNaturalDeathSlope;
    age = 0;
    myParent = mP;
    offspring = 0;
    descendents = 0;
    offspringDisplayObject = new StatsDisplayObject(Color.WHITE);
    offspringWorld.putObjectAt(getX(), getY(), offspringDisplayObject);
    descendentsDisplayObject = new StatsDisplayObject(Color.WHITE);
    descendentsWorld.putObjectAt(getX(), getY(), descendentsDisplayObject);
}

// setupHostDrawing
// set the guiModel address, which we can test to see if in GUI mode
public static void setupHostDrawing ( GUIModel m )
{
    guiModel = m;
}
// ///////////////////////////////////////////////////////////////////////
// step
// apply the CA rules

// The static block is essentially a constructor for the class
static
{
    colorMap = new ColorMap();
    setColorMap();
}
public void step()
{
    double randNum;
    int numNeighbors;
    Vector<Host> neighbors;

    // check to see if the hosts should execute this time step
    if(model.executeHost())
    {
        age ++;

        // See if myParent has died. If so, forget about them.
        if(myParent != null)
        {
            if(myParent.getDoomed()==true)
            {
                myParent = null;
            }
        }

        // calculate the probability of death based on the number of hosts in the von Neumann
           neighborhood
        neighbors = hostsWorld.getVonNeumannNeighbors(x, y, false);
        numNeighbors = neighbors.size();

        // see if the host will die a natural death
        randNum = ModelParameters.getUniformDoubleFromTo(0, 1);
        if(randNum < (naturalDeathZero+naturalDeathSlope*numNeighbors))
        {
            setNaturalDeath(true);
            model.addHostDeath(this);
            doomed = true;
            if(infected)
            {
                model.addPathogenDeath(this.myPathogen);
            }

        }
    }

    if((model.executeHost() && !model.getNewMethod()) | model.getNewMethod())
    {
        // see if the Host is infected, and, if so, if it will die
        if (infected)
        {
            // die with probability = virulence
            randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

            if (randNum < myPathogen.getVirulence())
            {
                setNaturalDeath(false);
                model.addHostDeath(this);
                doomed = true;
                model.addPathogenDeath(this.myPathogen);
            }

        }
    }
    if(offspring != 0)
    {
        offspringDisplayObject.setMyColor(colorMap.getColor(Math.round((float)(63*(offspring/
            maxOffspring)))));
    }
    if(descendents != 0)
    {
        descendentsDisplayObject.setMyColor(colorMap.getColor(Math.round((float)(63*(descendents/
            maxDescendents)))));
    }
}

public void reproduce(int tempX, int tempY)
{
    double randNum;
    double newGrowthProb; // the growthProb that the offspring will have

    // check to see if the hosts should execute this time step
    if(model.executeHost())
    {
        // reset the infectionTries count of the pathogen
        if(infected)
        {
            myPathogen.resetInfectionTries();
        }

        // see if the cell is unoccupied
        // IMPORTANT: INFECTED HOSTS CANNOT REPRODUCE

        // getObject is expensive, so check infected==false first
        // I've replaced getObject with getOccupancyMatrix, but I think the previous
        // structure is still OK
        if(infected==false)
        {
            if (model.getOccupancyMatrix(tempX, tempY) == 0)
```

```
                {
                    // reproduce with probability = growthProb
                    randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

                    if (randNum < growthProb)
                    {
                        // mutate with probability probGrowthMutate
                        randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

                        // subtract growthMutation w/ a probability of probGrowthMutate/2;
                        // add growthMutation w/ a probability of probGrowthMutate/2
                        if (randNum < probGrowthMutate/2)
                        {
                            newGrowthProb = Math.max(0, growthProb - growthMutation);
                        }
                        else if (randNum < probGrowthMutate)
                        {
                            newGrowthProb = Math.min(1, growthProb + growthMutation);
                        }
                        else
                        {
                            newGrowthProb = growthProb;
                        }

                        Host aHost = new Host(tempX, tempY, newGrowthProb, this);
                        model.addHostBirth(aHost);
                    }
                }
            }

        }
        // the host's pathogen also gets a chance to reproduce
        if(infected)
        {
            myPathogen.reproduce(tempX, tempY);
        }

    }

    // ///////////////////////////////////////////////////////////////////////
    // setters and getters
    public void setID(int i)
    {
        ID = i;
    }

    public int getID()
    {
        return ID;
    }

    // note these are class methods, to set class variables
    public static void setHostsWorld(Object2DTorus world)
    {
        hostsWorld = world;
    }

    public static void setOffspringWorld(Object2DTorus world)
```

```
{
    offspringWorld = world;
}

public static void setDescendentsWorld(Object2DTorus world)
{
    descendentsWorld = world;
}

public static void setModel(HostPathogen m)
{
    model = m;
}

public static void setGUIModel(GUIModel m)
{
    guiModel = m;
}

public static void setDefaultGrowthProb(double dGP)
{
    defaultGrowthProb = dGP;
}

public static void setDefaultNaturalDeathZero(double dNDZ)
{
    defaultNaturalDeathZero = dNDZ;
}

public static void setDefaultNaturalDeathSlope(double dNDS)
{
    defaultNaturalDeathSlope = dNDS;
}

public static double getProbGrowthMutate()
{
    return probGrowthMutate;
}
public static void setProbGrowthMutate(double pGM)
{
    probGrowthMutate = pGM;
}

public static double getGrowthMutation()
{
    return growthMutation;
}
public static void setGrowthMutation(double gM)
{
    growthMutation = gM;
}

public static void setColorMap()
{
    colorMap.mapColor(0, 0, 0, 0.5625);
    colorMap.mapColor(1, 0, 0, 0.625);
    colorMap.mapColor(2, 0, 0, 0.6875);
    colorMap.mapColor(3, 0, 0, 0.75);
    colorMap.mapColor(4, 0, 0, 0.8125);
    colorMap.mapColor(5, 0, 0, 0.875);
    colorMap.mapColor(6, 0, 0, 0.9375);
    colorMap.mapColor(7, 0, 0, 1);
    colorMap.mapColor(8, 0, 0.0625, 1);
    colorMap.mapColor(9, 0, 0.125, 1);
    colorMap.mapColor(10, 0, 0.1875, 1);
    colorMap.mapColor(11, 0, 0.25, 1);
    colorMap.mapColor(12, 0, 0.3125, 1);
    colorMap.mapColor(13, 0, 0.375, 1);
    colorMap.mapColor(14, 0, 0.4375, 1);
    colorMap.mapColor(15, 0, 0.5, 1);
    colorMap.mapColor(16, 0, 0.5625, 1);
    colorMap.mapColor(17, 0, 0.625, 1);
    colorMap.mapColor(18, 0, 0.6875, 1);
    colorMap.mapColor(19, 0, 0.75, 1);
    colorMap.mapColor(20, 0, 0.8125, 1);
    colorMap.mapColor(21, 0, 0.875, 1);
    colorMap.mapColor(22, 0, 0.9375, 1);
    colorMap.mapColor(23, 0, 1, 1);
    colorMap.mapColor(24, 0.0625, 1, 0.9375);
    colorMap.mapColor(25, 0.125, 1, 0.875);
    colorMap.mapColor(26, 0.1875, 1, 0.8125);
    colorMap.mapColor(27, 0.25, 1, 0.75);
    colorMap.mapColor(28, 0.3125, 1, 0.6875);
    colorMap.mapColor(29, 0.375, 1, 0.625);
    colorMap.mapColor(30, 0.4375, 1, 0.5625);
    colorMap.mapColor(31, 0.5, 1, 0.5);
    colorMap.mapColor(32, 0.5625, 1, 0.4375);
    colorMap.mapColor(33, 0.625, 1, 0.375);
    colorMap.mapColor(34, 0.6875, 1, 0.3125);
    colorMap.mapColor(35, 0.75, 1, 0.25);
    colorMap.mapColor(36, 0.8125, 1, 0.1875);
    colorMap.mapColor(37, 0.875, 1, 0.125);
    colorMap.mapColor(38, 0.9375, 1, 0.0625);
    colorMap.mapColor(39, 1, 1, 0);
    colorMap.mapColor(40, 1, 0.9375, 0);
    colorMap.mapColor(41, 1, 0.875, 0);
    colorMap.mapColor(42, 1, 0.8125, 0);
    colorMap.mapColor(43, 1, 0.75, 0);
    colorMap.mapColor(44, 1, 0.6875, 0);
    colorMap.mapColor(45, 1, 0.625, 0);
    colorMap.mapColor(46, 1, 0.5625, 0);
    colorMap.mapColor(47, 1, 0.5, 0);
    colorMap.mapColor(48, 1, 0.4375, 0);
    colorMap.mapColor(49, 1, 0.375, 0);
    colorMap.mapColor(50, 1, 0.3125, 0);
    colorMap.mapColor(51, 1, 0.25, 0);
    colorMap.mapColor(52, 1, 0.1875, 0);
    colorMap.mapColor(53, 1, 0.125, 0);
    colorMap.mapColor(54, 1, 0.0625, 0);
    colorMap.mapColor(55, 1, 0, 0);
    colorMap.mapColor(56, 0.9375, 0, 0);
    colorMap.mapColor(57, 0.875, 0, 0);
    colorMap.mapColor(58, 0.8125, 0, 0);
    colorMap.mapColor(59, 0.75, 0, 0);
    colorMap.mapColor(60, 0.6875, 0, 0);
    colorMap.mapColor(61, 0.625, 0, 0);
```

```
    colorMap.mapColor(62, 0.5625, 0, 0);
    colorMap.mapColor(63, 0.5, 0, 0);
}

public int getX()
{
    return x;
}

public void setX(int i)
{
    x = i;
}

public int getY()
{
    return y;
}

public void setY(int i)
{
    y = i;
}

public boolean getInfected()
{
    return infected;
}

public void setInfected(boolean i)
{
    infected = i;

    if(i)
    {
        myColor = Color.white;
    }
    else
    {
        myColor = myColor = colorMap.getColor(Math.round((float)(63*growthProb)));
    }

}

public boolean getDoomed()
{
    return doomed;
}

public Pathogen getPathogen()
{
    return myPathogen;
}

public void setPathogen(Pathogen p)
{
    myPathogen = p;
}
```

```
public double getGrowthProb()
{
    return growthProb;
}

public void setGrowthProb(double gP)
{
    growthProb = gP;

}

public double getNaturalDeathZero()
{
    return naturalDeathZero;
}

public void setNaturalDeathZero(double nDZ)
{
    naturalDeathZero = nDZ;

}

public double getNaturalDeathSlope()
{
    return naturalDeathSlope;
}

public void setNaturalDeathSlope(double nDS)
{
    naturalDeathSlope = nDS;

}

public String getName()
{
    return "Host";
}

public void setNaturalDeath(boolean nD)
{
    naturalDeath = nD;
}

public boolean getNaturalDeath()
{
    return naturalDeath;
}

public Host getParent()
{
    return myParent;
}

public int getOffspring()
{
    return offspring;
}
```

```
    public int getDescendents()
    {
        return descendents;
    }

    public int getAge()
    {
        return age;
    }

    public void incrementOffspring()
    {
        offspring++;
        incrementDescendents();
    }

    public void incrementDescendents()
    {
        descendents++;
        if(myParent != null)
        {
            myParent.incrementDescendents();
        }
    }
    // we implement Drawable interface, so we need this method
    // so that the Host can draw itself when requested
    // (by the GUI display).

    public void draw(SimGraphics g)
    {
        g.drawFastRoundRect(myColor);
    }

}
```

Pathogen.java

```
/**
 * Pathogen.java
 */

package hostPathogen_v7;

import hostPathogen_v7.GUIModel;

import java.awt.BasicStroke;
import java.lang.Math;

import uchicago.src.sim.gui.*;
import uchicago.src.sim.space.Object2DTorus;
import java.awt.Color;

public class Pathogen implements Drawable
{
    // class variables
    public static int nextID = 0; // to give each an ID
    public static HostPathogen model; // the model "in charge"
    public static Object2DTorus pathogensWorld; // where the hosts live
    public static GUIModel guiModel = null; // the gui model "in charge"
    // we'll use this to draw a border around the pathogens' cells (the f means
    // float)
    public static BasicStroke pathogenEdgeStroke = new BasicStroke(1.0f);

    public static double defaultTransmissibility;
    public static double defaultVirulence;
    public static double probTransMutate;
    public static double transMutation;
    public static double probVirulenceMutate;
    public static double virulenceMutation;

    // instance variables
    public  int              ID;
    public int x, y;
    public Color myColor;
    public int infectionTries;

    public double transmissibility;
    public double virulence;

    // the Pathogen constructor
    public Pathogen(int X, int Y)
    {
        ID = nextID++;
        x = X;
        y = Y;
        myColor = Color.white;
        transmissibility = defaultTransmissibility;
        virulence = defaultVirulence;
        infectionTries = 0;
    }

    public Pathogen(int X, int Y, double tau, double v)
    {
        ID = nextID++;
        x = X;
        y = Y;
        myColor = Color.white;
        transmissibility = tau;
        virulence = v;
        infectionTries = 0;
    }

    // setupPathogenDrawing
    // set the guiModel address, which we can test to see if in GUI mode
    public static void setupPathogenDrawing ( GUIModel m )
    {
        guiModel = m;
    }
    // ///////////////////////////////////////////////////////////////////////
    // step
    // apply the CA rules
    public void step()
    {
        // do nothing for now
    }

    public void reproduce(int tempX, int tempY)
    {
        double randNum;
        double newTransmissibility;
        double newVirulence;

        // each pathogen only gets four tries to infect, i.e., one try
        // for each cell in the von Neumann neighborhood
        infectionTries++;

        // only try to infect if infectionTries<4 (von Neumann neighborhood)
        if(infectionTries<=4 | model.getNewMethod())
        {
            // see if the cell has a host in it
            if (pathogensWorld.getObjectAt(tempX, tempY) != null)
            {

                // see if the host is not already infected
                if (((Host)pathogensWorld.getObjectAt(tempX, tempY)).getInfected() == false)
                {
                    // reproduce with probability = transmissibility
                    randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

                    if (randNum < transmissibility)
                    {
                        // mutate with probabilities probTransMutate and probVirulenceMutate
                        randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

                        // subtract transMutation w/ a probability of probTransMutate/2;
                        // add transMutation w/ a probability of probTransMutate/2
                        if (randNum < probTransMutate/2)
                        {
                            newTransmissibility = Math.max(0, transmissibility - transMutation);
                        }
                        else if (randNum < probTransMutate)
                    {
                        newTransmissibility = Math.min(1, transmissibility + transMutation);
                    }
                    else
                    {
                        newTransmissibility = transmissibility;
                    }

                    randNum = ModelParameters.getUniformDoubleFromTo(0, 1);

                    // subtract virulenceMutation w/ a probability of probVirulenceMutate/2;
                    // add virulenceMutation w/ a probability of probVirulenceMutate/2
                    if (randNum < probVirulenceMutate/2)
                    {
                        newVirulence = Math.max(0, virulence - virulenceMutation);
                    }
                    else if (randNum < probVirulenceMutate)
                    {
                        newVirulence = Math.min(1, virulence + virulenceMutation);
                    }
                    else
                    {
                        newVirulence = virulence;
                    }

                    Pathogen aPathogen = new Pathogen(tempX, tempY, newTransmissibility,
                                newVirulence);
                    model.addPathogenBirth(aPathogen);
                }
            }
        }
    }
}

public void resetInfectionTries()
{
    infectionTries = 0;
}
// ////////////////////////////////////////////////////////////////////////
// setters and getters
public void setID(int i)
{
    ID = i;
}

public int getID()
{
    return ID;
}

// note these are class methods, to set class variables
public static void setPathogensWorld(Object2DTorus world)
{
    pathogensWorld = world;
}
public static void setModel(HostPathogen m)
{
    model = m;
```

```
}

public static void setGUIModel(GUIModel m)
{
    guiModel = m;
}

public static void setDefaultTransmissibility(double tau)
{
    defaultTransmissibility = tau;
}

public static void setDefaultVirulence(double v)
{
    defaultVirulence = v;
}

public static double getProbTransMutate()
{
    return probTransMutate;
}
public static void setProbTransMutate(double pTM)
{
    probTransMutate = pTM;
}

public static double getTransMutation()
{
    return transMutation;
}
public static void setTransMutation(double tM)
{
    transMutation = tM;
}

public static double getProbVirulenceMutate()
{
    return probVirulenceMutate;
}
public static void setProbVirulenceMutate(double pVM)
{
    probVirulenceMutate = pVM;
}

public static double getVirulenceMutation()
{
    return virulenceMutation;
}
public static void setVirulenceMutation(double vM)
{
    virulenceMutation = vM;
}

public int getX()
{
    return x;
}
```

```
    public void setX(int i)
    {
        x = i;
    }

    public int getY()
    {
        return y;
    }

    public void setY(int i)
    {
        y = i;
    }

    public double getVirulence()
    {
        return virulence;
    }

    public double getTransmissibility()
    {
        return transmissibility;
    }

    public String getName()
    {
        return "Pathogen";
    }

    // we implement Drawable interface, so we need this method
    // so that the pathogen can draw itself when requested
    // (by the GUI display).

    public void draw(SimGraphics g)
    {
        g.drawFastRoundRect(myColor);
        g.drawRectBorder(pathogenEdgeStroke, Color.cyan);
    }

}
```

StatsDisplayObject.java

```
package hostPathogen_v7;

import java.awt.Color;

import uchicago.src.sim.gui.Drawable;
import uchicago.src.sim.gui.SimGraphics;

public class StatsDisplayObject implements Drawable
{
    public Color myColor;
    public int x, y;

    public StatsDisplayObject(Color mC)
    {
        myColor = mC;
    }

    @Override
    public void draw(SimGraphics g)
    {
        g.drawFastRoundRect(myColor);
    }

    @Override
    public int getX() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public int getY() {
        // TODO Auto-generated method stub
        return 0;
    }

    public void setMyColor(Color mC)
    {
        myColor = mC;
    }

}
```

GUIModel.java

```
package hostPathogen_v7;

// A model of an evolutionary, spatially explicit host-pathogen system.
// Doug Jackson, Summer 2008

import uchicago.src.sim.gui.DisplaySurface;
import uchicago.src.sim.gui.Object2DDisplay;
import uchicago.src.sim.engine.AbstractGUIController;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.analysis.*;

public class GUIModel extends HostPathogen
{

    // (Repast)
    private Object2DDisplay worldDisplay; // 2D Object lattice -> display
    private DisplaySurface dsurf; // display surface

    // display and surface for # of offspring
    private Object2DDisplay offspringDisplay;
    private DisplaySurface offspringSurf;

    // display and surface for # of descendents
    private Object2DDisplay descendentsDisplay;
    private DisplaySurface descendentsSurf;

    private Plot transmissibilityGraph; // Graph 1
    private Plot virulenceGraph; // Graph 2
    private Plot growthGraph; // Graph 3
    private Plot numHostsGraph; // Graph 4
    private Plot numPathogensGraph; // Graph 5

    // ////////////////////////////////////////////////////////////////
    // setup
    //
    // this runs automatically when the model starts
    // and when you click the reload button, to "tear down" any
    // existing display objects, and get ready to initialize
    // them at the start of the next 'run'.
    //
    @Override
    public void setup()
    {
        super.setup(); // the super class does conceptual-model setup

        AbstractGUIController.CONSOLE_ERR = false;
        AbstractGUIController.CONSOLE_OUT = false;
        AbstractGUIController.UPDATE_PROBES = true;

        // dispose of any leftover display surfaces
        if (dsurf != null)
            dsurf.dispose();
        if (offspringSurf != null) offspringSurf.dispose();
        if (descendentsSurf != null) descendentsSurf.dispose();

        // create the new display surfaces
        dsurf = null;
        dsurf = new DisplaySurface(this, "Display");
        registerDisplaySurface("Main display", dsurf);
        offspringSurf = null;
        offspringSurf = new DisplaySurface(this, "Offspring display");
        registerDisplaySurface("Offspring display", offspringSurf);

        descendentsSurf = null;
        descendentsSurf = new DisplaySurface(this, "Descendents display");
        registerDisplaySurface("Descendents display", descendentsSurf);

        // clear any residual graphs
        if ( transmissibilityGraph != null ) transmissibilityGraph.dispose();
        transmissibilityGraph = null;
        if ( virulenceGraph != null ) virulenceGraph.dispose();
        virulenceGraph = null;
        if ( growthGraph != null ) growthGraph.dispose();
        growthGraph = null;
        if ( numHostsGraph != null ) numHostsGraph.dispose();
        numHostsGraph = null;
        if ( numPathogensGraph != null ) numPathogensGraph.dispose();
        numPathogensGraph = null;

        // tell the Host class we are in GUI mode.
        Host.setupHostDrawing(this);

        // init, setup and turn on the modelMinipulator stuff (in custom
        // actions)
        modelManipulator.init();

        if (rDebug > 0)
            System.out.printf("<== GUIModel setup() done.\n");
    }

    // ///////////////////////////////////////////////////////////////////////
    // begin
    //
    // this runs when you click the "initialize" button
    // (the button with the single arrow that goes around in a circle)
    //
    @Override
    public void begin()
    {
        DMSG(1, "==> enter GUIModel-begin()");
        buildModel(); // the base model does this
        buildDisplay();
        buildSchedule();
        dsurf.display();
        offspringSurf.display();
        descendentsSurf.display();
        DMSG(1, "<== leave GUIModel-begin() done.");
    }

    // ///////////////////////////////////////////////////////////////////////
    // buildDisplay
    //
    // builds the display and display related things
    //
```

```
public void buildDisplay()
{
    if (rDebug > 0)
        System.out.printf("==> GUIModel buildDisplay...\n");

    // create the link between the display surfaces and the Object2DTorus worlds,
    // and tell parts about each other as needed.

    worldDisplay = new Object2DDisplay(world);
    worldDisplay.setObjectList(hostList);

    offspringDisplay = new Object2DDisplay(offspringWorld);
    descendentsDisplay = new Object2DDisplay(descendentsWorld);

    // note we will be able to probe the objects with MB3 (right)
    dsurf.addDisplayableProbeable(worldDisplay, "Shade trees");

    offspringSurf.addDisplayable(offspringDisplay, "Number of offspring");
    descendentsSurf.addDisplayable(descendentsDisplay, "Number of descendents");

    addSimEventListener(dsurf); // link to the other parts of the repast gui
    addSimEventListener(offspringSurf);
    addSimEventListener(descendentsSurf);

    // enable the custom action(s)
    modelManipulator.setEnabled(true);

    if (rDebug > 0)
        System.out.printf("<== GUIModel buildDisplay done.\n");

    // Graphs

    //Graph 1 - a graph showing the average transmissibility vs. time
    transmissibilityGraph = new Plot("Transmissibility", this);
    transmissibilityGraph.setXRange( 0, 200 );
    transmissibilityGraph.setYRange( 0, 1 );
    transmissibilityGraph.setAxisTitles("Time", "Transmissibility");

    transmissibilityGraph.addLegend(1, "min. transmissibility");
    transmissibilityGraph.addLegend(2, "avg. transmissibility");
    transmissibilityGraph.addLegend(3, "max. transmissibility");

    //tell the model about it.
    HostPathogen.setTransmissibilityGraph(transmissibilityGraph);

    // now actually display the graph on the screen.
    transmissibilityGraph.display();

    // end Graph 1

    //Graph 2 - a graph showing the average transmissibility vs. time
    virulenceGraph = new Plot("Virulence", this);
    virulenceGraph.setXRange( 0, 200 );
    virulenceGraph.setYRange( 0, 1 );
    virulenceGraph.setAxisTitles("Time", "Virulence");

    virulenceGraph.addLegend(1, "min. virulence");
    virulenceGraph.addLegend(2, "avg. virulence");
    virulenceGraph.addLegend(3, "max. virulence");

    //tell the model about it.
    HostPathogen.setVirulenceGraph(virulenceGraph);

    // now actually display the graph on the screen.
    virulenceGraph.display();

    // end Graph 2

    //Graph 3 - a graph showing the average transmissibility vs. time
    growthGraph = new Plot("Host growth probability", this);
    growthGraph.setXRange( 0, 200 );
    growthGraph.setYRange( 0, 1 );
    growthGraph.setAxisTitles("Time", "Growth Probability");

    growthGraph.addLegend(1, "min. growth probability");
    growthGraph.addLegend(2, "avg. growth probability");
    growthGraph.addLegend(3, "max. growth probability");

    //tell the model about it.
    HostPathogen.setGrowthGraph(growthGraph);

    // now actually display the graph on the screen.
    growthGraph.display();

    // end Graph 3

    //Graph 4 - a graph showing the number of hosts vs. time
    numHostsGraph = new Plot("Number of hosts", this);
    numHostsGraph.setXRange( 0, 200 );
    numHostsGraph.setYRange( 0, 1 );
    numHostsGraph.setAxisTitles("Time", "Number of hosts");

    numHostsGraph.addLegend(1, "Number of hosts");

    //tell the model about it.
    HostPathogen.setNumHostsGraph(numHostsGraph);

    // now actually display the graph on the screen.
    numHostsGraph.display();

    // end Graph 4

    //Graph 5 - a graph showing the number of pathogens vs. time
    numPathogensGraph = new Plot("Number of pathogens", this);
    numPathogensGraph.setXRange( 0, 200 );
    numPathogensGraph.setYRange( 0, 1 );
    numPathogensGraph.setAxisTitles("Time", "Number of pathogens");

    numPathogensGraph.addLegend(1, "Number of pathogens");

    //tell the model about it.
    HostPathogen.setNumPathogensGraph(numPathogensGraph);

    // now actually display the graph on the screen.
    numPathogensGraph.display();
```

```
    // end Graph 5

}

// ////////////////////////////////////////////////////////
// buildSchedule
//
// This builds the entire schedule, i.e.,
// - the base model step
// - report step
// - display steps.

@Override
public void buildSchedule()
{

    if (rDebug > 0)
        System.out.printf("==> GUIModel buildSchedule...\n");

    // schedule the current GUIModel's step() function
    // to execute every time step starting with time step 0
    schedule.scheduleActionBeginning(0, this, "step");
    // start report at 1
    schedule.scheduleActionAtInterval(1, this, "stepReport", Schedule.LAST);

    // schedule the current GUIModel's processEndOfRun()
    // function to execute at the end of the run
    schedule.scheduleActionAtEnd(this, "processEndOfRun");
}

// ///////////////////////////////////////////////////////////////////////
// step
//
// executed each step of the model.
// Ask the super class to do its step() method,
// and then this does display related activities.
//
@Override
public void step()
{

    super.step(); // the model does whatever it does

    // add things after this for all displays (graphs, etc)
    dsurf.updateDisplay();
    offspringSurf.updateDisplay();
    descendentsSurf.updateDisplay();

    // update the graph every graphUpdatePeriod steps
    // note that getTickCount() and getGraphUpdatePeriod() are inherited
    if ((getTickCount() % getGraphUpdatePeriod())==0.0)
    {
        // automatically adjust the axes (this will automatically update the graph)
        transmissibilityGraph.fillPlot();
        virulenceGraph.fillPlot();
        growthGraph.fillPlot();
        numHostsGraph.fillPlot();
        numPathogensGraph.fillPlot();
        }

    }

    // processEndOfRun
    // called once, at end of run.
    @Override
    public void processEndOfRun()
    {
        if (rDebug > 0)
            System.out.printf("\n\n===== GUIModel processEndOfRun =====\n\n");
        applyAnyStoredChanges();
        endReportFile();
        this.fireStopSim();
    }

    // updateDisplay
    // if someone wants the dsurf redrawn...

    public void updateDisplay()
    {
        dsurf.updateDisplay();
        offspringSurf.updateDisplay();
        descendentsSurf.updateDisplay();
    }

    // ////////////////////////////////////////////////////////////////////
    // main entry point
    public static void main(String[] args)
    {

        uchicago.src.sim.engine.SimInit init = new uchicago.src.sim.engine.SimInit();
        GUIModel model = new GUIModel();

        // System.out.printf("==> GUIMOdel main...\n" );

        // set the type of model class, this is necessary
        // so the parameters object knows whether or not
        // to do GUI related updates of panels,etc when a
        // parameter is changed
        model.setModelType("GUIModel");

        // Do this to set the Update Probes option to true in the
        // Repast Actions panel
        AbstractGUIController.UPDATE_PROBES = true;

        model.setCommandLineArgs(args);
        init.loadModel(model, null, false); // does setup()

        // this new function calls ProbeUtilities.updateProbePanels() and
        // ProbeUtilities.updateModelProbePanel()
        model.updateAllProbePanels();

    }

}
```

BatchModel.java

```
package hostPathogen_v7;
import uchicago.src.sim.engine.*;

public class BatchModel extends HostPathogen
{

    // ///////////////////////////////////////////////////////////////////
    // main entry point
    public static void main(String[] args)
    {

        BatchModel model = new BatchModel();

        // set the type of model class, this is necessary
        // so the parameters object knows whether or not
        // to do GUI related updates of panels, etc when a
        // parameter is changed
        model.setModelType("BatchModel");

        model.setCommandLineArgs(args);

        PlainController control = new PlainController();
        model.setController(control);
        control.setExitOnExit(true);
        control.setModel(model);
        model.addSimEventListener(control);
        if (model.getRDebug() > 0)
            System.out.printf("\n==> BatchModel main...about to startSimulation...\n");
        control.startSimulation();
    }

    // setup() -- BatchModel just does what the super class does.
    @Override
    public void setup()
    {
        super.setup();
    }

    // begin()
    // ask the super class to do its building, then build a schedule.
    @Override
    public void begin()
    {
        // set schedule to null so buildModel knows not to
        // record changes ( changes are recorded if
        // schedule != null ). in buildSchedule() the
        // schedule is allocated before the actual schedule is created.
        schedule = null;
        buildModel(); // the base Model class does this
        buildSchedule();
    }

    // ///////////////////////////////////////////////////////////////
    // buildSchedule
    //
    // This may need to be changed, depending on what you want to
    // happen in a batch run (vs a GUI run).

    @Override
    public void buildSchedule()
    {

        schedule = new Schedule(1);

        // schedule the current BatchModel's step() function
        // to execute every time step starting with time step 0
        schedule.scheduleActionBeginning(0, this, "step");
        schedule.scheduleActionAtInterval(1, this, "stepReport", Schedule.LAST);

        // schedule the current BatchModel's processEndOfRun()
        // function to execute at the end of the Batch Run.
        // You need to specify the time to schedule it (instead
        // of doing scheduleActionAtEnd() or it will just run forever
        schedule.scheduleActionAt(getStopT(), this, "processEndOfRun");
    }

    // processEndOfRun
    // we need this to tell it to stop running!
    @Override
    public void processEndOfRun()
    {
        super.processEndOfRun();
        this.fireEndSim();
    }
}

// ////////////////////////////////////////////////////////////////////////////
// ///////////////////////////////////////////////////////////////////////////
// Why this class below?
//
// the reason we did that is because the repast "BatchController" had methods
// in it that started GUI stuff. this caused problems when we ssh'd into
// another machine and run a job--when we tried to disconnect, the ssh
// session would stay hung until the job was finished because the job needed
// the X11-forwarding to be open to run.
class PlainController extends BaseController
{
    private boolean exitonexit;

    public PlainController()
    {
        super();
        exitonexit = false;
    }

    public void startSimulation()
    {
        startSim();
    }

    public void stopSimulation()
    {
        stopSim();
    }
    public void exitSim()
    {
        exitSim();
    }

    public void pauseSimulation()
    {
        pauseSim();
    }

    @Override
    public boolean isBatch()
    {
        return true;
    }

    @Override
    protected void onTickCountUpdate()
    {
    }

    // this might not be necessary
    @Override
    public void setExitOnExit(boolean in_Exitonexit)
    {
        exitonexit = in_Exitonexit;
    }

    public void simEventPerformed(SimEvent evt)
    {
        if (evt.getId() == SimEvent.STOP_EVENT)
        {
            stopSimulation();
        } else if (evt.getId() == SimEvent.END_EVENT)
        {
            if (exitonexit)
            {
                System.exit(0);
            }
        } else if (evt.getId() == SimEvent.PAUSE_EVENT)
        {
            pauseSimulation();
        }
    }

    // function added because it is required for repast 2.2
    public long getRunCount()
    {
        return 0;
    }

    // function added because it is required for repast 2.2
    public boolean isGUI()
    {
        return false;
    }
}
```

ModelParameters.java

```
package hostPathogen_v7;

import java.io.*;
import java.util.*;
import java.util.Scanner;
import java.util.regex.*; // for MatchResult
import java.lang.reflect.*;

import uchicago.src.sim.engine.*;
import uchicago.src.sim.util.Random;
import uchicago.src.sim.util.*;

//for the xml parsing of input files
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Element;

import java.util.TreeMap;

public class ModelParameters extends SimModelImpl
{

    // setup
    // this should be called *last* in the Model setup() that
    // extends this class.
    public void setup()
    {
        changesVector = new Vector();
        setupParametersMap();
        // only process command line arguments if it is the first run
        // if it is the first run then schedule is null,
        // if not then schedule is initialized (and is set to null
        // on the next line)
        if (schedule == null)
            processCommandLinePars(commandLineArgs);
        schedule = null;

        if (rDebug > 0)
            System.out.printf("<--- ModelParameters setup() done.\n");
    }

    public void begin()
    {
        // this must be declared in the class that 'extends' this one
    }

    // buildModelStart
    // this should be called first by the buildModel in the extending class.
    public void buildModelStart()
    {
        if (getSeed() == 1234567 || getSeed() == 0)
        {
```

```
        long s = System.currentTimeMillis();
        setSeed(s);
        if (rDebug > 1)
            System.out.printf("\nseed was 1234567 or 0, now ==> s=%d\n", s);
    }
    if (rDebug > 1)
        System.out.printf("\nabout to setSeed(%d)\n", getSeed());
    resetRNGenerators();
}

// buildSchedule
// the extending classes must fill this in
public void buildSchedule()
{
    schedule = new Schedule();
}

public String[] getInitParam()
{
    // this must be declared in the class that 'extends' this one
    return null;
}

// Generic parameters
protected String initialParametersFileName = "";
protected String initialAgentsFileName = "";
protected String reportFileName = "report";
protected String outputDirName = "./";
protected int reportFrequency = 1;
protected int runNumber = 0;
protected int stopT = 100;
protected int rDebug = 0;
protected int saveRunEndState = 0;
protected long seed = 1234567;
protected PrintWriter reportFile, plaintextReportFile;
protected PrintWriter changesFile;

// other utilities
protected String[] commandLineArgs;
protected String modelType = "Model";

// for input file
protected boolean STRICT_FILE_FORMAT = true;
protected Vector changesVector;

// variables for processing run-time changes that are
// read in from the input file
protected int numberOfChanges = 0;
protected int nextChangeToDo = 0;
protected int[] changeSteps = new int[64];
protected int[] changeIDs = new int[64];
protected ArrayList changeSpecs = new ArrayList(16);

// required by SimModelImpl
protected BasicAction stepMethods;
protected Schedule schedule = null;

// setupParametersMap
// this implements the mapping from aliases to long names,
// for the 'base' parameters common to all models.
// For parameters for a particular model, add lines
// to addToParametersMap().
protected TreeMap parametersMap;

public void setupParametersMap()
{
    DMSG(1, "setupParametersMap()");

    parametersMap = null;
    parametersMap = new TreeMap();
    // generic model parameters
    parametersMap.put("D", "rDebug");
    parametersMap.put("S", "seed");
    parametersMap.put("iPFN", "initialParametersFileName");
    parametersMap.put("iAFN", "initialAgentsFileName");
    parametersMap.put("rFN", "reportFileName");
    parametersMap.put("T", "stopT");
    parametersMap.put("sRES", "saveRunEndState");
    parametersMap.put("oDN", "outputDirName");
    parametersMap.put("rF", "reportFrequency");
    parametersMap.put("rN", "runNumber");

    addModelSpecificParameters();
}

// addModelSpecificParameters
// a subclass should override this.
public void addModelSpecificParameters()
{
}

public void printParametersMap()
{

    ArrayList parameterNames = new ArrayList(parametersMap.values());
    ArrayList parameterAliases = new ArrayList(parametersMap.keySet());

    for (int i = 0; i < parameterAliases.size(); i++)
    {
        Method getmethod = null;
        String parAlias = (String) parameterAliases.get(i);
        String parName = (String) parametersMap.get(parAlias);

        getmethod = findGetMethodFor(parName);

        if (getmethod != null)
        {
            try
            {
                Object returnVal = getmethod.invoke(this, new Object[]
                {});
                String s = parName + " (" + parAlias + ") = " + returnVal;
                System.out.printf("%s\n", s);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
        } else
        {
            System.err.printf("COULD NOT FIND SET METHOD FOR:  %s\n", parameterNames.get(i));
            System.err.printf("Is the entry in the parametersMap for this correct?");
        }
    }

}

// ///////////////////////////////////////////////////////////////
// generic setters/getters

public String[] getCommandLineArgs()
{
    return commandLineArgs;
}

public void setCommandLineArgs(String[] arguments)
{
    commandLineArgs = arguments;
}

public String getModelType()
{
    return modelType;
}

public void setModelType(String s)
{
    modelType = s;
}

public String getInitialParametersFileName()
{
    return initialParametersFileName;
}

public void setInitialParametersFileName(String s)
{
    initialParametersFileName = s;
}

public String getInitialAgentsFileName()
{
    return initialAgentsFileName;
}

public void setInitialAgentsFileName(String s)
{
    initialAgentsFileName = s;
}

public String getReportFileName()
{
    return reportFileName;
}
```

```
public void setReportFileName(String s)
{
    reportFileName = s;
}

public String getOutputDirName()
{
    return outputDirName;
}

public void setOutputDirName(String s)
{
    outputDirName = s;
}

public int getReportFrequency()
{
    return reportFrequency;
}

public void setReportFrequency(int i)
{
    reportFrequency = i;
}

public int getRunNumber()
{
    return runNumber;
}

public void setRunNumber(int i)
{
    runNumber = i;
}

public int getStopT()
{
    return stopT;
}

public void setStopT(int i)
{
    stopT = i;
}

public int getSaveRunEndState()
{
    return saveRunEndState;
}

public void setSaveRunEndState(int i)
{
    saveRunEndState = i;
}

public int getRDebug()
{
    return rDebug;
}

public void setRDebug(int i)
{
    if (rDebug == i)
    {
        return;
    }
    rDebug = i;
    if (modelType.equals("GUIModel"))
    {
        updateAllProbePanels();
    }
    if (modelType.equals("GUIModel") && schedule != null)
        writeChangeToReportFile("rDebug", String.valueOf(i));
}

public long getSeed()
{
    return seed;
}

public void setSeed(long i)
{
    if (rDebug > 0)
        System.out.println("setSeed ( " + i + " ) called");
    seed = i;

    resetRNGenerators();

    if (modelType.equals("GUIModel"))
    {
        updateAllProbePanels();
    }
    if (modelType.equals("GUIModel") && schedule != null)
        writeChangeToReportFile("seed", String.valueOf(i));
}

public void resetRNGenerators()
{
    if (rDebug > 0)
        System.out.printf("\nresetRNGenerators with %d\n", getSeed());

    // this is required because once you change the seed you invalidate
    // any previously created distributions
    uchicago.src.sim.util.Random.setSeed(seed);
    uchicago.src.sim.util.Random.createUniform();
    uchicago.src.sim.util.Random.createNormal(0.0, 1.0);
}

// NOTE: these are class methods!
static public int getUniformIntFromTo(int low, int high)
{
    int randNum = Random.uniform.nextIntFromTo(low, high);
    return randNum;
}

static public double getNormalDouble(double mean, double var)
```

```
{
    double randNum = Random.normal.nextDouble(mean, var);
    return randNum;
}

static public double getUniformDoubleFromTo(double low, double high)
{
    double randNum = Random.uniform.nextDoubleFromTo(low, high);
    return randNum;
}

// loop until a number between 0 and 1 is generated,
// if mean and var are set correctly the loop will rarely happen
static public double getNormalDoubleProb(double mean, double var)
{
    if (mean < 0 || mean > 1)
    {
        System.out.println("Invalid value set for normal distribution mean");
        return -1;
    }
    double d = Random.normal.nextDouble(mean, var);
    while (d < 0 || d > 1)
        d = Random.normal.nextDouble(mean, var);
    return d;
}

public void setRngSeed(long i)
{
    System.out.println("setRngSeed ( " + i + " ) called");
    setSeed(i);
}

public PrintWriter getReportFile()
{
    return reportFile;
}

public PrintWriter getPlaintextReportFile()
{
    return plaintextReportFile;
}

public Schedule getSchedule()
{
    return schedule;
}

public String getName()
{
    return "ModelParameters";
}

// some generic utilities
public void updateAllProbePanels()
{
    DMSG(2, "updateAllProbePanels()");
    ProbeUtilities.updateProbePanels();
```

```
        // need this in case updateAllProbePanels gets called
        // before the probe panel is created (if it is called
        // before, then a RuntimeException occurs)
        // did have if(schedule != null), but that means panels
        // do not update at all during time=0, so people get confused.
        try
        {
            ProbeUtilities.updateModelProbePanel();
        } catch (RuntimeException e)
        {
            // ignore exception
            DMSG(3, "RuntimeException when updating model probe panel, ignoring ...");
        }
    }

    // captialize first character of s
    protected String capitalize(String s)
    {
        char c = s.charAt(0);
        char upper = Character.toUpperCase(c);
        return upper + s.substring(1, s.length());
    }

    // REPORT FILE PROCESSING ------------------------------
    //
    // startReportFile
    // opens two report files
    // one XML report file and one plaintext report file
    // call writeLineToReportFile to write to XML report file
    // and writeLineToPlaintextReportFile to write to plaintext file

    public PrintWriter startReportFile()
    {
        if (rDebug > 0)
            System.out.println("startReportFile called!");
        reportFile = null;
        plaintextReportFile = null;
        String fullFileName = reportFileName + String.format(".%02d", runNumber);
        String xmlFullFileName = reportFileName + ".xml" + String.format(".%02d", runNumber);

        // BufferedReader inFile =
        // IOUtils.openFileToRead(initialParametersFileName);

        reportFile = IOUtils.openFileToWrite(outputDirName, xmlFullFileName, "r");
        plaintextReportFile = IOUtils.openFileToWrite(outputDirName, fullFileName, "r");

        // the first line you have to write is the XML version line
        // DO NOT WRITE THIS LINE USING writeLineToReportFile()!
        reportFile.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");

        writeLineToReportFile("<reportfile>");
        writeLineToPlaintextReportFile("# begin reportfile");

        // write the initial parameters to the report file
        writeParametersToReportFile();

        writeHeaderCommentsToReportFile(); // the user must define this!
        return reportFile;
    }

    public void writeLineToReportFile(String line)
    {
        if (reportFile == null)
        {
            DMSG(3, "report file not opened yet");
            // click the initialize button to open it!
            // returning w/o writing to report file ...");
            return;
        } else
        {
            reportFile.println(line);
        }
    }

    public void writeLineToPlaintextReportFile(String line)
    {
        if (plaintextReportFile == null)
        {
            DMSG(3, "report file not opened yet");
            // click the initialize button to open it!
            // returning w/o writing to report file ...");
            return;
        } else
        {
            plaintextReportFile.println(line);
        }
    }

    public void writeChangeToReportFile(String varname, String value)
    {
        DMSG(1, "writeChangeToReportFile(): write change to report file: " + varname + " changed to " +
value);

        writeLineToReportFile("<change>");
        writeLineToReportFile("\t<" + varname + ">" + value + "</" + varname + ">");
        String s = String.format("\t<time>%.0f</time>", getTickCount());
        writeLineToReportFile(s);
        writeLineToReportFile("</change>");

        writeLineToPlaintextReportFile("# change:  " + varname + "=" + value);
    }

    public void endReportFile()
    {
        writeLineToReportFile("</reportfile>");
        writeLineToPlaintextReportFile("# end report file");
        IOUtils.closePWFile(reportFile);
        IOUtils.closePWFile(plaintextReportFile);
    }

    // this iterates through the values stored in the parametersMap,
    // calls the getter on each parameter, and outputs the
    // parameter and its value to the report file.
    // this is called right before the model run starts (after all
    // initial parameters are changed!) so
    // the initial parameters are in the report file.
    public void writeParametersToReportFile()
    {
        DMSG(1, "writeParametersToReportFile()");

        writeLineToReportFile("<parameters>");
        writeLineToPlaintextReportFile("# begin parameters");

        ArrayList parameterNames = new ArrayList(parametersMap.values());
        for (int i = 0; i < parameterNames.size(); i++)
        {
            Method getmethod = null;
            getmethod = findGetMethodFor((String) parameterNames.get(i));

            if (getmethod != null)
            {
                try
                {
                    Object returnVal = getmethod.invoke(this, new Object[]
                    {});

                    writeLineToReportFile("\t<" + parameterNames.get(i) + ">" + returnVal + "</" +
                       parameterNames.get(i) + ">");

                    writeLineToPlaintextReportFile(parameterNames.get(i) + "=" + returnVal);

                } catch (Exception e)
                {
                    e.printStackTrace();
                }
            } else
            {
                System.err.printf("COULD NOT FIND SET METHOD FOR:  %s\n", parameterNames.get(i));
                System.err.printf("Is the entry in the parametersMap for this correct?");
            }
        }
        writeLineToReportFile("</parameters>");
        writeLineToPlaintextReportFile("# end parameters");
    }

    // ///////////////////////////////////////////////////////////////////////////////
    //
    // Generic report file processing ------------------------------
    //
    // These are similar to those above, but these require the user
    // to specify a particular "basename" for the files, and they
    // require/allow the user to separately open/writeTo/close the xml and plain
    // text files.
    //
    // PrintWriter startReportFile ( String baseName ) -- an xml formated report
    // file
    // PrintWriter startPlainTextReportFile ( String baseName ) -- plain text
    // report file
    //
    // void writeParametersToReportFile( PrintWriter rfile )
    // void writeParametersToPlainTextReportFile( PrintWriter rfile )
    //
    // void writeLineToReportFile ( String line, PrintWriter rfile )
```

```
//
// void endReportFile ( PrintWriter rfile )
// void endPlainTextReportFile ( PrintWriter rfile )
//

public PrintWriter startReportFile(String baseName)
{
    if (rDebug > 0)
        System.err.printf("startReportFile called for baseName='%s'\n", baseName);
    PrintWriter rFile = null;
    String xmlFullFileName = baseName + ".xml" + String.format(".%02d", runNumber);

    rFile = IOUtils.openFileToWrite(outputDirName, xmlFullFileName, "r");

    // the first line you have to write is the XML version line
    // DO NOT WRITE THIS LINE USING writeLineToReportFile(...)!
    rFile.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");

    writeLineToReportFile("<reportfile>", rFile);
    // write the initial parameters to the report file
    writeParametersToReportFile(rFile);

    return rFile;
}

public PrintWriter startPlainTextReportFile(String baseName)
{
    if (rDebug > 0)
        System.err.printf("startPlainTextReportFile called for baseName='%s'\n", baseName);
    PrintWriter rFile = null;
    String fullFileName = baseName + String.format(".%02d", runNumber);

    rFile = IOUtils.openFileToWrite(outputDirName, fullFileName, "r");

    writeLineToReportFile("# begin reportfile", rFile);

    // write the initial parameters to the report file
    writeParametersToPlainTextReportFile(rFile);

    return rFile;
}

// this just writes whatever is sent to it, and then a newline!
public void writeLineToReportFile(String line, PrintWriter rFile)
{
    if (rFile == null)
    {
        System.err.printf("\nERROR - A user-defined report file not opened yet!\n");
        return;
    } else
    {
        rFile.println(line);
    }
}

// the following does NOT write a newline!
public void writeBufferToReportFile(String line, PrintWriter rFile)
{
    if (rFile == null)
    {
        System.err.printf("\nERROR - A user-defined report file not opened yet!\n");
        return;
    } else
    {
        rFile.printf(line);
    }
}

public void endReportFile(PrintWriter rFile)
{
    writeLineToReportFile("</reportfile>", rFile);
    IOUtils.closePWFile(rFile);
}

public void endPlainTextReportFile(PrintWriter rFile)
{
    writeLineToReportFile("# end report file", rFile);
    IOUtils.closePWFile(rFile);
}

// these iterate through the values stored in the parametersMap,
// calls the getter on each parameter, and outputs the
// parameter and its value to the report file.
// this is called right before the model run starts (after all
// initial parameters are changed!) so
// the initial parameters are in the report file.
public void writeParametersToReportFile(PrintWriter rFile)
{
    DMSG(1, "writeParametersToReportFile( rFile )");
    writeLineToReportFile("<parameters>", rFile);
    ArrayList parameterNames = new ArrayList(parametersMap.values());
    for (int i = 0; i < parameterNames.size(); i++)
    {
        Method getmethod = null;
        getmethod = findGetMethodFor((String) parameterNames.get(i));
        if (getmethod != null)
        {
            try
            {
                Object returnVal = getmethod.invoke(this, new Object[]
                {});
                writeLineToReportFile("\t<" + parameterNames.get(i) + ">" + returnVal + "</" +
                   parameterNames.get(i) + ">", rFile);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
        } else
        {
            System.err.printf("COULD NOT FIND SET METHOD FOR:  %s\n", parameterNames.get(i));
            System.err.printf("Is the entry in the parametersMap for this correct?");
        }
    }
    writeLineToReportFile("</parameters>", rFile);
}
```

```
public void writeParametersToPlainTextReportFile(PrintWriter rFile)
{
    DMSG(1, "writeParametersToPlainTextReportFile( rFile )");
    writeLineToReportFile("# begin parameters", rFile);
    ArrayList parameterNames = new ArrayList(parametersMap.values());
    for (int i = 0; i < parameterNames.size(); i++)
    {
        Method getmethod = null;
        getmethod = findGetMethodFor((String) parameterNames.get(i));
        if (getmethod != null)
        {
            try
            {
                Object returnVal = getmethod.invoke(this, new Object[]
                {});
                writeLineToReportFile(parameterNames.get(i) + "=" + returnVal, rFile);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
        } else
        {
            System.err.printf("COULD NOT FIND SET METHOD FOR:  %s\n", parameterNames.get(i));
            System.err.printf("Is the entry in the parametersMap for this correct?");
        }
    }
    writeLineToReportFile("# end parameters", rFile);
}

// ------> End of Report File Processing <------------------------------

// ///////////////////////////////////////////////////////////////////////
//
// ------> Input Parameter Processing <------------------------------
//
// ///////////////////////////////////////////////////////////////////////
// parseParametersFile
//
public void parseParametersFile()
{
    // a klunky way to see if the parameters file exists
    try
    {
        BufferedReader inFile = IOUtils.openFileToRead(initialParametersFileName);
        IOUtils.closeBRFile(inFile);
    } catch (Exception e)
    { // not an error, just not there!
        if (rDebug > 0)
            System.err.printf("  -- no initialParametersFileName '%s' to parse.\n",
                    initialParametersFileName);
        return;
    }

    try
    {
        // setup the input file
        DocumentBuilderFactory myDBF = DocumentBuilderFactory.newInstance();
        DocumentBuilder myDB = myDBF.newDocumentBuilder();
```

```
            Document myDocument = myDB.parse(initialParametersFileName);

            if (rDebug > 0)
                System.out.println("Parsing parameter file: " + initialParametersFileName);

            NodeList tmpList = myDocument.getElementsByTagName("parameters");
            Element tmpElement = (Element) tmpList.item(0);
            NodeList parameterList = tmpElement.getElementsByTagName("*");

            for (int i = 0; i < parameterList.getLength(); i++)
            {
                if (parameterList.item(i).getChildNodes().item(0) == null)
                    continue;
                DMSG(1, "name:  " + parameterList.item(i).getNodeName() + "  value:  " +
                       parameterList.item(i).getChildNodes().item(0).getNodeValue());
                set(parameterList.item(i).getNodeName(), parameterList.item(i).getChildNodes().item
                       (0).getNodeValue());
            }

            // process changes
            NodeList parameterChangeList = myDocument.getElementsByTagName("change");
            processChangeList(parameterChangeList);

            DMSG(1, "Done parsing file:  " + initialParametersFileName);
        } catch (Exception e)
        {
            System.out.println("Exception when parsing parameters file:  " + initialParametersFileName);
            System.out.println("Is the file in the correct format?");
            e.printStackTrace();
        }
    }

    // ////////////////////////////////////////////////////////////////////
    // processCommandLinePars
    // storeParameter
    //
    public void processCommandLinePars(String[] args)
    {
        int r;
        if (args.length > 0 && (args[0].equals("--help") || args[0].equals("-h")))
        {
            printProjectHelp();
        }

        for (int i = 0; i < args.length; ++i)
        {
            r = storeParameter(args[i]);
            if (r != 0)
            {
                System.out.println("Error processing cmdLine par:  " + args[i]);
            }
        }
    }

    // storeParameter
    // format: parname=value
    // parse out parname, and find method for setParname
    // if not found, return -1
```

```
// otherwise set the value and return 0.
// to set the value, we have to get the setMethod, and its par type.
// then convert the string value to the appropriate object, and
// use invoke to do the setting!

public int storeParameter(String line)
{
    int r = 0;
    String pname, pvalue;
    StringTokenizer st = new StringTokenizer(line, "=;,");
    Method setm = null;

    if ((pname = st.nextToken()) == null)
    {
        System.err.printf("\n** storeParameter -- couldn't find pname on '%s'.\n", line);
        return -1;
    }
    if ((pvalue = st.nextToken()) == null)
    {
        System.err.printf("\n** storeParameter -- couldn't find value on '%s'.\n", line);
        return -1;
    }
    pname = pname.trim();
    pvalue = pvalue.trim();

    pname = aliasToParameterName(pname);

    // if this is a scheduledChange, create the change
    // and insert it into the changesVector
    if (pname.equals("sC"))
    {
        String changetime = pvalue;
        String changepname, changepvalue;

        if ((changepname = st.nextToken()) == null)
        {
            System.out.println("\n** storeParameter -- couldn't find " + "scheduleChange pname on:  " +
                    line);
            return -1;
        }

        if ((changepvalue = st.nextToken()) == null)
        {
            System.out.println("\n** storeParameter -- couldn't find " + "scheduleChange pvalue on:  " +
                    line);
            return -1;
        }

        changepname = changepname.trim();
        changepvalue = changepvalue.trim();

        changepname = aliasToParameterName(changepname);

        ChangeObj newChange = new ChangeObj(Integer.parseInt(changetime), changepname,
                changepvalue);

        DMSG(1, "scheduledChange from command line created:  " + "  Time:  " + changetime + "
                 pname:  " + changepname + "  pvalue:  " + changepvalue);
        changesVector.add(newChange);

        return 0;
    }

    setm = findSetMethodFor(pname);
    String ptype = getParTypeOfSetMethod(setm);

    try
    {
        setm.invoke(this, new Object[]
        { valToObject(ptype, pvalue) });
    } catch (Exception e)
    {
        System.err.printf("\n storeParameter: '%s'='%s' invoke exception!\n", pname, pvalue);

        System.err.printf("  --> %s\n", e.toString());
        e.printStackTrace();
        return -1;
    }

    if (pname.equals("initialParametersFileName"))
    {
        DMSG(1, "Processing initial parameters file:  " + pvalue);
        parseParametersFile();
    }

    return r;
}

// returns the long parameter name if the parameter passed in is
// an alias. if it is not an alias, the name sent to it is returned.
public String aliasToParameterName(String alias)
{
    // check to see if "alias" is an alias in the parametersMap
    // if it is then "alias" is a valid alias, so set "alias" to the
    // actual parameter name that is in the map
    if (parametersMap.containsKey(alias))
    {
        DMSG(1, "Converting alias " + alias + " to " + parametersMap.get(alias));
        alias = (String) parametersMap.get(alias);
    }

    return alias;
}

// getParTypeOfSetMethod
// get type of setPar method parameter
public String getParTypeOfSetMethod(Method m)
{
    Class[] parTypes = m.getParameterTypes();
    String s = parTypes[0].getName();
    return s;
}

// findGetMethodFor
// find get<ParName> method for specified parameter name
protected Method findGetMethodFor(String varname)
```

```
{
    String methodname = new String("get" + capitalize(varname));
    Class c = getClass();
    Method[] methods = c.getMethods();
    Method getmethod = null;

    for (int j = 0; j < methods.length; j++)
    {
        if (methods[j].getName().equals(methodname))
        {
            getmethod = methods[j];
            break;
        }
    }
    if (getmethod == null)
    {
        System.err.printf("\n** findGetMethodFor -- couldn't find '%s'\n", methodname);
        return getmethod;
    }

    return getmethod;
}

// findSetMethodFor
// find set<ParName> method for specified parameter name
public Method findSetMethodFor(String pname)
{
    Class c = this.getClass();
    Method[] methods = c.getMethods();
    int nf = methods.length;
    String setmethodname = "set" + capitalize(pname);
    String mname;
    Method method = null;
    for (int i = 0; i < nf; ++i)
    {
        mname = methods[i].getName();
        if (mname.equals(setmethodname))
        {
            method = methods[i];
            break;
        }
    }
    if (method == null)
    {
        System.err.printf("\n** findSetMethodFor -- couldn't fine '%s'\n", setmethodname);
        return method;
    }
    return method;
}

// valToObject
// return value stored in object of appropriate type
private Object valToObject(String type, String val)
{
    if (type.equals("int"))
    {
        return Integer.valueOf(val);
    } else if (type.equals("double"))
    {
        return Double.valueOf(val);
    } else if (type.equals("float"))
    {
        return Float.valueOf(val);
    } else if (type.equals("long"))
    {
        return Long.valueOf(val);
    } else if (type.equals("boolean"))
    {
        return Boolean.valueOf(val);
    } else if (type.equals("java.lang.String"))
    {
        return val;
    } else
    {
        throw new IllegalArgumentException("illegal type");
    }
}

public String skipCommentLines(BufferedReader inFile)
{
    String line;
    while ((line = IOUtils.readBRLine(inFile)) != null)
    {
        if (line.charAt(0) != '#')
            break;
    }
    return line;
}

// ///////////////////////////////////////////////////////////////////////////
// applyAnyStoredChanges
// look through all of the changes, if any have time of this time
// step execute the change
public void applyAnyStoredChanges()
{
    if (rDebug > 0)
    {
        System.out.println("applyAnyStoredChanges called at time step: " + getTickCount());
    }

    for (int i = 0; i < changesVector.size(); i++)
    {
        ChangeObj tmpObj = (ChangeObj) changesVector.get(i);
        if (tmpObj.time == getTickCount())
        {
            if (rDebug > 0)
            {
                System.out.println("applyAnyStoredChanges():  Changing " + tmpObj.varname + " to " +
                   tmpObj.value);
            }
            set(tmpObj.varname, tmpObj.value);
        }
    }
}

// //////////////////////////////////////////////////////////////////////
// utility methods for accessing parts of model

private void setObjectParameter(Object inObject, String varname, String value)
{
    String methodname = new String("set" + capitalize(varname));
    Class c = inObject.getClass();
    Method[] methods = c.getMethods();
    Method setmethod = null;

    for (int j = 0; j < methods.length; j++)
    {
        if (methods[j].getName().equals(methodname))
        {
            setmethod = methods[j];
            break;
        }
    }

    if (setmethod != null)
    {
        try
        {
            Class[] parameterTypes = setmethod.getParameterTypes();
            if (parameterTypes[0].getName().equals("int"))
            {
                DMSG(3, "int parameter type");
                setmethod.invoke(inObject, new Object[]
                { Integer.valueOf(value) });
            } else if (parameterTypes[0].getName().equals("long"))
            {
                DMSG(3, "long parameter type");
                setmethod.invoke(inObject, new Object[]
                { Long.valueOf(value) });
            } else if (parameterTypes[0].getName().equals("double"))
            {
                DMSG(3, "double parameter type");
                setmethod.invoke(inObject, new Object[]
                { Double.valueOf(value) });
            } else if (parameterTypes[0].getName().equals("float"))
            {
                DMSG(3, "float parameter type");
                setmethod.invoke(inObject, new Object[]
                { Float.valueOf(value) });
            } else if (parameterTypes[0].getName().equals("java.lang.String"))
            {
                DMSG(3, "String parameter type");
                setmethod.invoke(inObject, new Object[]
                { value });
            } else
            {
                System.out.println("COULD NOT DETERMINE PARAMETER TYPE");
            }
            DMSG(1, "setObjectParameter():  " + varname + " changed to " + value);
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    } else
```

```
        {
            System.out.println("COULD NOT FIND SET METHOD FOR:  " + varname);
        }
    }

    private void processChange(Element c)
    {

        DMSG(3, "Processing A Change");

        NodeList tmpList = c.getElementsByTagName("*");

        ChangeObj newChange = new ChangeObj(0, "", "");

        for (int i = 0; i < tmpList.getLength(); i++)
        {
            Element tmpElement = (Element) tmpList.item(i);
            // System.out.println("tmpElement.getTagName(): " +
            // tmpElement.getTagName());
            if (tmpElement.getTagName().equals("time"))
            {
                newChange.time = Integer.parseInt(tmpElement.getChildNodes().item(0).getNodeValue());
            } else
            {
                newChange.varname = tmpElement.getTagName();
                newChange.value = tmpElement.getChildNodes().item(0).getNodeValue();
            }
        }

        changesVector.add(newChange);

        DMSG(3, "Done processing a Change");
    }

    private void processChangeList(NodeList c)
    {

        DMSG(3, "Processing " + c.getLength() + " changes ...");
        for (int i = 0; i < c.getLength(); i++)
            processChange((Element) c.item(i));

        for (int i = 0; i < changesVector.size(); i++)
        {
            ChangeObj tmpObj = (ChangeObj) changesVector.get(i);
            DMSG(3, "Time:  " + tmpObj.time + "  VarName:  " + tmpObj.varname + "  Value:  " +
                        tmpObj.value);
        }
    }

    private void set(String varname, String value)
    {

        // first convert varname to the alias, if it is an alias
        varname = aliasToParameterName(varname);

        Method setmethod = findSetMethodFor(varname);

        if (setmethod != null)
        {
            try
            {
                Class[] parameterTypes = setmethod.getParameterTypes();
                if (parameterTypes[0].getName().equals("int"))
                {
                    DMSG(3, "int parameter type");
                    setmethod.invoke(this, new Object[]
                    { Integer.valueOf(value) });
                } else if (parameterTypes[0].getName().equals("long"))
                {
                    DMSG(3, "long parameter type");
                    setmethod.invoke(this, new Object[]
                    { Long.valueOf(value) });
                } else if (parameterTypes[0].getName().equals("float"))
                {
                    DMSG(3, "float parameter type");
                    setmethod.invoke(this, new Object[]
                    { Float.valueOf(value) });
                } else if (parameterTypes[0].getName().equals("double"))
                {
                    DMSG(3, "double parameter type");
                    setmethod.invoke(this, new Object[]
                    { Double.valueOf(value) });
                } else if (parameterTypes[0].getName().equals("java.lang.String"))
                {
                    DMSG(3, "String parameter type");
                    setmethod.invoke(this, new Object[]
                    { value });
                } else
                {
                    System.out.println("COULD NOT DETERMINE PARAMETER TYPE");
                }
                DMSG(1, "set():  " + varname + " changed to " + value);
            } catch (Exception e)
            {
                e.printStackTrace();
            }
        } else
        {
            System.out.println("COULD NOT FIND SET METHOD FOR:  " + varname);
            System.out.println("Is the parameter name correct?");
        }
    }

    // loadChangeParameters
    // we expect to see
    // @changeParameters
    // step=<timeStep>
    // parName=parValue
    // ...
    // @endChangeParameters
    // <timeStep> is time step changes are to occur.
    // store in changeSteps[numberOfChanges]
    // store number of parameters to change in changeIDs[numberOfChanges]
    // increment numberOfChanges
    // Return 0 if ok, 1 if not. next line will be after @endChangeParameters
    public int loadChangeParameters(BufferedReader inFile)
```

```
{
    ArrayList lines = new ArrayList(16);
    String line, ends = "@endChangeParameters";
    int r, step = 0, numPars = 0, done = 0;
    if (rDebug > 0)
        System.out.printf("\n\n*** loadChangeParameters \n\n");

    // first get the step= line, and the time and ID values
    line = skipCommentLines(inFile);
    if (rDebug > 0)
        System.out.printf("0: %s\n", line);

    /*
     * was r = Format.sscanf( line, "step=%i", p.add(iV) ); step =
     * iV.intValue();
     */
    Scanner scanner = new Scanner(line);
    scanner.findInLine("step=(\\d+)");
    MatchResult result = scanner.match();
    try
    {
        step = Integer.parseInt(result.group());
    } catch (NumberFormatException e)
    {
    }

    // get lines into a bunch of strings, add to list of these sets of
    // lines.
    while (done == 0)
    {
        line = skipCommentLines(inFile);
        if (line.equals(ends))
            done = 1;
        else
        {
            // *** It would be nice to check these here...
            lines.add(line);
            ++numPars;
        }
    }
    changeSpecs.add(lines);

    if (numPars == 0)
    {
        System.err.printf("\n*** loadChangeParameters found 0 changes! Last line='%s'\n", line);
        return -1;
    }

    // store time and id in next place in arrays.
    changeSteps[numberOfChanges] = step;
    changeIDs[numberOfChanges] = 0 - numPars;
    ++numberOfChanges;

    for (int c = 0; c < numberOfChanges; ++c)
    {
        if (changeIDs[c] >= 0)
            continue;
        lines = (ArrayList) changeSpecs.get(c);
```

```
                System.out.printf("Change %d at t=%d, ID=%d:\n", c, changeSteps[c], changeIDs[c]);
                for (int i = 0; i < numPars; ++i)
                {
                    System.out.printf("%d: %s\n", i + 1, (String) lines.get(i));
                }
            }

            return 0;
        }

        public void DMSG(int debugLevel, String debugStr)
        {
            if (rDebug >= debugLevel)
            {
                System.out.println("debug:\t" + debugStr);
            }
        }

        // //////////////////////////////////////////////////////////////
        // printProjectHelp
        // this could be filled in with some help to get from running with -help
        // parameter
        //
        public void printProjectHelp()
        {
            // this is declared in the class that 'extends' this one
        }

        // //////////////////////////////////////////////////////////////
        // writeHeaderCommentsToReportFile
        // include comments to be written just after the list of parameter
        // values and just before the step-by-step data lines.

        public void writeHeaderCommentsToReportFile()
        {
            // this is declared in the class that 'extends' this one
        }

    }

    // //////////////////////////////////////////////////////////////////////
    // //////////////////////////////////////////////////////////////////////
    // auxilliary classes for processing changes
    //
    //
    class ChangeObj
    {
        public ChangeObj()
        {
        }

        public ChangeObj(int in_time, String in_varname, String in_value)
        {
            time = in_time;
            varname = in_varname;
            value = in_value;
        }
        public int time;
        public String varname;
        public String value;
    }

    class ACChangeObj
    {
        public ACChangeObj()
        {
        }

        public ACChangeObj(int in_time, int in_id, String in_varname, String in_value)
        {
            time = in_time;
            id = in_id;
            varname = in_varname;
            value = in_value;
        }

        public int time;
        public int id;
        public String varname;
        public String value;
    }

    // ///////////////////////////////////////////////////////////////////////////////
    // ///////////////////////////////////////////////////////////////////////////////
    // auxilliary class for file opening/closing
    // and string processing
    //
    class IOUtils
    {

        public static String readBRLine(BufferedReader file)
        {
            String s;
            try
            {
                s = file.readLine();
            } catch (IOException e)
            {
                // System.out.println( "closeBRFile error!" );
                s = null;
            }
            return s;
        }

        public static BufferedReader openFileToRead(String filename)
        {
            BufferedReader in;
            try
            {
                in = new BufferedReader(new FileReader(filename));
            } catch (IOException e)
            {
                // no file, etc
                // System.out.println( "openFileToRead error on filename="+filename
                // );
                in = null;
            }
            // System.err.printf("openFileToRead: '%s'\n", filename );
            return in;
        }

        public static PrintWriter openFileToWrite(String dir, String filename, String how)
        {
            PrintWriter out;
            try
            {
                File f = new File(dir, filename);
                out = new PrintWriter(new FileWriter(f));
            } catch (IOException e)
            {
                // no file, etc
                // System.out.println( "openFileToWrite error on dir/filename="
                // + dir + "/" + filename );
                out = null;
            }
            // System.err.printf("openFileToWrite: '%s'\n", filename );
            return out;
        }

        public static int closeBRFile(BufferedReader file)
        {
            int r = 0;
            try
            {
                file.close();
            } catch (IOException e)
            {
                // System.out.println( "closeBRFile error!" );
                r = -1;
            }
            return r;
        }

        public static int closePWFile(PrintWriter file)
        {
            int r = 0;
            file.close();
            return r;
        }

        // ////////////////////////////////////////////////////////////

        public static int tokenToInt(String token)
        {
            int i;
            token = token.trim();
            try
            {
                i = Integer.parseInt(token);
            } catch (NumberFormatException ex)
            {
                throw new IllegalArgumentException(" tokenToInt error, token=" + token);
            }
```

```
        return i;
    }

    public static double tokenToDouble(String token)
    {
        double d;
        token = token.trim();
        try
        {
            d = Double.parseDouble(token);
        } catch (NumberFormatException ex)
        {
            throw new IllegalArgumentException(" tokenToDouble error, token=" + token);
        }
        return d;
    }

}
```

Appendix B

Computer code for Chapter VI: Self-organization of background habitat determines the nature of population spatial structure

Software specifications

MATLAB R2008bSV

Script list

run_metapopulation.m 223

create_metapopulation.m 228

count_clumps_continuous.m 229

synth_clump_dist.m 232

disperse_clumps.m 235

Script details

run_metapopulation.m

```
% a function to run the metapopulation created by create_metapopulation.m
function [clumps] = run_metapopulation(in_mat, plot_each_step, e_0, e_1, m_0, m_1, equation_type)

RandStream.setDefaultStream (RandStream('mt19937ar','seed',sum(100*clock)));

[x y] = convert_matrix_to_x_y(in_mat);

[rows cols] = size(in_mat);

clump_radius = 1.5;
min_x = 1;
max_x = cols;
plot_clumps = 0;
[clump_size frequency clumps perc_LR] = count_clumps_continuous(x, y, clump_radius, min_x, max_x,
                                                  plot_clumps);

[clumps] = create_metapopulation(clumps);

% only run the metapopulation if there are more than 1 clumps
if length(clumps)>1

    % 0 = only the nearest occupied neighbor can rescue an extinct clump
    % 1 = all other clumps can rescue an extinct clump
```

```
all_neighbor = 0;

end_time = 1000;

for i = 1:length(clumps)
    clumps(i).occupied = zeros(end_time, 1);
    clumps(i).occupied(1) = 1;
end

if plot_each_step
  plot_handle = figure();
  set(gca, 'FontSize', 14);
  frac_handle = figure();
  set(gca, 'FontSize', 14);
end

num_occupied = zeros(end_time, 1);
fraction_occupied = zeros(end_time, 1);
for time = 2:end_time

    % loop through every clump
    for i = 1:length(clumps)

        clumps(i).migration_prob = 0;
        clumps(i).extinction_prob = 0;
        clumps(i).died = 0;
        clumps(i).rescued = 0;

        if(clumps(i).occupied(time-1))

            % determine if the clump goes extinct
            switch equation_type
                case 0
                    % linear
                    clumps(i).extinction_prob = min(1,max(0,e_0 + e_1*clumps(i).clump_size));
                case 1
                    % negative exponential
                    clumps(i).extinction_prob = min(1,max(0,e_0*exp(-e_1*clumps(i).clump_size)));
            end

            if (rand() < clumps(i).extinction_prob)
                clumps(i).died = 1;
            end
        else
            % determine if the clump becomes occupied
            switch all_neighbor
                case 0
                    % first, find the nearest occupied neighbor
                    for j = 1:length(clumps(i).neighbor)
                        if clumps(clumps(i).neighbor(j)).occupied(time-1) == 1
                            dist_nearest_neighbor = clumps(i).neighbor_distance(j);
                            break;
                        end
                    end

                    switch equation_type
                        case 0
                            % linear
                        migration_prob = m_0 + m_1*dist_nearest_neighbor;
                    case 1
                        % negative exponential
                        migration_prob = m_0*exp(-m_1*dist_nearest_neighbor);
                end

            case 1
                no_migration_prob = 1;
                for j = 1:length(clumps(i).neighbor)
                    if clumps(clumps(i).neighbor(j)).occupied(time-1) == 1

                        switch equation_type
                            case 0
                                % linear
                                temp_migration_prob = min(max(0,m_0 + m_1*clumps
                                      (i).neighbor_distance(j)),1);
                            case 1
                                % negative exponential
                                temp_migration_prob = min(1, max(0,m_0*exp(-m_1*clumps
                                      (i).neighbor_distance(j))));
                        end
                        no_migration_prob = no_migration_prob*(1-temp_migration_prob);
                    end
                end
                migration_prob = 1 - no_migration_prob;
            otherwise
                disp('Invalid value of all_neighbors');
        end
        clumps(i).migration_prob = migration_prob;
        if (rand() < migration_prob)
            clumps(i).rescued = 1;
        end
    end
end

% update the state of each clump
for i = 1:length(clumps)
    if clumps(i).died
        clumps(i).occupied(time) = 0;
    elseif clumps(i).rescued
        clumps(i).occupied(time) = 1;
    else
        clumps(i).occupied(time) = clumps(i).occupied(time - 1);
    end
end

% calculate stats and plot metapopulation
num_occupied(time) = 0;
occupied_index = 0;
empty_index = 0;
occupied_x = [];
occupied_y = [];
empty_x = [];
empty_y = [];
for i=1:length(clumps)

    if clumps(i).occupied(time) == 1;
        num_occupied(time) = num_occupied(time) + 1;
```

```
                for j = 1:length(clumps(i).x)
                    occupied_index = occupied_index + 1;
                    occupied_x(occupied_index) = clumps(i).x(j);
                    occupied_y(occupied_index) = clumps(i).y(j);
                end
            else
                for j = 1:length(clumps(i).x)
                    empty_index = empty_index + 1;
                    empty_x(empty_index) = clumps(i).x(j);
                    empty_y(empty_index) = clumps(i).y(j);
                end
            end
        end

        fraction_occupied(time) = num_occupied(time)./length(clumps);

        if plot_each_step
            figure(plot_handle);
            scatter(occupied_x, occupied_y, 15, 'r','filled');
            hold on;
            scatter(empty_x, empty_y, 15, 'b', 'filled');
            axis([0 cols 0 rows]);
            title(['e0=' num2str(e_0) ', e1=' num2str(e_1) ', m0=' num2str(m_0) ', m1=' num2str(m_1)...
                ', red = occupied, blue = empty']);
            hold off;

            figure(frac_handle);
            plot(fraction_occupied);
            title(['Fraction of clumps occupied, e0=' num2str(e_0) ', e1=' num2str(e_1)...
                ', m0=' num2str(m_0) ', m1=' num2str(m_1)],  'FontSize', 14);
            xlabel('Time', 'FontSize', 14);
            ylabel('Fraction of clumps occupied', 'FontSize', 14);
        end

    end

    if plot_each_step
      scatter(occupied_x, occupied_y, 15, 'r','filled');
      hold on;
      scatter(empty_x, empty_y, 15, 'b', 'filled');
      axis([0 cols 0 rows]);
      title(['e0=' num2str(e_0) ', e1=' num2str(e_1) ', m0=' num2str(m_0) ', m1=' num2str(m_1)...
          ', red = occupied, blue = empty']);
      hold off;

      figure(frac_handle);
      plot(fraction_occupied);
      title(['Fraction of clumps occupied, e0=' num2str(e_0) ', e1=' num2str(e_1)...
          ', m0=' num2str(m_0) ', m1=' num2str(m_1)],  'FontSize', 14);
      xlabel('Time', 'FontSize', 14);
      ylabel('Fraction of clumps occupied', 'FontSize', 14);
    end

    clumps(1).fraction_occupied = fraction_occupied;
else
    % put in some placeholders
    clumps(1).occupied = [];
    clumps(1).neighbor = [];
    clumps(1).neighbor_distance = [];
    clumps(1).migration_prob = [];
    clumps(1).extinction_prob = [];
    clumps(1).died = [];
    clumps(1).rescued = [];
    clumps(1).fraction_occupied = [];
end
```

create_metapopulation.m

```
% a script to determine the sizes of clusters and the minimum distance between clusters
% for an ant occupancy matrix
function [clumps] = create_metapopulation(clumps)

if length(clumps)>1

    for i = 1:length(clumps)

        neighbor_index = 0;
        for j = 1:length(clumps)

            if j ~= i

                neighbor_index = neighbor_index + 1;

                % calculate the distance from this clump to the next one
                clumps(i).neighbor(neighbor_index) = j;
                clumps(i).neighbor_distance(neighbor_index) = 1E99;

                % loop through every point in both clumps
                for k = 1:length(clumps(i).x)
                    for m = 1:length(clumps(j).x)

                        % calculate the distance between the points
                        distance = ((clumps(i).x(k) - clumps(j).x(m))^2 + (clumps(i).y(k) - clumps(j).y(m))^2)
                                ^0.5;

                        if distance < clumps(i).neighbor_distance(neighbor_index)
                            clumps(i).neighbor_distance(neighbor_index) = distance;
                        end
                    end
                end
            end
        end

        % sort the distances
        [clumps(i).neighbor_distance sort_indices] = sort(clumps(i).neighbor_distance);
        clumps(i).neighbor = clumps(i).neighbor(sort_indices);

    end

end
```

count_clumps_continuous.m

```
% a script to count clumps in a continuous-space plot
% This script takes as inputs:
% x = vector of x-coordinates
% y = vector of y-coordinates
% clump_radius = maximum distance between points for them to be considered part of the same clump
function [clump_size frequency clumps perc_LR] = count_clumps_continuous(x, y, clump_radius, min_x,
max_x, plot_clumps)

clump_index = 0;

perc_LR = 0;
if length(x) > 11000
   pre_calc_distance = 0;
else
   pre_calc_distance = 1;

end

if pre_calc_distance
   % calculate the distance between all points
   distance = zeros(length(x), length(x));
   for first_point = 1:length(x)
      for second_point = 1:length(x)

         distance(first_point, second_point) = ...
            ((x(first_point) - x(second_point))^2+(y(first_point) - y(second_point))^2)^0.5;

      end

   end
end

% state: 1 = in main list, 2 = in temp list, 3 = already processed
% all points start out in the main list
state = ones(length(x),1);

% remove the points from the main list one by one
main_list = find(state == 1);
while ~isempty(main_list)

   % start the temporary list with this point
   state(main_list(1)) = 2;
   clump_index = clump_index + 1;
   clump_sizes(clump_index) = 0;
   point_index = 0;

   % search around the points in the temporary list one by one
   temp_list = find(state == 2);
   touches_L = 0;
   touches_R = 0;
   while ~isempty(temp_list)

    % detect whether the point is within clump_radius of either the left or right edge
    if x(temp_list(1)) < min_x + clump_radius;
        touches_L = 1;
    end
```

```
        if x(temp_list(1)) > max_x - clump_radius;
            touches_R = 1;
        end

            point_index = point_index + 1;

            clump_sizes(clump_index) = clump_sizes(clump_index) + 1;

            % mark this point as processed
            state(temp_list(1)) = 3;

            clumps(clump_index).x(point_index) = x(temp_list(1));
            clumps(clump_index).y(point_index) = y(temp_list(1));

            if pre_calc_distance
                % find and mark all the points within clump_radius of this point that haven't already been processed
                state(find(distance(temp_list(1), :)' < clump_radius & state == 1)) = 2;
            else
                distance_vector = zeros(length(x), 1);
                for i = 1:length(x)
                    distance_vector(i) = ((x(temp_list(1)) - x(i))^2+(y(temp_list(1)) - y(i))^2)^0.5;
                end
                state(distance_vector < clump_radius & state == 1) = 2;
            end

            temp_list = find(state == 2);

        end

        if touches_L && touches_R
         perc_LR = 1;
        end

        clumps(clump_index).clump_size = clump_sizes(clump_index);

        main_list = find(state == 1);

    end

    % count the number of clumps of each size
    clump_size = sort(unique(clump_sizes));
    frequency = zeros(length(clump_size), 1);
    for i = 1:length(clump_size)
        frequency(i) = length(find(clump_sizes == clump_size(i)));
    end

    if plot_clumps

        % define a larger ColorOrder for plotting clumps
        num_colors = 100;
        colors = zeros(num_colors, 3);
        for i = 1:num_colors
            colors(i, :) = [rand() rand() rand()];
        end

        figure();
        set(gcf,'DefaultAxesColorOrder',colors);
        plot(x, y, '.', 'MarkerSize', 20);
        set(gcf, 'Position', [10   116   795   568]);
        hold all;
        for i = 1:length(clumps)
            pause(0.05);
            plot(clumps(i).x, clumps(i).y, '.', 'MarkerSize', 20);
            text(clumps(i).x, clumps(i).y, num2str(i), 'FontSize', 8);
        end

    end
```

synth_clump_dist.m

```
% a function to create a synthetic clump distribution
function [out_mat clump_size frequency succeeded] = synth_clump_dist(num_points, alpha, x_cells,
y_cells)

enable_disp = 1;

% minimum power law x (see Clauset et al. 2007)
x_min = 1;

max_tries = 1000;
max_fails = 1000;

% assume that the function succeeds unless it fails too many times
succeeded = 1;

% repeatedly choose clumps until the number of points = num_points
temp_num_points = 0;
index = 0;
while temp_num_points < num_points

    index = index + 1;
    r = rand();
    clumps(index) = round((x_min - 0.5)*(1-r)^(-1/(alpha - 1))+0.5);

    temp_num_points = temp_num_points + clumps(index);

end

% count the number of clumps of each size
clump_size = sort(unique(clumps));
frequency = zeros(length(clump_size), 1);
for i = 1:length(clump_size)

    frequency(i) = length(find(clumps == clump_size(i)));

end

out_mat = zeros(y_cells, x_cells);
% place the clumps
i = 0;
num_failed = 0;
while i < length(clumps)

    i = i + 1;

    if mod(i, 100) == 0 && enable_disp
        disp(['Placing clump # ' num2str(i)]);
    end

    failed = 0;

    temp_out_mat = zeros(y_cells, x_cells);

    tried = 0;
    % choose the first point of the clump at random
    spot_free = 0;
```

```
while ~spot_free

    x = randi(x_cells);
    y = randi(y_cells);

    spot_free = 1;
    for x_offset = -1:1
        for y_offset = -1:1

            test_x = min(x_cells, max(1, x+x_offset));
            test_y = min(y_cells, max(1, y+y_offset));

            if out_mat(test_y, test_x) ~= 0
                spot_free = 0;
            end

        end

    end

    tried = tried + 1;
    if tried > max_tries
        failed = 1;
        break;
    end
end

temp_out_mat(y, x) = 1;

% place the other points at random
placed = 1;
while placed < clumps(i)

    last_x = x;
    last_y = y;

    x = min(x_cells, max(1, last_x + (randi(3) - 2)));
    y = min(y_cells, max(1, last_y + (randi(3) - 2)));

    spot_free = 1;
    for x_offset = -1:1
        for y_offset = -1:1

            test_x = min(x_cells, max(1, x+x_offset));
            test_y = min(y_cells, max(1, y+y_offset));

            if out_mat(test_y, test_x) ~= 0
                spot_free = 0;
            end

        end

    end

    if spot_free
        if temp_out_mat(y, x) == 0
            temp_out_mat(y,x) = 1;
            placed = placed + 1;
            end
        else
            x = last_x;
            y = last_y;
        end

        tried = tried + 1;
        if tried > max_tries
            failed = 1;
            break;
        end

    end

    if failed
        num_failed = num_failed + 1;
        if enable_disp
            disp(['Failed placing clump # ' num2str(i)...
                ', x = ' num2str(x) ', y = ' num2str(y) ', clump size = ' num2str(clumps(i))...
                ', num_failed = ' num2str(num_failed)]);
        end
        i = i-1;
    else
        out_mat = out_mat + temp_out_mat;
    end

    if num_failed > max_fails

        succeeded = 0;
        break;

    end

end
```

disperse_clumps.m

```
% A script to take an existing clump_mat and randomly disperse the clusters
function [out_mat success] = disperse_clumps(in_mat)

% Moore neighborhood
clump_radius = 1.5;
plot_clumps = 0;

[rows cols] = size(in_mat);

min_x = 1;
max_x = cols;
min_y = 1;
max_y = rows;

[x y] = convert_matrix_to_x_y(in_mat);

% count the clumps
[clump_size frequency clumps perc_LR] = count_clumps_continuous(x, y, clump_radius, min_x, max_x,
                                                    plot_clumps);

% scatter the clumps
mat_tries = 0;
mat_fail = 1;
while mat_fail && mat_tries < 100

    mat_tries = mat_tries + 1;

    out_mat = zeros(rows, cols);

    for i=1:length(clumps)

        clump_tries = 0;
        clump_fail = 1;
        while clump_fail && clump_tries < 100000

            clump_tries = clump_tries + 1;

            % randomly displace the clump
            range_x = max(clumps(i).x) - min(clumps(i).x);
            range_y = max(clumps(i).y) - min(clumps(i).y);

            min_clump_x = randi(max_x-range_x);
            min_clump_y = randi(max_y-range_y);

            % define the proposed new coordinates of the clump
            proposed_x = clumps(i).x + (min_clump_x - min(clumps(i).x));
            proposed_y = clumps(i).y + (min_clump_y - min(clumps(i).y));

            % create a matrix containing the proposed clump
            proposed_clump_mat = zeros(rows, cols);
            buffer_mat = zeros(rows, cols);
            for j=1:length(proposed_x)
                proposed_clump_mat(proposed_y(j), proposed_x(j)) = 1;

                % mark the clump including a buffer in the Moore neighborhood
                for k = -1:1
                for m = -1:1
                    buffer_mat(max(min(max_y, proposed_y(j)+m), min_y), ...
                        max(min(max_x, proposed_x(j)+k), min_x)) = 1;
                end
            end

        end

        % see if the proposed clump overlaps another existing clump
        if max(max(out_mat + buffer_mat))>1
            clump_fail = 1;
        else
            out_mat = out_mat + proposed_clump_mat;
            clump_fail = 0;
        end

    end

    if clump_fail
        mat_fail = 1;
    else
        mat_fail = 0;
    end

end

end

if mat_fail
    success = 0;
else
    success = 1;
end
```

Appendix C

Computer code for Chapter VII: Detection of imminent, non-catastrophic regime shifts

Software specifications

Recursive Porous Agent Simulation Toolkit (Repast) 3.0

Processing 1.2.1

Class list

LecLecMain.java	237
Site.java	254
PoissonMean.java	265
Controller.java	268
BubblePlot.java	269
Circle.java	270
GUIModel.java	272
BatchModel.java	275
ModelParameters.java	278

Class details

LecLecMain.java

```
package lecLecABM_v3;
import processing.core.*;
import java.applet.Applet;

// A host-pathogen, agent-based model of Lecanicillium lecanii epizootiology.
// This model is:
// continuous-space
// stochastic
// based on the Gillespie tau-leap algorithm
// comprised of individual epidemiological models adapted from the Hochberg reservoir model
// Doug Jackson, Winter 2011

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Vector;
import java.util.Collections;
import java.util.Scanner;

import lecLecABM_v3.Site;

import uchicago.src.sim.analysis.Plot;
import uchicago.src.sim.engine.Schedule;

public class LecLecMain extends ModelParameters
{

     // class variables
    public static Plot numSusceptibleGraph;
    public static Plot numInfectedGraph;
    public static Plot infectiousGraph;
    public static Plot latentGraph;
    public static boolean gui = false;

    public Controller testController;

     // instance variables
    public BubblePlot plot;
    public PApplet bubble;

    public double time;
     public ArrayList<Site> siteList = new ArrayList<Site>();
     public ArrayList<Circle> circleList = new ArrayList<Circle>();
     public double[][] distanceMatrix;

     public Schedule schedule;

     public int width;
     public int height;
     public double sizeX;
     public double sizeY;
     public int numSites;
     public int graphUpdatePeriod;

     // default parameter values
     public double defaultS0;
     public double defaultI0;
     public double defaultW0;
     public double defaultQ0;
     public double defaultQDecayConstant;
     public double defaultB;
     public double defaultD;
     public double defaultK;
     public double defaultBeta;
     public double defaultSigma;
     public double defaultTheta1;
     public double defaultTheta2;
     public double defaultMu;
     public double defaultEpsilon;
     public double defaultLambda;
     public double defaultNu;
     public double defaultPhi;
     public double defaultRho;
     public double defaultAlpha;
     public double defaultDelta;
     public double defaultTimeWet;
     public double defaultTimeDry;
     public double defaultTau;
     public double radiusCoef;

     // stats
     public double totalS;
     public double totalI;
     public double totalW;
     public double totalQ;
     public int infectedSites;
     public int uninfectedSites;

     // flags
     public boolean wetSeasonDynamics;

     // ////////////////////////////////////////////////////////////////////////
     // addModelSpecificParameters
     // add alias and long name for Model parameters you want to set at run time
     // the long name should be same as instance variable
     //
     // Note: the generic parameters from ModelParameters are already available.

     @Override
     public void addModelSpecificParameters()
     {
         parametersMap.put("X", "sizeX");
         parametersMap.put("Y", "sizeY");
         parametersMap.put("nSi", "numSites");
         parametersMap.put("gUP", "graphUpdatePeriod");
         parametersMap.put("dS0", "defaultS0");
         parametersMap.put("dI0", "defaultI0");
         parametersMap.put("dW0", "defaultW0");
         parametersMap.put("dQ0", "defaultQ0");
         parametersMap.put("dQd", "defaultQDecayConstant");
         parametersMap.put("dB", "defaultB");
         parametersMap.put("dD", "defaultD");
         parametersMap.put("dK", "defaultK");
         parametersMap.put("dBe", "defaultBeta");
         parametersMap.put("dSi", "defaultSigma");
         parametersMap.put("dT1", "defaultTheta1");
         parametersMap.put("dT2", "defaultTheta2");
         parametersMap.put("dMu", "defaultMu");
         parametersMap.put("dE", "defaultEpsilon");
         parametersMap.put("dLa", "defaultLambda");
         parametersMap.put("dNu", "defaultNu");
         parametersMap.put("dPh", "defaultPhi");
         parametersMap.put("dRh", "defaultRho");
         parametersMap.put("dAl", "defaultAlpha");
         parametersMap.put("dDe", "defaultDelta");
         parametersMap.put("dTw", "defaultTimeWet");
         parametersMap.put("dTd", "defaultTimeDry");
         parametersMap.put("dTa", "defaultTau");
         parametersMap.put("rC", "radiusCoef");
     }
```

```
// control what appears in the repast parameter panel
@Override
public String[] getInitParam()
{
    String[] params =
    { "sizeX","sizeY", "numSites","defaultS0",
            "defaultI0", "defaultW0", "defaultQ0", "defaultQDecayConstant",
            "defaultB", "defaultD",  "defaultK", "defaultBeta", "defaultSigma",
            "defaultTheta1", "defaultTheta2", "defaultMu", "defaultEpsilon",
            "defaultLambda", "defaultNu", "defaultPhi", "defaultRho",
            "defaultAlpha", "defaultDelta", "defaultTimeWet", "defaultTimeDry",
            "defaultTau", "radiusCoef"};
    return params;
}

// ////////////////////////////////////////////////////////////////////////
// constructor, if needed.
public LecLecMain()
{

}

// ////////////////////////////////////////////////////////////////////////
// setup
// set defaults after a run start or restart

@Override
public void setup()
{
    if (rDebug > 0)
        System.out.printf("==> setup...\n");
    schedule = null;
    System.gc();

    time = 0;

    siteList = new ArrayList<Site>();

    // size of the arena
    sizeX = 243;
    sizeY = 243;

    // number of ant nests
    numSites = 100;

    graphUpdatePeriod = 1000000000;
    radiusCoef = 0.008;

    defaultS0 = 50;
    defaultI0 = 10;
    defaultW0 = 10;
    defaultQ0 = 300;
    defaultQDecayConstant = 0.012;
    defaultB = 0.0668;
    defaultD = 0.0;
    defaultK = 1100;
    defaultBeta = 0.01;
    defaultSigma = 0.07;
    defaultTheta1 = 0.5;
    defaultTheta2 = 0.05;
    defaultLambda = 0.05;
    defaultMu = 0.1;
    defaultEpsilon = 0.000005;
    defaultNu = 0.01;
    defaultPhi = 0;
    defaultRho = defaultQDecayConstant;
    defaultAlpha = 0.1;
    defaultDelta = 0.3;
    defaultTimeWet = 183;
    defaultTimeDry = 365-defaultTimeWet;
    defaultTau = 1;

    wetSeasonDynamics=true;

    super.setup(); // THIS SHOULD BE CALLED after setting defaults in
    // setup().
    schedule = new Schedule(1); // create AFTER calling super.setup()

    if (rDebug > 0)
        System.out.printf("\n<=== setup() done.\n");

}

// ////////////////////////////////////////////////////////////////////
// buildModel
// We build the "conceptual" parts of the model.
// (vs the display parts, and the schedule)
//
// Create a 2D world, tell the organisms about it.
// Create organisms and add them to the lists.

public void buildModel()
{
    if (rDebug > 0)
        System.out.printf("==> buildModel...\n");

    // CALL FIRST -- defined in super class -- it starts RNG, etc
    buildModelStart();

    // tell the hosts and pathogens about "this"
    Site.setModel(this);

    defaultTimeDry = 365-defaultTimeWet;

    // width and height of the bubble plot
    width = (int) Math.ceil(sizeX);
    height = (int) Math.ceil(sizeY+22);

    // Instantiate Applet object if we're in GUI mode
    if(!this.modelType.equals("BatchModel"))
    {
        //Applet p55 = new EmbeddedP55(w, h);
        bubble = new BubblePlot(circleList, width, height);

        testController = new Controller(bubble, width, height);
        testController.setVisible(true);
    }


    // create and scatter the sites
    //scatterRandomSites();
    readSites();

    // enable drawing
    if(!this.modelType.equals("BatchModel"))
    {
        BubblePlot.lockDraw = false;
    }

    // generate the distance matrix
    distanceMatrix = new double[numSites][numSites];

    calcDistances();

    // some post-load finishing touches
    startReportFile();

    // for the initial state, calculate these numbers, store in instance
    // variables
    // record some stats every step
    calcStatistics();

    // calls to process parameter changes and write the
    // initial state to the report file.
    // NB -> you might remove/add more agentChange processing
    applyAnyStoredChanges();
    stepReport();
    getReportFile().flush();
    getPlaintextReportFile().flush();

    if (rDebug > 0)
        System.out.printf("<== buildModel done.\n");
}

// Create a new Site and put it at x, y
public Site createNewSite(double x, double y)
{
    // Create the bubbles for the different system variables. The order that these are added to the
    // list determines which bubble is displayed on top, i.e., the plot order.
    Circle circleQ = new Circle(bubble, (float) x, (float) y, (float) (radiusCoef*defaultQ0));
    circleQ.setRGBAlpha(139, 69, 19, 160);
    circleList.add(circleQ);

    Circle circleW = new Circle(bubble, (float) x, (float) y, (float) (radiusCoef*defaultW0));
    circleW.setRGBAlpha(255, 0, 0, 160);
    circleList.add(circleW);

    Circle circleS = new Circle(bubble, (float) x, (float) y, (float) (radiusCoef*(defaultS0+defaultI0)));
    circleS.setRGBAlpha(0, 255, 0, 160);
    circleList.add(circleS);

    Circle circleI = new Circle(bubble, (float) x, (float) y, (float) (radiusCoef*defaultI0));
    circleI.setRGBAlpha(255, 255, 255, 160);
    circleList.add(circleI);
    Site aSite = new Site(x, y, circleS, circleI, circleW, circleQ);
    siteList.add(aSite);
    return aSite;

}

// Add random sites
public void scatterRandomSites()
{

    double randomX, randomY;

    if (rDebug > 0)
        System.out.printf("==> scattering sites...\n");

    for (int i = 0; i<numSites; i++)
    {
        // let's find a random place to put a nest
        randomX = getUniformDoubleFromTo(0, sizeX);
        randomY = getUniformDoubleFromTo(0, sizeY);
        createNewSite(randomX, randomY);
    }

    if (rDebug > 0)
        System.out.printf("==> ...done scattering sites\n");
}

// read sites from a file
public void readSites()
{
    File file = new File("/Users/djackson/Documents/Graduate_school/L_lecanii_modeling/sites.csv");
    try
    {
        Scanner scanner = new Scanner(file);
        while(scanner.hasNextLine())
        {
            String line = scanner.nextLine();
            String[] coords = new String[2];
            coords = line.split(",");
            createNewSite(Double.parseDouble(coords[0]),Double.parseDouble(coords[1]));
        }
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
// ////////////////////////////////////////////////////////////////////////
// step
// The top of the "conceptual" model's main dynamics
public void step()
{
    double tau;

    if (rDebug > 0)
        System.out.printf("==> CML step %.0f:\n", getTickCount());
    // calculate the time increment, tau
    tau = getTau();

    // increment time
    time = time + tau;

    // loop through all of the sites to execute their local dynamics
    if(wetSeasonDynamics)
    {
        // run the wet season dynamics at each site
        // They will run until their individual times are >= time.
        for (Site aSite : siteList)
        {
            aSite.dynamicsWet();
        }
    }
    else
    {
        // run the dry season dynamics at each site
        for (Site aSite : siteList)
        {
            aSite.dynamicsDry();
        }
    }

    // calculate statistics
    calcStatistics();

    // call method to update graphs
    updateGraphs();

    if (rDebug > 0)
    {
        System.out.printf("<== main step done.\n");
    }

}

// ///////////////////////////////////////////////////////////////////////
// stepReport
// each step write out:
// Note: update the writeHeaderCommentsToReportFile() to print
// lines of text describing the data written to the report file.

public void stepReport()
{

    // set up a string with the values to write
    String s = String.format( "%5.0f", getTickCount() );
    s += String.format("  %10.3f", time);
    s += String.format("  %10.3f", totalS);
    s += String.format("  %10.3f", totalI);
    s += String.format("  %10.3f", totalW);
    s += String.format("  %10.3f", totalQ);
    s += String.format("  %10d", infectedSites);
    s += String.format("  %10d", uninfectedSites);

    // write it to the xml and plain text report files
    writeLineToReportFile("<stepreport>" + s + "</stepreport>");
    writeLineToPlaintextReportFile(s);

    // flush the buffers so the data is not lost in a "crash"
    getReportFile().flush();
    getPlaintextReportFile().flush();
}

// ////////////////////////////////////////////////////////////////////////////
// writeHeaderCommentsToReportFile
// customize to match what you are writing to the report files in
// stepReport.

@Override
public void writeHeaderCommentsToReportFile()
{
    writeLineToReportFile("<comment>");
    writeLineToReportFile("      ");
    writeLineToReportFile("  tick     time     totalS     totalI     totalW     totalQ  infectedSites
       uninfectedSites");
    writeLineToReportFile("</comment>");

    writeLineToPlaintextReportFile("#      ");
    writeLineToPlaintextReportFile("#  tick     time     totalS     totalI     totalW     totalQ  infectedSites
       uninfectedSites");
}

// ////////////////////////////////////////////////////////////////////////////
// printProjectHelp
// this could be filled in with some help to get from running with -help
// parameter

@Override
public void printProjectHelp()
{
    // print project help
    System.out.printf("\n%s -- \n", getName());

    System.out.printf("\n **** Add more info here!! **** \n");

    System.out.printf("\nactivationOrder           value\n");
    System.out.printf("\nfixed                     0\n");
    System.out.printf("\nrandomWithReplacement       1\n");
    System.out.printf("\nrandomWithoutReplacement    2\n");

    System.out.printf("\n");

    printParametersMap();

    System.exit(0);

}

public void updateGraphs()
{

    //check one of the graphs to see if we are in GUI mode
    if (numSusceptibleGraph != null )
```

```
        {
            numSusceptibleGraph.plotPoint(time, totalS, 1);
            numInfectedGraph.plotPoint(time, totalI, 1);
            infectiousGraph.plotPoint(time, totalW, 1);
            latentGraph.plotPoint(time, totalQ, 1);

        }

    }

    public void calcStatistics()
    {
        totalS = 0;
        totalI = 0;
        totalW = 0;
        totalQ = 0;
        infectedSites = 0;
        uninfectedSites = 0;

        for(Site aSite : siteList)
        {
            totalS = totalS + aSite.getS();
            totalI = totalI + aSite.getI();
            totalW = totalW + aSite.getW();
            totalQ = totalQ + aSite.getQ();
            if(aSite.getI()>0)
            {
                infectedSites ++;
            }
            else
            {
                uninfectedSites ++;
            }
        }
    }

    @Override
    public Schedule getSchedule()
    {
        return schedule;
    }

    @Override
    public String getName()
    {
        return "HostPathogen";
    }

    // setters and getters
    // notes:
    // - we use the schedule != null to indicated model has been initialized
    // - some things can't be changed after model initialization
    // (which things just depends on how the model is implemented)
    // - if we set something after model initialization,
    // we need to write an change entry to the report file.
    // - some things need to send messages to update class variables.
    //
    // NOTE: if you want changes a user makes to parameter like numBugs
```

```
// to be used after a restart (vs going back to defaults),
// you probably have to change setup() to not reinitialize IVs.

public static void setNumSusceptibleGraph (Plot graph) {numSusceptibleGraph = graph; };
public static void setNumInfectedGraph (Plot graph) {numInfectedGraph = graph; };
public static void setInfectiousGraph (Plot graph) {infectiousGraph = graph; };
public static void setLatentGraph (Plot graph) {latentGraph = graph; };

public static void setGUI(boolean b)
{
    gui = b;
}

public double getSizeX()
{
    return sizeX;
}

public void setSizeX(double sizeX)
{
    this.sizeX = sizeX;
}

public double getSizeY()
{
    return sizeY;
}

public void setSizeY(double sizeY)
{
    this.sizeY = sizeY;
}

public void setDistanceMatrix(int site1, int site2, double distance)
{
    this.distanceMatrix[site1][site2] = distance;
}

public double getDistanceMatrix(int site1, int site2)
{
    return distanceMatrix[site1][site2];
}

public int getNumSites()
{
    return numSites;
}

public void setNumSites(int numSites)
{
    this.numSites = numSites;
}

////////////////////////////////
public int getGraphUpdatePeriod()
{
    return graphUpdatePeriod;
}
```

```
public void setGraphUpdatePeriod(int gUP)
{
    graphUpdatePeriod = gUP;
}

///////////////////////////////
public double getDefaultS0()
{
    return defaultS0;
}
public void setDefaultS0(double dS0)
{
    defaultS0 = dS0;
}

///////////////////////////////
public double getDefaultI0()
{
    return defaultI0;
}
public void setDefaultI0(double dI0)
{
    defaultS0 = dI0;
}

///////////////////////////////
public double getDefaultW0()
{
    return defaultW0;
}
public void setDefaultW0(double dW0)
{
    defaultW0 = dW0;
}

///////////////////////////////
public double getDefaultQ0()
{
    return defaultQ0;
}
public void setDefaultQ0(double dQ0)
{
    defaultQ0 = dQ0;
}

///////////////////////////////
public double getDefaultQDecayConstant()
{
    return defaultQDecayConstant;
}
public void setDefaultQDecayConstant(double dQd)
{
    defaultQDecayConstant = dQd;
}

///////////////////////////////
public double getDefaultB()
{
    return defaultB;
}
public void setDefaultB(double dB)
{
    defaultB = dB;
}

///////////////////////////////
public double getDefaultD()
{
    return defaultD;
}
public void setDefaultD(double dD)
{
    defaultD = dD;
}

///////////////////////////////
public double getDefaultK()
{
    return defaultK;
}
public void setDefaultK(double dK)
{
    defaultK = dK;
}

///////////////////////////////
public double getDefaultBeta()
{
    return defaultBeta;
}
public void setDefaultBeta(double dBe)
{
    defaultBeta = dBe;
}

///////////////////////////////
public double getDefaultSigma()
{
    return defaultSigma;
}
public void setDefaultSigma(double dSi)
{
    defaultSigma = dSi;
}

///////////////////////////////
public double getDefaultTheta1()
{
    return defaultTheta1;
}
public void setDefaultTheta1(double dT1)
{
    defaultTheta1 = dT1;
}

///////////////////////////////
```

```
public double getDefaultTheta2()
{
    return defaultTheta2;
}
public void setDefaultTheta2(double dT2)
{
    defaultTheta2 = dT2;
}

////////////////////////////////
public double getDefaultMu()
{
    return defaultMu;
}
public void setDefaultMu(double dMu)
{
    defaultMu = dMu;
}

////////////////////////////////
public double getDefaultEpsilon()
{
    return defaultEpsilon;
}
public void setDefaultEpsilon(double dE)
{
    defaultEpsilon = dE;
}

////////////////////////////////
public double getDefaultLambda()
{
    return defaultLambda;
}
public void setDefaultLambda(double dLa)
{
    defaultLambda = dLa;
}

////////////////////////////////
public double getDefaultNu()
{
    return defaultNu;
}
public void setDefaultNu(double dNu)
{
    defaultNu = dNu;
}

////////////////////////////////
public double getDefaultPhi()
{
    return defaultPhi;
}
public void setDefaultPhi(double dPh)
{
    defaultPhi = dPh;
}
```

```
//////////////////////////////
public double getDefaultRho()
{
    return defaultRho;
}
public void setDefaultRho(double dRh)
{
    defaultRho = dRh;
}

public ArrayList<Site> getSiteList()
{
    return siteList;
}
//////////////////////////////
public double getDefaultAlpha()
{
    return defaultAlpha;
}
public void setDefaultAlpha(double dAl)
{
    defaultAlpha = dAl;
}

//////////////////////////////
public double getDefaultDelta()
{
    return defaultDelta;
}
public void setDefaultDelta(double dDe)
{
    defaultDelta = dDe;
}

//////////////////////////////
public double getDefaultTimeWet()
{
    return defaultTimeWet;
}
public void setDefaultTimeWet(double dTw)
{
    defaultTimeWet = dTw;
}

//////////////////////////////
public double getDefaultTimeDry()
{
    return defaultTimeDry;
}
public void setDefaultTimeDry(double dTd)
{
    defaultTimeDry = dTd;
}

//////////////////////////////
public double getDefaultTau()
{
    return defaultTau;
}
public void setDefaultTau(double dTa)
{
    defaultTau = dTa;
}

//////////////////////////////
public double getRadiusCoef()
{
    return radiusCoef;
}
public void setRadiusCoef(double rC)
{
    radiusCoef = rC;
}

public double getTau()
{
    // if it's the wet season, time will advance by tau;
    // otherwise, time will advance by the length of the dry season
    if(wetSeason())
    {
        return defaultTau;
    }
    else
    {
        return defaultTimeDry;
    }

}

public double getTime()
{
    return time;
}

public boolean wetSeason()
{
    if(time%(defaultTimeWet+defaultTimeDry)<defaultTimeWet)
    {
        wetSeasonDynamics=true;
        return true;
    }
    else
    {
        wetSeasonDynamics=false;
        return false;
    }
}

private void calcDistances()
{
    int i = 0;
    int j = 0;

    for(Site aSite: siteList)
    {
            j = 0;

            for(Site bSite: siteList)
            {
                distanceMatrix[i][j] = Math.sqrt(
                        Math.pow((aSite.getX()-bSite.getX()), 2)+Math.pow((aSite.getY()-bSite.getY()),2));
                j++;
            }
            i++;
        }
    }

    public double getDistance(Site siteA, Site siteB)
    {
        return distanceMatrix[siteA.getID()][siteB.getID()];
    }

    // ////////////////////////////////////////////////////////////////////////////
    // processEndOfRun
    // called once, at end of run.
    // writes some final info, closes report files, etc.
    public void processEndOfRun()
    {
        if (rDebug > 0)
            System.out.printf("\n\n===== processEndOfRun =====\n\n");
        applyAnyStoredChanges();
        endReportFile();
        this.fireStopSim();
    }

    public void closeSiteReports()
    {
        for(Site aSite : siteList)
        {
            aSite.closeOutputFile();
        }
    }
}
```

Site.java

```
// Simulate Hochberg reservoir disease dynamics in a site
// Doug Jackson
// Fall 2011

package lecLecABM_v3;

import java.awt.BasicStroke;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.Boolean;
import java.util.Vector;

import uchicago.src.sim.gui.*;
import java.awt.Color;

import java.util.Random;
import org.uncommons.maths.random.MersenneTwisterRNG;
import org.uncommons.maths.random.PoissonGenerator;

import lecLecABM_v3.GUIModel;

public class Site
{
    // class variables
    public static int nextID = 0; // to give each an ID
    public static LecLecMain model; // the model "in charge"
    public static GUIModel guiModel = null; // the gui model "in charge"
    static final int criticalThr = 10;
    static final double eps = 0.03;
    static final int g = 2;
    static final int directThr = 10;
    static final int numDirect = 100;
    static final boolean doChangeParameters = false;

    // instance variables
    public int ID;
    public double x, y;
    public double S;
    public double I;
    public double W;
    public double Q;
    public double QDecayConstant;
    public double b;
    public double d;
    public double K;
    public double beta;
    public double sigma;
    public double theta1;
    public double theta2;
    public double mu;
    public double epsilon;
    public double lambda;
    public double nu;
    public double phi;
    public double rho;
public double alpha;
public double delta;
public double radiusCoef;
public Circle circleS;
public Circle circleI;
public Circle circleQ;
public Circle circleW;

// Each site has its own unique time
public double time;

// variables for generating Poisson random variables using
// org.uncommons.maths.random
public Random rng;
public PoissonGenerator gen;
public PoissonMean meanSeed;

// variables for the OTL
public double[][] nuMat;
public double[] a;
public boolean[] critical;
public double tau;
public double tau1;
public double tau2;

// BufferedWriter for output file
public BufferedWriter out;

// changeParameters schedule (currently changes rho and QDecayConstant)
// To change a different parameter, modify the methods changeParameters, writeState, and
// setupOutputFile

// first nu hysteresis sweep
public double[] changeParametersTimes =  {0, 3650, 7300, 10950, 14600, 18250, 21900, 25550, 29200,
        32850, 36500, 40150, 43800, 47450, 51100, 54750, 58400, 62050, 65700, 69350, 73000, 76650,
        80300, 83950, 87600, 91250, 94900, 98550, 102200, 105850, 109500, 113150, 116800, 120450,
        124100, 127750, 131400, 135050, 138700, 142350, 146000, 149650, 153300, 156950, 160600,
        164250, 167900, 171550, 175200, 178850, 182500, 186150, 189800, 193450, 197100, 200750,
        204400, 208050, 211700, 215350, 219000, 222650};
public double[] changeParametersValues =  {0.01, 0.0095, 0.009, 0.0085, 0.008, 0.0075, 0.007, 0.0065,
         0.006, 0.0055, 0.005, 0.0045, 0.004, 0.0035, 0.003, 0.0025, 0.002, 0.0015, 0.001, 0.0005,
        0.00025, 0.000125, 0.0000625, 0.00003125, 0.000015625, 7.8125E-06, 3.9062E-06, 1.9531E-06,
        9.766E-07, 4.883E-07, 2.441E-07, 2.441E-07, 4.883E-07, 9.766E-07, 1.9531E-06, 3.9062E-06,
        7.8125E-06, 0.000015625, 0.00003125, 0.0000625, 0.000125, 0.00025, 0.0005, 0.001, 0.0015,
        0.002, 0.0025, 0.003, 0.0035, 0.004, 0.0045, 0.005, 0.0055, 0.006, 0.0065, 0.007, 0.0075, 0.008,
        0.0085, 0.009, 0.0095, 0.01};

// the Host constructors
public Site(double X, double Y, Circle circleS, Circle circleI, Circle circleW, Circle circleQ)
{
    ID = nextID++;
    x = X;
    y = Y;
    S = model.getDefaultS0();
    I = model.getDefaultI0();
    W = model.getDefaultW0();
    Q = model.getDefaultQ0();
    QDecayConstant = model.getDefaultQDecayConstant();
    b = model.getDefaultB();
    d = model.getDefaultD();
    K = model.getDefaultK();
    beta = model.getDefaultBeta();
    sigma = model.getDefaultSigma();
    theta1 = model.getDefaultTheta1();
    theta2 = model.getDefaultTheta2();
    mu = model.getDefaultMu();
    epsilon = model.getDefaultEpsilon();
    lambda = model.getDefaultLambda();
    nu = model.getDefaultNu();
    phi = model.getDefaultPhi();
    rho = model.getDefaultRho();
    alpha = model.getDefaultAlpha();
    delta = model.getDefaultDelta();

    // circle setup
    radiusCoef = model.getRadiusCoef();
    this.circleS = circleS;
    this.circleI = circleI;
    this.circleW = circleW;
    this.circleQ = circleQ;

    // set up PoissonGenerator
    rng = new MersenneTwisterRNG();
    meanSeed = new PoissonMean(0);
    gen = new PoissonGenerator(meanSeed, rng);

    // state change matrix
    nuMat = new double[4][8];

    // propensity functions
    a = new double[8];

    // critical reaction flag vector
    critical = new boolean[8];

    // set up OTL (optimized tau-leap) method
    // This is based on:
    // Efficient step size selection for the tau-leaping simulation method
    // Cao, Gillespie, Petzold
    setupOTL();

    // initialize the time
    time = 0;

    // Set up the output file
    setupOutputFile();

}

public void dynamicsWet()
{
    boolean calcTaus;
    double oldS;
    double oldI;
    double oldW;
    double oldQ;
```

```
// check to see if there's anything to do at this site
if(S==0 & I==0 & W==0 & Q==0)
{
    time = model.getTime();
}
// keep running the dynamics until the local time catches
// up to the model time
while(time < model.getTime())
{
    if(doChangeParameters)
    {
        // change the parameters based on a predifined schedule
        changeParameters();
    }

    calcTaus = true;
    oldS = S;
    oldI = I;
    oldW = W;
    oldQ = Q;

    // OTL method
    calcCritical();
    calcTau1();

    // keep trying until you get non-negative results
    while(calcTaus)
    {
        // choose which method to execute based on the values of tau1 and tau2
        if(tau1 < directThr/calcSumA())
        {
            directMethod();
            calcTaus = false;
        }
        else
        {
            calcTau2();
            if(tau1<tau2)
            {
                tau = tau1;
                nonCritETL();
            }
            else
            {
                tau = tau2;
                oneCrit();
                nonCritETL();
            }

            // Check to see if any of the species are negative.
            // If so, return to step 3
            if(S<0 | I<0 | W<0 | Q<0)
            {
                // cut tau1 in half and try again
                tau1 = tau1/2;

                // revert to the previous state
```

```
                    S = oldS;
                    I = oldI;
                    W = oldW;
                    Q = oldQ;

                    calcTaus = true;
                }
                else
                {

                    // update time
                    time = time + tau;

                    calcTaus = false;
                }

            }

            // get rid of fractional remainders
            if(S<1) S=0;
            if(I<1) I=0;
            if(W<1) W=0;
            if(Q<1) Q=0;
            if(S==0 & I==0 & W==0 & Q==0)
            {
                time = model.getTime();
            }

        }
    }

    writeState();
    plotResults();

}

public void dynamicsDry()
{
    if(doChangeParameters)
    {
        // change the parameters based on a predifined schedule
        changeParameters();
    }

    S = model.getDefaultS0();
    Q = (theta2*I+Q)*Math.exp(-QDecayConstant*model.getDefaultTimeDry());
    I = 0;
    W = 0;

    // update individual time to match overall time
    time = model.getTime();

    writeState();
    plotResults();
}

// ///////////////////////////////////////////////////////////////////////////
// note these are class methods, to set class variables
public static void setModel(LecLecMain m)
{
    model = m;
}

public static void setGUIModel(GUIModel m)
{
    guiModel = m;
}

// setters and getters
public void setID(int i)
{
    ID = i;
}
public int getID()
{
    return ID;
}

public double getX()
{
    return x;
}
public void setX(double x)
{
    this.x = x;
}

public double getY()
{
    return y;
}
public void setY(double y)
{
    this.y = y;
}

public double getS()
{
    return S;
}

public double getI()
{
    return I;
}

public double getW()
{
    return W;
}

public double getQ()
{
    return Q;
}
```

```
// methods
public void setupOTL()
{
    double[][] newNuMat =
    {
            {1, -1, -1, 0, 0, 0, 0, 0},
            {0, 0, 1, -1, 0, 0, 0, 0},
            {0, 0, 0, theta1, -1, -1, 1, 0},
            {0, 0, 0, theta2, 0, 1, -1, -1}
    };
    // set up state change matrix
    // rows = S, I, W, Q
    // col = births, deaths, infections, removedInfecteds, removedInfectious, infectiousToLatent,
    //        latentToInfectious, removedLatent
    nuMat = newNuMat;

}

// Calculate the propensity vector
public void calcA()
{
    double birthRate;
    double deathRate;
    double selfInfectionRate;
    double externalInfectionRate;
    double infectionRate;
    double removedInfectedsRate;
    double removedInfectiousRate;
    double infectiousToLatentRate;
    double latentToInfectiousRate;
    double removedLatentRate;
    double distance;

    // set up the propensity equations
    // births and deaths from the logistic growth component
    // See GillespieSSA: Implementing the Stochastic Simulation Algorithm in R
    birthRate = b*S;
    deathRate = S*(d+(b-d)*S/K);

    // infections from local infectious spores
    selfInfectionRate = beta*S*W;

    // infections from infectious spores in other sites
    externalInfectionRate = 0;
    for (Site aSite : model.getSiteList())
    {
        if(aSite.getID() != this.ID)
        {
            distance = model.getDistance(this, aSite);
            externalInfectionRate = externalInfectionRate +
                alpha*mu*aSite.getW()/Math.exp(delta*distance);
        }
    }
    externalInfectionRate = (externalInfectionRate+epsilon)*beta*S;

    // removal of infected scales
    removedInfectedsRate = sigma*I;
    // removal of infectious spores
    removedInfectiousRate = mu*W;

    // translocation of infectious spores to latent class
    infectiousToLatentRate = lambda*W;

    // translocation of latent spores to infectious class
    latentToInfectiousRate = nu*Q;

    // removal of latent spores
    removedLatentRate = rho*Q;

    // col = births, deaths, infections, removedInfecteds, removedInfectious, infectiousToLatent,
    //        latentToInfectious, removedLatent
    double[] newA =
    {
            birthRate,
            deathRate,
            selfInfectionRate+externalInfectionRate,
            removedInfectedsRate,
            removedInfectiousRate,
            infectiousToLatentRate,
            latentToInfectiousRate,
            removedLatentRate
    };
    a = newA;

}

public void calcCritical()
{
    boolean[] newCritical =
    {
            false,
            S<criticalThr,
            S<criticalThr,
            I<criticalThr,
            W<criticalThr,
            W<criticalThr,
            Q<criticalThr,
            Q<criticalThr
    };
    critical = newCritical;
}

// Calculate the first candidate tau
public void calcTau1()
{
    double mu;
    double sigmaSquared;
    double leftTerm;
    double rightTerm;

    // recalculate the propensity function
    calcA();

    if(!critical[0] | !critical[1] | !critical[2] | !critical[3] | !critical[4] |
            !critical[5] | !critical[6] | !critical[7])
    {
        mu = 0;
        sigmaSquared = 0;
        for(int j=0; j<8; j++)
        {
            if(!critical[j])
            {
                for(int i=0; i<4; i++)
                {
                    // equation 32a
                    mu = mu + nuMat[i][j]*a[j];
                    // equation 32b
                    sigmaSquared = sigmaSquared + Math.pow(nuMat[i][j],2)*a[j];
                }
            }
        }

        // equation 33
        leftTerm = Math.max(eps*S/g,
                Math.max(eps*I/g,
                        Math.max(eps*W/g,
                                Math.max(eps*Q/g, 1))))/Math.abs(mu);
        rightTerm = Math.pow(Math.max(eps*S/g,
                Math.max(eps*I/g,
                        Math.max(eps*W/g,
                                Math.max(eps*Q/g, 1)))),2)/sigmaSquared;
        tau1 = Math.min(leftTerm, rightTerm);

    }
    else
    {
        tau1 = Double.POSITIVE_INFINITY;
    }

}

// Calculate the second candidate tau
public void calcTau2()
{
    double sumCritA = 0;

    // recalculate the propensity function
    calcA();

    if(critical[0] | critical[1] | critical[2] | critical[3] | critical[4] |
            critical[5] | critical[6] | critical[7])
    {
        for(int i=0; i<8; i++)
        {
            if(critical[i])
            {
                sumCritA = sumCritA + a[i];

            }
        }
        tau2 = -Math.log(Math.random())/sumCritA;
    }
```

```
        else
        {
            tau2 = Double.POSITIVE_INFINITY;
        }
    }

    // Execute the non-critical reactions using the explicit tau-leap
    // method
    public void nonCritETL()
    {
        int numReactions;

        // recalculate the propensity function
        calcA();

        for(int j=0; j<8; j++)
        {
            // only fire non-critical reactions
            if(!critical[j])
            {
                // calculate the number of times the reaction occurs
                meanSeed.setMean(tau*a[j]);
                numReactions = gen.nextValue();

                S = Math.max(0, S + numReactions*nuMat[0][j]);
                I = Math.max(0, I + numReactions*nuMat[1][j]);
                W = Math.max(0, W + numReactions*nuMat[2][j]);
                Q = Math.max(0, Q + numReactions*nuMat[3][j]);
            }
        }
    }

    // Execute one of the critical reactions at random
    public void oneCrit()
    {
        double sumCritA = 0;
        double[] probCrit = new double[8];
        double cumulProb=0;
        double randNum;

        // recalculate the propensity function
        calcA();

        // Calculate the sum of the critical propensity functions
        for(int j=0; j<8; j++)
        {
            if(critical[j])
            {
                sumCritA = sumCritA + a[j];

            }
        }

        // Calculate the probability of each critical reaction occurring
        for(int j=0; j<8; j++)
        {
            if(critical[j])
            {
                probCrit[j] = a[j]/sumCritA;
            }
            else
            {
                // not a critical reaction
                probCrit[j] = 0;
            }
        }

        // Choose the next reaction randomly
        randNum = Math.random();
        for(int j=0; j<8; j++)
        {
            cumulProb += probCrit[j];
            if(randNum<cumulProb)
            {
                // execute the reaction once
                S = Math.max(0, S + nuMat[0][j]);
                I = Math.max(0, I + nuMat[1][j]);
                W = Math.max(0, W + nuMat[2][j]);
                Q = Math.max(0, Q + nuMat[3][j]);
                break;
            }
        }
    }

    // Calculate the sum of the propensity vector
    public double calcSumA()
    {
        double sumA = 0;

        for(int i=0; i< 8; i++)
        {
            sumA = sumA + a[i];
        }

        return sumA;

    }
    // Execute the direct method
    public void directMethod()
    {
        double randNum;
        double sumA;
        double tempTime;
        double[] cumulProb = new double[8];

        for(int i=0; i<numDirect; i++)
        {
            // recalculate the propensity function
            calcA();
            sumA = calcSumA();

            // Calculate the cumulative probability of each critical reaction occurring
            cumulProb[0] = a[0]/sumA;
            for(int j=1; j<8; j++)
            {
                cumulProb[j] = cumulProb[j-1]+a[j]/sumA;
            }

            // calculate the time step
            randNum = Math.random();
            tempTime = time + -Math.log(randNum)/sumA;

            // Check to see if the next event would occur in the future.
            // If so, don't do it, and abort the direct method.
            if(tempTime > model.getTime())
            {
                time = model.getTime();
                break;
            }
            else
            {
                time = tempTime;
            }

            // determine which reaction occurs
            randNum = Math.random();
            for(int j=0; j<8; j++)
            {
                if(randNum<cumulProb[j])
                {
                    // execute the reaction once
                    S = Math.max(0, S + nuMat[0][j]);
                    I = Math.max(0, I + nuMat[1][j]);
                    W = Math.max(0, W + nuMat[2][j]);
                    Q = Math.max(0, Q + nuMat[3][j]);
                    break;
                }
            }

        }

    }

    public void plotResults()
    {
        // plot results
        circleS.setRadius((float) (radiusCoef*(S+I)));
        circleI.setRadius((float) (radiusCoef*I));
        circleW.setRadius((float) (radiusCoef*W));
        circleQ.setRadius((float) (radiusCoef*Q));
    }

    public void setupOutputFile()
    {
        try
        {
            out = new BufferedWriter(new FileWriter("site_" + Integer.toString(ID) + ".csv"));
            out.write("x, y");
            out.newLine();
            out.write(Double.toString(x));
            out.write(",");
            out.write(Double.toString(y));
            out.newLine();
            out.write("time, S, I, Q, W, nu");
            out.newLine();
        } catch (IOException e)
        {

        }
    }
    public void changeParameters()
    {
        // There's definitely a more efficient way to do this...
        for(int i=0; i<changeParametersTimes.length; i++)
        {
            if(time>changeParametersTimes[i])
            {
                //QDecayConstant = changeParametersValues[i];
                //rho = changeParametersValues[i];
                nu = changeParametersValues[i];
            }
        }

    }
    public void writeState()
    {
        // write this Site's state to its own file
        try
        {
            out.write(Double.toString(model.getTime()));
            out.write(",");
            out.write(Double.toString(S));
            out.write(",");
            out.write(Double.toString(I));
            out.write(",");
            out.write(Double.toString(Q));
            out.write(",");
            out.write(Double.toString(W));
            out.write(",");
            out.write(Double.toString(nu));
            out.newLine();
        } catch (IOException e)
        {

        }
    }

    public void closeOutputFile()
    {
        try
        {
            out.close();
        } catch (IOException e)
        {
        }
    }
}
```

PoissonMean.java

```
package lecLecABM_v3;

import org.uncommons.maths.number.NumberGenerator;

public class PoissonMean implements NumberGenerator<Double>
{

    public double mean;

    public PoissonMean(double mean)
    {
        this.mean = mean;
    }

    //@Override
    public Double nextValue()
    {
        // TODO Auto-generated method stub
        return mean;
    }

    public void setMean(double mean)
    {
        this.mean = mean;
    }

}
```

Controller.java

```
package lecLecABM_v3;

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Controller extends Frame
{
    // constructor
    public Controller(Applet bubble, int width, int height)
    {

        // call to superclass needs to come first in constructor
        super("green=healthy, white=infected, red=infectious, brown=reservoir");

        // set up frame (which will hold applet)
        setSize(width, height);
        setLayout(new FlowLayout(FlowLayout.LEFT, 0, 0));

        // add Applet component to frame
        add(bubble);

        // won't allow frame to be resized
        setResizable(false);

        // allow window and application to be closed
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });

        // Next comment taken directly from PApplet class:
        /* "...ensures that the animation thread is started and that other internal variables are properly set."*/
        bubble.init();
    }

}
```

BubblePlot.java

```
package lecLecABM_v3;

import java.util.ArrayList;

import processing.core.*;

public class BubblePlot extends PApplet
{
    // the lockDraw variable is necessary to keep the draw function from trying to
    // access the circleList ArrayList while I'm adding elements to it (which
    // throws a ConcurrentModification exception)
    public static boolean lockDraw = true;

    // instance variables
    private int width;
    private int height;
    public ArrayList<Circle> circleList;

    // constructor
    public BubblePlot(ArrayList<Circle> circleList, int width, int height)
    {
        this.circleList = circleList;
        this.width = width;
        this.height = height;
    }

    public void setup()
    {
        size(width, height);

        smooth();
        noStroke();

    }

    public void draw()
    {
        background(100);
        if(!lockDraw)
        {
            for (Circle aCircle : circleList)
            {
                fill(aCircle.getFillR(), aCircle.getFillG(), aCircle.getFillB(), aCircle.getFillAlpha());
                aCircle.display();
            }
        }
    }
}
```

Circle.java

```
package lecLecABM_v3;
import processing.core.PApplet;

public class Circle
{
    float x;
    float y;
    float radius;
    public float fillR;
    public float fillG;
    public float fillB;
    public float fillAlpha;

    PApplet parent; // The parent PApplet that we will render ourselves onto

    Circle(PApplet p, float x, float y, float radius )
    {
        parent = p;

       // store the values of the parameters into the matching object
       // variables
       this.x = x;
       this.y = y;
       this.radius = radius;

       // default color
       setRGBAlpha(255, 0, 0, 160);

     }

    // Draw circle
    void display()
    {
       // draw the circle
       parent.ellipse(this.x, this.y, this.radius*2, this.radius*2);
    }

    void setRadius(float radius)
    {
        this.radius = radius;
    }

    void setRGBAlpha(float r, float g, float b, float alpha)
    {
        fillR = r;
        fillG = g;
        fillB = b;
        fillAlpha = alpha;
    }

    float getFillR()
    {
        return fillR;
    }

    float getFillG()
    {
        return fillG;
    }

    float getFillB()
    {
        return fillB;
    }

    float getFillAlpha()
    {
        return fillAlpha;
    }
}
```

GUIModel.java

```
package lecLecABM_v3;

import uchicago.src.sim.gui.DisplaySurface;
import uchicago.src.sim.gui.Object2DDisplay;
import uchicago.src.sim.engine.AbstractGUIController;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.analysis.*;

public class GUIModel extends LecLecMain
{

    // ///////////////////////////////////////////////////////////////
    // setup
    //
    // this runs automatically when the model starts
    // and when you click the reload button, to "tear down" any
    // existing display objects, and get ready to initialize
    // them at the start of the next 'run'.
    //
    @Override
    public void setup()
    {
        super.setup(); // the super class does conceptual-model setup

        AbstractGUIController.CONSOLE_ERR = false;
        AbstractGUIController.CONSOLE_OUT = false;
        AbstractGUIController.UPDATE_PROBES = true;

        // tell the Host class we are in GUI mode.
        LecLecMain.setGUI(true);

        // init, setup and turn on the modelMinipulator stuff (in custom
        // actions)
        modelManipulator.init();

        if (rDebug > 0)
            System.out.printf("<== GUIModel setup() done.\n");
    }

    // ///////////////////////////////////////////////////////////////
    // begin
    //
    // this runs when you click the "initialize" button
    // (the button with the single arrow that goes around in a circle)
    //
    @Override
    public void begin()
    {
        DMSG(1, "==> enter GUIModel-begin()");
        buildModel(); // the base model does this
        buildDisplay();
        buildSchedule();
        DMSG(1, "<== leave GUIModel-begin() done.");
    }

    // ///////////////////////////////////////////////////////////////
// buildDisplay
//
// builds the display and display related things
//
public void buildDisplay()
{
    // Graphs
    // Graphs in Repast are too slow, so I'll just graph everything in R

}

// ////////////////////////////////////////////////////////////
// buildSchedule
//
// This builds the entire schedule, i.e.,
// - the base model step
// - report step
// - display steps.

@Override
public void buildSchedule()
{

    if (rDebug > 0)
        System.out.printf("==> GUIModel buildSchedule...\n");

    // schedule the current GUIModel's step() function
    // to execute every time step starting with time step 0
    schedule.scheduleActionBeginning(0, this, "step");
    // start report at 1
    schedule.scheduleActionAtInterval(1, this, "stepReport", Schedule.LAST);

    // schedule the current GUIModel's processEndOfRun()
    // function to execute at the end of the run
    schedule.scheduleActionAtEnd(this, "processEndOfRun");
}

// //////////////////////////////////////////////////////////////////////////
// step
//
// executed each step of the model.
// Ask the super class to do its step() method,
// and then this does display related activities.
//
@Override
public void step()
{

    super.step(); // the model does whatever it does

}

// processEndOfRun
// called once, at end of run.
@Override
public void processEndOfRun()
{
    if (rDebug > 0)
```

```
            System.out.printf("\n\n===== GUIModel processEndOfRun =====\n\n");
        applyAnyStoredChanges();
        endReportFile();
        closeSiteReports();
        this.fireStopSim();
    }

    // ////////////////////////////////////////////////////////////////////
    // main entry point
    public static void main(String[] args)
    {

        uchicago.src.sim.engine.SimInit init = new uchicago.src.sim.engine.SimInit();
        GUIModel model = new GUIModel();

        // set the type of model class, this is necessary
        // so the parameters object knows whether or not
        // to do GUI related updates of panels,etc when a
        // parameter is changed
        model.setModelType("GUIModel");

        // Do this to set the Update Probes option to true in the
        // Repast Actions panel
        AbstractGUIController.UPDATE_PROBES = true;

        model.setCommandLineArgs(args);
        init.loadModel(model, null, false); // does setup()

        // this new function calls ProbeUtilities.updateProbePanels() and
        // ProbeUtilities.updateModelProbePanel()
        model.updateAllProbePanels();

    }

}
```

BatchModel.java

```
package lecLecABM_v3;

import uchicago.src.sim.engine.*;
import processing.core.*;
import java.applet.Applet;

public class BatchModel extends LecLecMain
{

    // ////////////////////////////////////////////////////////////
    // main entry point
    public static void main(String[] args)
    {

        BatchModel model = new BatchModel();

        // set the type of model class, this is necessary
        // so the parameters object knows whether or not
        // to do GUI related updates of panels, etc when a
        // parameter is changed
        model.setModelType("BatchModel");

        model.setCommandLineArgs(args);

        PlainController control = new PlainController();
        model.setController(control);
        control.setExitOnExit(true);
        control.setModel(model);
        model.addSimEventListener(control);
        if (model.getRDebug() > 0)
            System.out.printf("\n==> BatchModel main...about to startSimulation...\n");
        control.startSimulation();
    }

    // setup() -- BatchModel just does what the super class does.
    @Override
    public void setup()
    {
        super.setup();
    }

    // begin()
    // ask the super class to do its building, then build a schedule.
    @Override
    public void begin()
    {
        // set schedule to null so buildModel knows not to
        // record changes ( changes are recorded if
        // schedule != null ). in buildSchedule() the
        // schedule is allocated before the actual schedule is created.
        schedule = null;
        buildModel(); // the base Model class does this
        buildSchedule();
    }

    // ////////////////////////////////////////////////////////////
    // buildSchedule
    @Override
    public void buildSchedule()
    {

        schedule = new Schedule(1);

        // schedule the current BatchModel's step() function
        // to execute every time step starting with time step 0
        schedule.scheduleActionBeginning(0, this, "step");
        schedule.scheduleActionAtInterval(1, this, "stepReport", Schedule.LAST);

        // schedule the current BatchModel's processEndOfRun()
        // function to execute at the end of the Batch Run.
        // You need to specify the time to schedule it (instead
        // of doing scheduleActionAtEnd() or it will just run forever
        schedule.scheduleActionAt(getStopT(), this, "processEndOfRun");
    }

    // processEndOfRun
    // we need this to tell it to stop running!
    @Override
    public void processEndOfRun()
    {
        super.processEndOfRun();
        this.fireEndSim();
    }
}

// ////////////////////////////////////////////////////////////////////////////
// ////////////////////////////////////////////////////////////////////////////
// Why this class below?
//
// the reason we did that is because the repast "BatchController" had methods
// in it that started GUI stuff. this caused problems when we ssh'd into
// another machine and run a job--when we tried to disconnect, the ssh
// session would stay hung until the job was finished because the job needed
// the X11-forwarding to be open to run.

class PlainController extends BaseController
{
    private boolean exitonexit;

    public PlainController()
    {
        super();
        exitonexit = false;
    }

    public void startSimulation()
    {
        startSim();
    }

    public void stopSimulation()
    {
        stopSim();
    }
    public void exitSim()
    {
        exitSim();
    }

    public void pauseSimulation()
    {
        pauseSim();
    }

    @Override
    public boolean isBatch()
    {
        return true;
    }

    @Override
    protected void onTickCountUpdate()
    {
    }

    @Override
    public void setExitOnExit(boolean in_Exitonexit)
    {
        exitonexit = in_Exitonexit;
    }

    public void simEventPerformed(SimEvent evt)
    {
        if (evt.getId() == SimEvent.STOP_EVENT)
        {
            stopSimulation();
        } else if (evt.getId() == SimEvent.END_EVENT)
        {
            if (exitonexit)
            {
                System.exit(0);
            }
        } else if (evt.getId() == SimEvent.PAUSE_EVENT)
        {
            pauseSimulation();
        }
    }

    // function added because it is required for repast 2.2
    public long getRunCount()
    {
        return 0;
    }

    // function added because it is required for repast 2.2
    public boolean isGUI()
    {
        return false;
    }
}
```

ModelParameters.java

See *ModelParameters.java* in Appendix A. This class is identical except the package name, which reads:

package lecLecABM_v3;

Bibliography

Ahimera, N., S. Gisler, D. Morgan, and T. Michailides. 2004. Effects of single-drop impactions and natural and simulated rains on the dispersal of *Botryosphaeria dothidea* conidia. Phytopathology **94**:1189-1197.

Alados, C., A. Aich, B. Komac, and Y. Pueyo. 2007. Self-organized spatial patterns of vegetation in alpine grasslands. Ecological Modelling **201**:233-242.

Alarcón, R. and G. Carrión. 1994. Uso de *Verticillium lecanii* en cafetales como control biológico de la roya del cafeto. Fitopatología (Perú) **29**:82-85.

Alonso, D. and M. Pascual. 2006. Comment on "A keystone mutualism drives pattern in a power function". Science **313**:1739b.

Amarasekare, P. and R. Nisbet. 2001. Spatial heterogeneity, source-sink dynamics, and the local coexistence of competing species. The American Naturalist **158**:572-584.

Arnold, S. J. 1992. Constraints on phenotypic evolution. The American Naturalist **140 Suppl 1**:S85-107.

Avelino, J., L. Willocquet, and S. Savary. 2004. Effects of crop management patterns on coffee rust epidemics. Plant Pathology **53**:541-547.

Avelino, J., H. Zelaya, A. Merlo, A. Pineda, M. Ordoñez, and S. Savary. 2006. The intensity of a coffee rust epidemic is dependent on production situations. Ecological Modelling **197**:431-447.

Axelrod, R. and W. D. Hamilton. 1981. The evolution of cooperation. Science **211**:1390.

Aylor, D. E. 1990. The role of intermittent wind in the dispersal of fungal pathogens. Annual Review of Phytopathology **28**:73-92.

Badgley, C., J. Moghtader, E. Quintero, E. Zakem, M. Chappell, K. Avilés-Vázquez, A. Samulon, and I. Perfecto. 2007. Organic agriculture and the global food supply. Renewable Agriculture and Food Systems **22**:86-108.

Badgley, C. and I. Perfecto. 2007. Can organic agriculture feed the world? Renewable Agriculture and Food Systems **22**:80-86.

Bak, P. 1996. How nature works. Springer, New York.

Bale, J., J. Van Lenteren, and F. Bigler. 2008. Biological control and sustainable food production. Philosophical Transactions B **363**:761-776.

Barbosa, P. 1998. Conservation biological control. Academic Press, San Diego, California.

Bascompte, J. and R. Solé. 1998. Spatiotemporal patterns in nature. Trends in Ecology & Evolution **13**:173-174.

Bengtsson, J., J. Ahnström, and A. Weibull. 2005. The effects of organic agriculture on biodiversity and abundance: a meta-analysis. Journal of Applied Ecology **42**:261-269.

Bianchi, F., C. Booij, and T. Tscharntke. 2006. Sustainable pest regulation in agricultural landscapes: a review on landscape composition, biodiversity and natural pest control. Proceedings of the Royal Society B: Biological Sciences **273**:1715-1727.

Bird, A. E., H. Hesketh, J. V. Cross, and M. Copland. 2004. The common black ant, *Lasius niger* (Hymenoptera : Formicidae), as a vector of the entomopathogen *Lecanicillium longisporum* to rosy apple aphid, *Dysaphis plantaginea* (Homoptera : Aphididae). Biocontrol Science and Technology **14**:757-767.

Boots, M. and M. Mealor. 2007. Local interactions select for lower pathogen infectivity. Science **315**:1284-1286.

Brito, G. G., E. T. Caixeta, A. P. Gallina, E. M. Zambolim, L. Zambolim, V. Diola, and M. E. Loureiro. 2010. Inheritance of coffee leaf rust resistance and identification of AFLP markers linked to the resistance gene. Euphytica **173**:255-264.

Brosi, B., P. Armsworth, and G. Daily. 2008. Optimal design of agricultural landscapes for pollination services. Conservation Letters **1**:27-36.

Bruck, D. and L. Lewis. 2002. Rainfall and crop residue effects on soil dispersion and *Beauveria bassiana* spread to corn. Applied Soil Ecology **20**:183-190.

Bulman, C., R. Wilson, A. Holt, L. Bravo, R. Early, M. Warren, and C. Thomas. 2007. Minimum viable metapopulation size, extinction debt, and the conservation of a declining species. Ecological Applications **17**:1460-1473.

Burdon, J. J. and G. A. Chilvers. 1982. Host density as a factor in plant-disease ecology. Annual Review of Phytopathology **20**:143-166.

Butt, T. M., C. Jackson, and N. Magan. 2001. Fungi as biocontrol agents: progress, problems, and potential. CABI Publishing, Wallingford.

Canjura-Saravia, E. M., V. Sánchez-Garita, U. Krauss, and E. Somarriba. 2002. Reproducción masiva de *Verticillium* sp., hiperparásita de la roya del café, *Hemileia vastatrix*. Manejo Integrado de Plagas y Agroecología **66**:13-19.

Cao, Y., D. T. Gillespie, and L. R. Petzold. 2006. Efficient step size selection for the tau-leaping simulation method. J. Chem. Phys. **124**:044109.

Carlsson-Graner, U. and P. Thrall. 2002. The spatial distribution of plant populations, disease dynamics and evolution of resistance. Oikos **97**:97-110.

Carpenter, S. R. 2005. Eutrophication of aquatic ecosystems: bistability and soil phosphorus. Proceedings of the National Academy of Sciences of the United States of America **102**:10002-10005.

Chandler, D., J. B. Heale, and A. T. Gillespie. 1993. Germination of the entomopathogenic fungus *Verticillium lecanii* on scales of the glasshouse whitefly *Trialeurodes vaporariorum*. Biocontrol Science and Technology **3**:161-164.

Charudattan, R. and A. Dinoor. 2000. Biological control of weeds using plant pathogens: accomplishments and limitations. Crop Protection **19**:691-695.

Chen, X. 2005. Epidemiology and control of stripe rust [*Puccinia striiformis* f. sp. *tritici*] on wheat. Canadian Journal of Plant Pathology **27**:314-337.

Clauset, A., C. Shalizi, and M. Newman. 2007. Power-law distributions in empirical data. Physics:0706.1062 E-print.

Council, N. R. 1989. Alternative agriculture. The National Academies Press, Washington DC.

Cruz, L., A. Gaitan, and C. Gongora. 2006. Exploiting the genetic diversity of *Beauveria bassiana* for improving the biological control of the coffee berry borer through the use of strain mixtures. Applied Microbiology and Biotechnology **71**:918-926.

Cullen, R., K. Warner, M. Jonsson, and S. Wratten. 2008. Economics and adoption of conservation biological control. Biological Control **45**:272-280.

Currie, C. R. 2001. Prevalence and impact of a virulent parasite on a tripartite mutualism. Oecologia **128**:99-106.

Currie, C. R., G. M. Ulrich, and D. Malloch. 1999. The agricultural pathology of ant fungus gardens. Proceedings of the National Academy of Science **96**:7998-8002.

Dakos, V., E. H. Nes, R. Donangelo, H. Fort, and M. Scheffer. 2009. Spatial correlation as leading indicator of catastrophic shifts. Theoretical Ecology **3**:163-174.

Daskalov, G. M., A. N. Grishin, S. Rodionov, and V. Mihneva. 2007. Trophic cascades triggered by overfishing reveal possible mechanisms of ecosystem regime shifts. Proceedings of the National Academy of Sciences of the United States of America **104**:10518-10523.

Duffy, M. A. and L. Sivars-Becker. 2007. Rapid evolution and ecological host-parasite dynamics. Ecology Letters **10**:44-53.

Eapen, S. J., B. Beena, and K. V. Ramana. 2005. Tropical soil microflora of spice-based cropping systems as potential antagonists of root-knot nematodes. Journal of Invertebrate Pathology **88**:218-225.

Easwaramoorthy, S. and S. Jayaraj. 1977. The effect of shade on the coffee green bug, *Coccus viridis* (Green) and its entomopathogenic fungus, *Cephalosporium lecanii* Zimm. Journal of Coffee Research **7**:111-113.

Easwaramoorthy, S. and S. Jayaraj. 1978. Effectiveness of the white halo fungus, *Cephalosporium lecanii*, against field populations of coffee green bug, *Coccus viridis*. Journal of Invertebrate Pathology **32**:88-96.

Enkerli, J. and F. Widmer. 2010. Molecular ecology of fungal entomopathogens: molecular genetic tools and their applications in population and fate studies. Biocontrol **55**:17-37.

Eskes, A., M. Mendes, and C. Robbs. 1991. Laboratory and field studies on parasitism of *Hemileia vastatrix* with *Verticillium lecanii* and *V. leptobactrum*. Café Cacao Thé **35**:275-282.

Eskes, A. B. 1989. Natural enemies and biological control. Pages 162-168 *in* A. C. Kushalappa and A. B. Eskes, editors. Coffee Rust: Epidemiology, Resistance, and Management. CRC Press, Boca Raton, FL.

Feng, K., B. Liu, and Y. Tzeng. 2000. *Verticillium lecanii* spore production in solid-state and liquid-state fermentations. Bioprocess and Biosystems Engineering **23**:25-29.

Fernandez-Garcia, E. and B. Fitt. 1993. Dispersal of the entomopathogen *Hirsutella cryptosclerotium* by simulated rain. Journal of Invertebrate Pathology **61**:39-43.

Fiedler, A., D. Landis, and S. Wratten. 2008. Maximizing ecosystem services from conservation biological control: the role of habitat management. Biological Control **45**:254-271.

Fitt, B., H. McCartney, and P. Walklate. 1989. The role of rain in dispersal of pathogen inoculum. Annual Review Phytopathology **27**:241-270.

Foitzik, S., C. DeHeer, D. Hunjan, and J. Herbers. 2001. Coevolution in host-parasite systems: behavioural strategies of slave-making ants and their hosts. Proceedings of the Royal Society of London Series B-Biological Sciences **268**:1139-1146.

Fravel, D. 2005. Commercialization and implementation of biocontrol 1. Annual Review Phytopathology **43**:337-359.

Fry, W. 2008. *Phytophthora infestans*: the plant (and R gene) destroyer. Molecular Plant Pathology **9**:385-402.

Fuxa, J. R. and Y. Tanada. 1987. Epizootiology of insect disease. John Wiley and Sons, New York, NY.

Galtsoff, P. S., H. H. Brown, C. L. Smith, and F. G. W. Smith. 1939. Sponge mortality in the Bahamas. Nature **143**:807-808.

Gams, W. and R. Zare. 2001. A revision of *Verticillium* sect. Prostrata. III. Generic classification. Nova Hedwigia **72**:329-337.

Gao, L., X. Liu, M. Sun, S. Li, and J. Wang. 2009. Use of a novel two-stage cultivation method to determine the effects of environmental factors on the growth and sporulation of several biocontrol fungi. Mycoscience **50**:317-321.

Gao, L., M. Sun, X. Liu, and Y. Che. 2007. Effects of carbon concentration and carbon to nitrogen ratio on the growth and sporulation of several biocontrol fungi. Mycological Research **111**:87-92.

Geagea, L., L. Huber, I. Sache, D. Flura, H. McCartney, and B. Fitt. 2000. Influence of simulated rain on dispersal of rust spores from infected wheat seedlings. Agricultural and Forest Meteorology **101**:53-66.

Ghini, R., W. Bettiol, and E. Hamada. 2011. Diseases in tropical and plantation crops as affected by climate changes: current knowledge and perspectives. Plant Pathology **60**:122-132.

Gindin, G., I. Barash, B. Raccah, S. Singer, I. Ben-Ze'ev, and M. Klein. 1996. The potential of some entomopathogenic fungi as biocontrol agents against the onion thrips, *Thrips tabaci* and the western flower thrips, *Frankliniella occidentalis*. Folia Entomologica Hungarica **57**:37-42.

Gindin, G., N. Geschtovt, B. Raccah, and I. Barash. 2000. Pathogenicity of *Verticillium lecanii* to different developmental stages of the silverleaf whitefly, *Bemisia argentifolii*. Phytoparasitica **28**:229-239.

Goldberg, D. E. 1989. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading, Mass.

González, E., N. Bravo, and M. Carone. 1995. Caracterización de *Verticillium lecanii* (Zimm.) Viegas hiperparasitando *Hemileia vastatrix* Berk y Br y *Coccus viridis* Green. Revista de Protección Vegetal **10**:169-171.

Goreaud, F. and R. Pélissier. 1999. On explicit formulas of edge effect correction for Ripley's K-function. Journal of Vegetation Science **10**:433-438.

Gould, S. J. and R. C. Lewontin. 1979. The spandrels of San Marco and the Panglossian paradigm: a critique of the adaptationist programme. Proceedings of the Royal

Society of London. Series B, Containing papers of a Biological character. Royal Society (Great Britain) **205**:581-598.

Grenfell, B., O. Bjørnstad, and J. Kappey. 2001. Travelling waves and spatial hierarchies in measles epidemics. Nature **414**:716-723.

Gurr, G. M., S. D. Wratten, and P. Barbosa. 2000. Success in conservation biological control of arthropods. Pages 105-132 *in* G. M. Gurr and S. D. Wratten, editors. Biological control: measures of success. Kluwer Academic Publishers, London, UK.

Guttal, V. and C. Jayaprakash. 2008. Changing skewness: an early warning signal of regime shifts in ecosystems. Ecology Letters **11**:450-460.

Guttal, V. and C. Jayaprakash. 2009. Spatial variance and spatial skewness: leading indicators of regime shifts in spatial ecological systems. Theoretical Ecology **2**:3-12.

Haddad, F., L. A. Maffia, E. S. G. Mizubuti, and H. Teixeira. 2009. Biological control of coffee rust by antagonistic bacteria under field conditions in Brazil. Biological Control **49**:114-119.

Hall, R. A. 1981. The fungus *Verticillium lecanii* as a microbial insecticide against aphids and scales. Pages 483-498 *in* H. D. Burges, editor. Microbial Control of Pests and Plant Diseases. Academic Press, New York, NY.

Hamilton, W. D. 1964a. The genetical evolution of social behavior. I. Journal of Theoretical Biology **7**:1-16.

Hamilton, W. D. 1964b. The genetical evolution of social behaviour. II. Journal of Theoretical Biology **7**:17-52.

Hanski, I. 1998. Metapopulation dynamics. Nature **396**:41-49.

Hanski, I. 1999. Metapopulation ecology. Oxford University Press, Oxford, New York.

Hanski, I. and M. Gilpin. 1997. Metapopulation biology: ecology, genetics and evolution. Academic Press, London, UK.

Hanski, I. and C. Thomas. 1994. Metapopulation dynamics and conservation: a spatially explicit model applied to butterflies. Biological Conservation **68**:167-180.

Hardin, G. 1968. The tragedy of the commons. Science **162**:1243-1248.

Hasan, S. and P. Ayres. 1990. The control of weeds through fungi: principles and prospects. New Phytologist **115**:201-222.

Hassell, M., H. Comins, and R. May. 1991. Spatial structure and chaos in insect population dynamics. Nature **353**:255-258.

Hastings, A. and D. B. Wysham. 2010. Regime shifts in ecological systems can occur with no warning. Ecology Letters **13**:464-472.

Heale, J. B. 1988. The potential impact of fungal genetics and molecular biology on biological control, with particular reference to entomopathogens. Pages 211-234 *in* M. N. Burge, editor. Fungi in biological control systems. Manchester University Press, Manchester, U.K.

Hein, L. and F. Gatzweiler. 2006. The economic value of coffee (*Coffea arabica*) genetic resources. Ecological Economics **60**:176-185.

Helms, K. R. and S. Bradleigh Vinson. 2008. Plant resources and colony growth in an invasive ant: the importance of honeydew-producing hemiptera in carbohydrate transfer across trophic levels. Environmental Entomology **37**:487-493.

Helyer, N. 1993. *Verticillium lecanii* for control of aphids and thrips on cucumber. IOBC/WPRS Bulletin **16**:63-66.

Henry, M., M. Beguin, F. Requier, O. Rollin, J. F. Odoux, P. Aupinel, J. Aptel, S. Tchamitchian, and A. Decourtye. 2012. A common pesticide decreases foraging success and survival in honey bees. Science.

Hesketh, H., H. E. Roy, J. Eilenberg, J. K. Pell, and R. S. Hails. 2010. Challenges in modelling complexity of fungal entomopathogens in semi-natural populations of insects. Biocontrol **55**:55-73.

Hochberg, M. 1989. The potential role of pathogens in biological control. Nature **337**:262-265.

Holt, R. 1997. On the evolutionary stability of sink populations. Evolutionary Ecology **11**:723-731.

Hsiao, W. F., M. J. Bidochka, and G. G. Khachatourians. 1992. Effect of temperature and relative-humidity on the virulence of the entomopathogenic fungus, *Verticillium lecanii*, toward the oat-bird berry aphid, *Rhopalosiphum padi* (Hom, Aphididae). Journal of Applied Entomology-Zeitschrift Fur Angewandte Entomologie **114**:484-490.

Huber, L., L. V. Madden, and B. D. L. Fitt. 2006. Environmental biophysics applied to the dispersal of fungal spores by rain-splash. Pages 417-444 *in* B. M. Cooke, D. G. Jones, and B. Kaye, editors. The Epidemiology of Plant Diseases. Springer Netherlands.

Jackson, D., J. Skillman, and J. Vandermeer. 2012. Indirect biological control of the coffee leaf rust, *Hemileia vastatrix*, by the entomogenous fungus *Lecanicillium lecanii* in a complex coffee agroecosystem. Biological Control **61**:89-97.

Jackson, D., J. Vandermeer, D. Allen, and I. Perfecto. In review. Self-organization of background habitat determines the nature of population spatial structure. Ecology.

Jackson, D., J. Vandermeer, and I. Perfecto. 2009. Spatial and temporal dynamics of a fungal pathogen promote pattern formation in a tropical agroecosystem. The Open Ecology Journal **2**:62-73.

Jackson, D., K. Zemenick, and G. Huerta. In press. Occurrence in the soil and dispersal of *Lecanicillium lecanii*, a fungal pathogen of the green coffee scale, *Coccus viridis*. Tropical and Subtropical Agroecosystems.

Jacob, F. 1977. Evolution and tinkering. Science **196**:1161-1166.

Jarvis, D. I., C. Padoch, and H. D. Cooper, editors. 2007. Managing biodiversity in agricultural ecosystems. Columbia University Press, New York.

Kamp, A. and M. Bidochka. 2002. Conidium production by insect pathogenic fungi on commercially available agars. Letters in Applied Microbiology **35**:74-77.

Kanoun-Boulé, M., M. B. De Albuquerque, C. Nabais, and H. Freitas. 2008. Copper as an environmental contaminant: phytotoxicity and human health implications. Pages 653-678 *in* M. Prasad, editor. Trace Elements as Contaminants and Nutrients. John Wiley and Sons Inc., New York.

Kéfi, S., M. Rietkerk, C. L. Alados, Y. Pueyo, V. P. Papanastasis, A. Elaich, and P. C. De Ruiter. 2007. Spatial vegetation patterns and imminent desertification in Mediterranean arid ecosystems. Nature **449**:213-217.

Kéfi, S., M. Rietkerk, M. Roy, A. Franc, P. C. De Ruiter, and M. Pascual. 2011. Robust scaling in ecosystems and the meltdown of patch size distributions before extinction. Ecology Letters **14**:29-35.

Kerr, B., C. Neuhauser, B. J. M. Bohannan, and A. M. Dean. 2006. Local migration promotes competitive restraint in a host–pathogen 'tragedy of the commons'. Nature **442**:75-78.

Khalil, S., J. Bartos, and Z. Landa. 1985a. Effectiveness of *Verticillium lecanii* to reduce populations of aphids under glasshouse and field conditions. Agriculture, Ecosystems and Environment **12**:151-156.

Khalil, S., M. Shah, and M. Naeem. 1985b. Laboratory studies on the compatibility of the entomopathogenic fungus *Verticillium lecanii* with certain pesticides. Agriculture, Ecosystems and Environment **13**:329-334.

Kim, K. and C. Harvell. 2004. The rise and fall of a six-year coral-fungal epizootic. The American Naturalist **164**:52-63.

Kiss, L. 2003. A review of fungal antagonists of powdery mildews and their potential as biocontrol agents. Pest Management Science **59**:475-483.

Klausmeier, C. 1999. Regular and irregular patterns in semiarid vegetation. Science **284**:1826-1828.

Klingen, I., J. Eilenberg, and R. Meadow. 2002. Effects of farming system, field margins and bait insect on the occurrence of insect pathogenic fungi in soils. Agriculture, Ecosystems & Environment **91**:191-198.

Kouvelis, V., A. Sialakouma, and M. Typas. 2008. Mitochondrial gene sequences alone or combined with ITS region sequences provide firm molecular criteria for the classification of *Lecanicillium* species. Mycological Research **112**:829-844.

Kouvelis, V., R. Zare, P. Bridge, and M. Typas. 1999. Differentiation of mitochondrial subgroups in the *Verticillium lecanii* species complex. Letters in Applied Microbiology **28**:263-268.

Kritzer, J. P. and P. F. Sale. 2004. Metapopulation ecology in the sea: from Levins' model to marine ecology and fisheries science. Fish and Fisheries **5**:131-140.

Kushalappa, A. C. and A. B. Eskes. 1989. Advances in coffee rust research. Annual Review of Phytopathology **27**:503-531.

Legendre, P. and M. Fortin. 1989. Spatial pattern and ecological analysis. Plant Ecology **80**:107-138.

Lenton, T. M., H. Held, E. Kriegler, J. W. Hall, W. Lucht, S. Rahmstorf, and H. J. Schellnhuber. 2008. Tipping elements in the Earth's climate system. Proceedings of the National Academy of Sciences **105**:1786-1793.

Levins, R. 1969. Some demographic and genetic consequences of environmental heterogeneity for biological control. Bulletin of the Entomological Society of America **15**:237-240.

Lewontin, R. C. and R. Levins. 2007. Biology under the influence: Dialectical essays on ecology, agriculture, and health. Monthly Review Press, New York.

Liere, H., D. Jackson, and J. Vandermeer. In review. Population regulation in a coffee agroecosystem: an example of how ecological complexity promotes spatial heterogeneity. PLoS ONE.

Liere, H. and I. Perfecto. 2008. Cheating on a mutualism: indirect benefits of ant attendance to a coccidophagous coccinellid. Environmental Entomology **37**:143-149.

Lion, S. and M. V. Baalen. 2008. Self-structuring in spatial evolutionary ecology. Ecology Letters **11**:277-295.

Livingston, G., A. White, and C. Kratz. 2008. Indirect interactions between ant-tended hemipterans, a dominant ant *Azteca instabilis* (Hymenoptera: Formicidae), and shade trees in a tropical agroecosystem. Environmental Entomology **37**:734-740.

Loland, J. and B. Singh. 2004. Copper contamination of soil and vegetation in coffee orchards after long-term use of Cu fungicides. Nutrient Cycling in Agroecosystems **69**:203-211.

Lomolino, M., G. Smith, and M. Willig. 2001. Dynamic biogeography of prairie dog (*Cynomys ludovicianus*) towns near the edge of their range. Journal of Mammalogy **82**:937-945.

Madden, L. 1997. Effects of rain on splash dispersal of fungal pathogens. Canadian Journal of Plant Pathology **19**:225-230.

Martinez, E. and W. Peters. 1996. La cafeticultura biológica: la finca Irlanda como estudio de caso de un deseño agricoecológico. Pages 159-183 *in* J. T. Arriaga, F. L. González, R. C. Arózqueta, and P. T. Lima, editors. Ecología aplicada a la agricultura: Temas selectos de México. Universidad Autonomo Metropolitana, Unidad Xochimilco, DF, Mexico.

McCook, S. 2006. Global rust belt: *Hemileia vastatrix* and the ecological integration of world coffee production since 1850. Journal of Global History **1**:177-195.

Meyling, N. and J. Eilenberg. 2006. Occurrence and distribution of soil borne entomopathogenic fungi within a single organic agroecosystem. Agriculture, Ecosystems & Environment **113**:336-341.

Meyling, N. and J. Eilenberg. 2007. Ecology of the entomopathogenic fungi *Beauveria bassiana* and *Metarhizium anisopliae* in temperate agroecosystems: Potential for conservation biological control. Biological Control **43**:145-155.

Michaud, J. and H. Browning. 1999. Seasonal abundance of the brown citrus aphid, *Toxoptera citricida* (Homoptera: Aphididae) and its natural enemies in Puerto Rico. The Florida Entomologist **82**:424-447.

Milinski, M., D. Semmann, and H.-J. Krambeck. 2002. Reputation helps solve the 'tragedy of the commons'. Nature **415**:424-426.

Moilanen, A. and I. Hanski. 1998. Metapopulation dynamics: Effects of habitat quality and landscape structure. Ecology **79**:2503-2515.

Moricca, S. and A. Ragazzi. 2008. Biological and integrated means to control rust diseases. Pages 303-329 *in* A. Ciancio and K. G. Mukerji, editors. Integrated Management of Diseases Caused by Fungi, Phytoplasma and Bacteria. Springer, Netherlands.

Newman, M. 2005. Power laws, Pareto distributions and Zipf's law. Contemporary Phys. **46**:323-351.

Niblack, L. T. and S. R. Hussey. 1987. Extracción de nematodes del suelo y de tejidos vegetales. Pages 235-242 *in* B. M. Zuckerman, W. F. Mai, and M. B. Harrison, editors. Fitonematologia: manual de laboratorio. Centro Agronómico Tropical de Investigación y Enseñanza, Turrialba, Costa Rica.

Nowak, M. A. 2006. Five rules for the evolution of cooperation. Science **314**:1560-1563.

Pascual, M., M. Roy, F. Guichard, and G. Flierl. 2002. Cluster size distributions: signatures of self-organization in spatial ecologies. Philosophical Transactions of the Royal Society B: Biological Sciences **357**:657-666.

Paul, N. D. and D. Gwynn-Jones. 2003. Ecological roles of solar UV radiation: towards an integrated approach. Trends in Ecology & Evolution **18**:48-55.

Pell, J., J. Hannam, and D. Steinkraus. 2010. Conservation biological control using fungal entomopathogens. Biocontrol **55**:187-198.

Perfecto, I., A. Mas, T. Dietsch, and J. Vandermeer. 2003. Conservation of biodiversity in coffee agroecosystems: a tri-taxa comparison in southern Mexico. Biodiversity Conservation **12**:1239-1252.

Perfecto, I. and J. Vandermeer. 2002. Quality of agroecological matrix in a tropical montane landscape: ants in coffee plantations in southern Mexico. Conservation Biology **16**:174-182.

Perfecto, I. and J. Vandermeer. 2006. The effect of an ant-hemipteran mutualism on the coffee berry borer (*Hypothenemus hampei*) in southern Mexico. Agriculture Ecosystems & Environment **117**:218-221.

Perfecto, I. and J. Vandermeer. 2008a. Biodiversity conservation in tropical agroecosystems. Annals of the New York Academy of Sciences **1134**:173-200.

Perfecto, I. and J. Vandermeer. 2008b. Spatial pattern and ecological process in the coffee agroforestry system. Ecology **89**:915-920.

Perfecto, I. and J. Vandermeer. 2010. The agroecological matrix as alternative to the land-sparing/agriculture intensification model. Proceedings of the National Academy of Sciences **107**:5786-5791.

Philpott, S. M., I. Perfecto, and J. Vandermeer. 2006. Effects of management intensity and season on arboreal ant diversity and abundance in coffee agroecosystems. Biodiversity and Conservation **15**:139-155.

Pineda-Krch, M. 2008. GillespieSSA: Implementing the stochastic simulation algorithm in R. J Stat Softw **25**:1-18.

Pueyo, S. and R. Jovani. 2006. Comment on "A keystone mutualism drives pattern in a power function. Science **313**:1739c.

Pulliam, H. R. 1988. Sources, sinks, and population regulation. The American Naturalist **132**:652-661.

Rand, D. A., M. Keeling, and H. B. Wilson. 1995. Invasion, stability, and evolution to criticality in spatially extended, artificial host-pathogen ecologies. Proceedings of the Royal Society London B **259**:55-63.

Ravensberg, W. J., M. Malais, and D. A. Vanderschaaf. 1990. *Verticillium lecanii* as a microbial insecticide against glasshouse whitefly. Brighton Crop Protection Conference - Pests and Diseases, 1990 : Proceedings, Vols 1-3:265-268.

Reddy, K. B. and P. K. Bhat. 1989. Effect of relative humidity and temperature on the biotic agents of green scale *Coccus viridis* (Green). Journal of Coffee Research **19**:82-87.

Reimer, N. J. and J. W. Beardsley. 1992. Epizootic of white halo fungus, *Verticillium lecanii* (Zimmerman), and effectiveness of insecticides on *Coccus viridis* (Green) (Homoptera: Coccidae) on coffee at Kona, Hawaii. Proceedings, Hawaiian Entomological Society **31**:73-81.

Reimer, N. J., M. Cope, and G. Yasuda. 1993. Interference of *Pheidole megacephala* (Hymenoptera: Formicidae) with biological control of *Coccus viridis* (Homoptera: Coccidae) in coffee. Environmental Entomology **22**:483-488.

Resampling. 2006. Resampling stats for Excel user's guide version 3. Resampling Stats.

Reynolds, H. T. and C. R. Currie. 2004. Pathogenicity of *Escovopsis weberi*: the parasite of attine ant-microbe symbiosis directly consumes the ant-cultivated fungus. Mycologia **96**:955-959.

Rietkerk, M., M. Boerlijst, F. van Langevelde, R. HilleRisLambers, J. de Koppel, L. Kumar, H. Prins, and A. de Roos. 2002. Self-organization of vegetation in arid ecosystems. The American Naturalist **160**:524-530.

Rietkerk, M., S. Dekker, P. de Ruiter, and J. van de Koppel. 2004. Self-organized patchiness and catastrophic shifts in ecosystems. Science **305**:1926-1929.

Rietkerk, M. and J. van de Koppel. 2008. Regular pattern formation in real ecosystems. Trends in Ecology & Evolution **23**:169-175.

Roditakis, E., I. Couzin, K. Balrow, and N. Franks. 2000. Improving secondary pick up of insect fungal pathogen conidia by manipulating host behaviour. Annals of Applied Biology **137**:329-335.

Rodríguez Dos Santos, A. and E. del Pozo Núñez. 2003. Aislamiento de hongos entomopatógenos en Uruguay y su virulencia sobre *Trialeurodes vaporariorum* West. Agrociencia **7**:71-78.

Rohani, P., T. Lewis, D. Grünbaum, and G. Ruxton. 1997. Spatial self-organization in ecology: pretty patterns or robust reality? Trends in Ecology & Evolution **12**:70-74.

Rohani, P., R. M. May, and M. P. Hassell. 1996. Metapopulations and equilibrium stability: The effects of spatial structure. Journal of Theoretical Biology **181**:97-109.

Roy, H. E., J. K. Pell, and P. G. Alderson. 2001. Targeted dispersal of the aphid pathogenic fungus *Erynia neoaphidis* by the aphid predator *Coccinella septempunctata*. Biocontrol Science and Technology **11**:99-110.

Sauerborn, J., D. Müller-Stöver, and J. Hershenhorn. 2007. The role of biological control in managing parasitic weeds. Crop Protection **26**:246-254.

Scanlon, T. M., K. K. Caylor, S. A. Levin, and I. Rodriguez-Iturbe. 2007. Positive feedbacks promote power-law clustering of Kalahari vegetation. Nature **449**:209-212.

Schaffer, J. D., L. J. Eshelman, and D. Offutt. 1991. Spurious correlations and premature convergence in genetic algorithms. Pages 102-112 *in* G. Rawlins, editor. Foundations of Genetic Algorithms. Morgan Kaufmann, San Mateo, CA.

Scheffer, M., J. Bascompte, W. A. Brock, V. Brovkin, S. R. Carpenter, V. Dakos, H. Held, E. H. v. Nes, M. Rietkerk, and G. Sugihara. 2009. Early-warning signals for critical transitions. Nature **461**:53-59.

Scheffer, M. and E. H. Nes. 2007. Shallow lakes theory revisited: various alternative regimes driven by climate, nutrients, depth and lake size. Hydrobiologia **584**:455-466.

Scholz, N. L., E. Fleishman, L. Brown, I. Werner, M. L. Johnson, M. L. Brooks, C. L. Mitchelmore, and D. Schlenk. 2012. A perspective on modern pesticides, pelagic fish declines, and unknown ecological resilience in highly managed ecosystems. BioScience **62**:428-434.

Scialabba, N. E.-H. and C. Hattam, editors. 2002. Organic agriculture, environment, and food security. Food and Agriculture Organization of the United Nations, Rome, Italy.

Shah, P. and J. Pell. 2003. Entomopathogenic fungi as biological control agents. Applied microbiology and biotechnology **61**:413-423.

Shaw, D. E. 1988. *Verticillium lecanii* a hyperparasite on the coffee rust pathogen in Papua New Guinea. Australasian Plant Pathology **17**:2-3.

Shi, Y., X. Xu, and Y. Zhu. 2009. Optimization of *Verticillium lecanii* spore production in solid-state fermentation on sugarcane bagasse. Applied microbiology and biotechnology **82**:921-927.

Shiomi, H. F., H. S. A. Silva, I. S. Melo, F. V. Nunes, and W. Bettiol. 2006. Bioprospecting endophytic bacteria for biological control of coffee leaf rust. Scientia Agricola **63**:32-39.

Sitch, J. C. and C. W. Jackson. 1997. Pre-penetration events affecting host specificity of *Verticillium lecanii*. Mycological Research **101**:535-541.

Smith, A. 1776. An inquiry into the nature and causes of the wealth of nations. Project Gutenberg.

Smith, J. M. 1978. Optimization theory in evolution. Annual Review of Ecology and Systematics **9**:31-56.

Socolar, J. E. S. and W. G. Wilson. 2001. Evolution in a spatially structured population subject to rare epidemics. Physical Review E **63**:8.

Solé, R. V. and J. Bascompte. 2006. Self-organization in complex ecosystems. Princeton University Press, Princeton, N.J.

Staver, C., F. Guharay, D. Monterroso, and R. Muschler. 2001. Designing pest-suppressive multistrata perennial crop systems: shade-grown coffee in Central America. Agroforestry Systems **53**:151-170.

Strogatz, S. H. 1994. Nonlinear dynamics and chaos with applications to physics, biology, chemisty and engineering. Perseus, New York.

Suffert, F., É. Latxague, and I. Sache. 2009. Plant pathogens as agroterrorist weapons: assessment of the threat for European agriculture and forestry. Food Security **1**:221-232.

Sung, G. H., J. W. Spatafora, R. Zare, K. T. Hodge, and W. Gams. 2001. A revision of *Verticillium* sect. Prostrata. II. Phylogenetic analyses of SSU and LSU nuclear rDNA sequences from anamorphs and teleomorphs of the Clavicipitaceae. Nova Hedwigia **72**:311-328.

Szilágyi, A., I. Scheuring, D. P. Edwards, J. Orivel, and D. W. Yu. 2009. The evolution of intermediate castration virulence and ant coexistence in a spatially structured environment. Ecology Letters:1-11.

Te Beest, D., X. Yang, and C. Cisar. 1992. The status of biological control of weeds with fungal pathogens. Annual Review Phytopathology **30**:637-657.

Thies, C. and T. Tscharntke. 1999. Landscape structure and biological control in agroecosystems. Science **285**:893-895.

Tilman, D. and P. Kareiva. 1997. Spatial ecology: the role of space in population dynamics and interspecific interactions. Princeton University Press, Princeton, N.J.

Tuininga, A., J. Miller, S. Morath, and T. Daniels. 2009. Isolation of entomopathogenic fungi from soils and *Ixodes scapularis* (Acari: Ixodidae) ticks: prevalence and methods. Journal of Medical Entomology **46**:557-565.

Uno, S. 2007. Effects of management intensification on coccids and parasitic hymenopterans in coffee agroecosystems in Mexico. University of Michigan, Ann Arbor, MI.

van de Koppel, J., M. Rietkerk, N. Dankers, and P. Herman. 2005. Scale-dependent feedback and regular spatial patterns in young mussel beds. The American Naturalist **165**:E66-E77.

Vandermeer, J. 2011. The inevitability of surprise in agroecosystems. Ecological Complexity **8**:377-382.

Vandermeer, J. and I. Perfecto. 2006a. A keystone mutualism drives pattern in a power function. Science **311**:1000-1002.

Vandermeer, J. and I. Perfecto. 2006b. Response to Comments on "A keystone mutualism drives pattern in a power function". Science **313**:1739.

Vandermeer, J., I. Perfecto, G. Ibarra Nuñez, S. Phillpott, and A. Garcia Ballinas. 2002. Ants (*Azteca* sp.) as potential biological control agents in shade coffee production in Chiapas, Mexico. Agroforestry Systems **56**:271-276.

Vandermeer, J., I. Perfecto, and H. Liere. 2009. Evidence for hyperparasitism of coffee rust (*Hemileia vastatrix*) by the entomogenous fungus, *Lecanicillium lecanii*, through a complex ecological web. Plant Pathology **58**:636-641.

Vandermeer, J., I. Perfecto, and S. Philpott. 2008. Clusters of ant colonies and robust criticality in a tropical agroecosystem. Nature **451**:457-459.

Vandermeer, J., I. Perfecto, and S. Philpott. 2010a. Ecological complexity and pest control in organic coffee production: uncovering an autonomous ecosystem service. BioScience **60**:527-537.

Vandermeer, J., I. Perfecto, and N. Schellhorn. 2010b. Propagating sinks, ephemeral sources and percolating mosaics: conservation in landscapes. Landscape Ecology **25**:509-518.

Waller, J. 1982. Coffee rust--epidemiology and control. Crop Protection **1**:385-404.

Werner, E., K. Yurewicz, D. Skelly, and R. Relyea. 2007. Turnover in an amphibian metacommunity: the role of local and regional factors. Oikos **116**:1713-1725.

Whitehorn, P. R., S. O'Connor, F. L. Wackers, and D. Goulson. 2012. Neonicotinoid pesticide reduces bumble bee colony growth and queen production. Science.

Whitley, D. 1994. A genetic algorithm tutorial. Statistics and Computing **4**:65-85.

Willer, H. and L. Kilcher, editors. 2010. The world of organic agriculture: statistics and emerging trends 2010. IFOAM, Bonn and FiBL, Frick.

Wissel, C. 1984. A universal law of the characteristic return time near thresholds. Oecologia **65**:101-107.

Wohlfahrt-Veje, C., H. R. Andersen, T. K. Jensen, P. Grandjean, N. E. Skakkebaek, and K. M. Main. 2012. Smaller genitals at school age in boys whose mothers were exposed to non-persistent pesticides in early pregnancy. International Journal of Andrology:1-8.

Zare, R. and W. Gams. 2001. A revision of *Verticillium* section Prostrata. IV. The genera *Lecanicillium* and *Simplicillium* gen. nov. Nova Hedwigia **73**:1-50.

Zare, R., W. Gams, and A. Culham. 2000. A revision of *Verticillium* sect. Prostrata - I. Phylogenetic studies using ITS sequences. Nova Hedwigia **71**:465-480.

Zare, R., W. Gams, and H. C. Evans. 2001. A revision of *Verticillium* section Prostrata. V. The genus *Pochonia*, with notes on *Rotiferophthora*. Nova Hedwigia **73**:51-86.

Zimmermann, G. 1986. The *Galleria* bait method for detection of entomopathogenic fungi in soil. Journal of Applied Entomology **102**:213-215.

Zinck, R. D. and V. Grimm. 2009. Unifying wildfire models from ecology and statistical physics. The American Naturalist **174**:E170-E185.