# Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer

by

Daniel N. Coore

SB, Massachusetts Institute of Technology, 1994
SM, Massachusetts Institute of Technology, 1994

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
January 28, 1999

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Gerald J. Sussman
Matsushita Professor of Electrical Engineering, MIT
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Harold Abelson
Class of 1922 Professor of Computer Science and Engineering, MIT
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

**Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer**

by
Daniel N. Coore

Submitted to the Department of Electrical Engineering and Computer Science on January 28, 1999, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

## Abstract

An amorphous computing medium is a system of irregularly placed, asynchronous, locally interacting computing elements. I have demonstrated that amorphous media can be configured by a program, common to all computing elements, to generate highly complex prespecified patterns. For example, I can specify that an amorphous medium manifest a pattern representing the interconnection structure of an arbitrary electrical circuit.

My strategy is inspired by a botanical metaphor based on growing points and tropisms. To make this strategy explicit, I have developed the Growing Point Language (GPL). A growing point is a locus of activity in an amorphous medium. A growing point propagates through the medium by transferring its activity from one computing element to a neighbor. As a growing point passes through the medium it effects the differentiation of the behaviors of the computing elements it visits. The trajectory of the growing point is controlled by signals that are automatically carried through the medium from other differentiated elements. Such a response is called a tropism. In this way a GPL program can exploit locality to make crude geometric inferences.

There is a wide variety of patterns that are expressible in GPL. Examples include: Euclidean constructions, branching structures and simple text. I prove that amorphous media can be programmed to draw any prespecified planar graph, and I obtain upper bounds on the amount of storage required by the individual processors to realize such a graph. I also analyze how the effectiveness of GPL programs depends upon the distribution of the computing elements.

Thesis Supervisor: Gerald J. Sussman
Title: Matsushita Professor of Electrical Engineering, MIT

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor of Computer Science and Engineering, MIT

Thesis Reader: Thomas F. Knight
Title: Senior Research Scientist, Artificial Intelligence Laboratory, MIT

## Acknowledgments[1]

There are a number of people to whom I am indebted for their support and encouragement.

Kanchi, my wife and strongest supporter, who proofread my drafts, discussed my inchoate ideas with me and helped me to develop them, and generally kept me on track over the past year.

Gerry Sussman for issuing the challenge in the first place, for providing plenty of nurturing guidance and energetic encouragement along the way. Hal Abelson for the many useful comments and keen advice on how to improve the dissertation.

Tom Knight for the useful and encouraging comments at the beginning of the project. Jacob Katzenelson for carefully reading the proposal as well as the dissertation and asking generally good questions. Those questions forced me to address many issues I might otherwise have overlooked.

Radhika Nagpal, Ron Weiss, Erik Rauch, Jeremy Zucker, Piotr Mitros, Chris Laas and the rest of Swiss group for providing helpful input at various times throughout the whole endeavour.

Rajeev Surati, Elmer Hung and Natalya Cohen for offering playful competition and supportive encouragement along the way.

Hoang Tran, Nyssim Lefford and Frank Cortez, my supportive apartment mates, who would offer to feed me (when Kanchi wasn't around) whenever it was clear that I had forgotten to eat again.

Janaki Bandara, my sister-in-law, Anthony and Rita my brother and sister, my parents, my parents-in-law and generally everybody else who remembered I was suffering through a dissertation and cared enough to ask every now and then how things were progressing.

[1]The research for this report was conducted at the MIT Artificial Intelligence Laboratory under the Amorphous Computing project. Support was provided in part by DARPA under the contract number N00014-96-1-1228 administered by ONR.

*Dedicated to the memory of*

*Mrs. Elaine Imogene Barton (Nanny),*

*whose unmatched service is a constant source of inspiration.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The cells of an embryo specialize and develop under the control of a common genetic program housed in each cell. The organism that forms is almost always correctly formed: the appropriate body parts are present and are connected appropriately. By what means is the morphology of the organism expressed in its genetic code? How do two distantly separated cells specialize in different ways to perform different tasks, despite the fact that they are independently directed by identical genetic programs?

This dissertation presents a programming paradigm and an implemented programming language that can control a programming environment that is similar to a collection of interacting biological cells. Specifically, assume that we are given a system of myriad computational elements, irregularly placed, interacting only locally, and asynchronously running identical programs. I present the Growing Point Language (GPL) for specifying the construction of interconnect topologies on that system. A GPL program is expressed in terms of concepts that are meaningful only when the system is regarded as a single programmable entity. Yet when that program is interpreted simultaneously on every element of the system—in much the same way that the cells of any embryo "execute" their common genetic program—it is capable of producing prespecified effects that may involve coordinated differentiation (of behaviour) on distantly separated elements.

The principal concept in GPL is the *growing point*. It is a locus of activity that describes a path through connected elements in the system. A growing point propagates through the system by transferring its activity from one computing element to a neighbour. As a growing point passes through the system, it effects the differentiation of the behaviours of the elements it visits. The trajectory of the growing point is controlled by signals that are automatically carried through the system from other differentiated elements.

The work presented here is not an attempt to model biological processes. The goal is to try to understand, from an engineering perspective, how macroscopic properties of a system arise from microscopic perturbations that take place within it. Specifically, we want to know when given a macroscopic goal, what microscopic actions ought to be prescribed in order to achieve it.

### 1.0.1 The Growing Point Language

The implementation of GPL rises to this challenge by providing a means of expressing logical global topological constraints that can be enforced through local interactions by exploiting interconnect geometry. We are able to specify non-trivial topologically constrained patterns from a modest number of initial conditions via a GPL program. Figure 1-1 shows an example

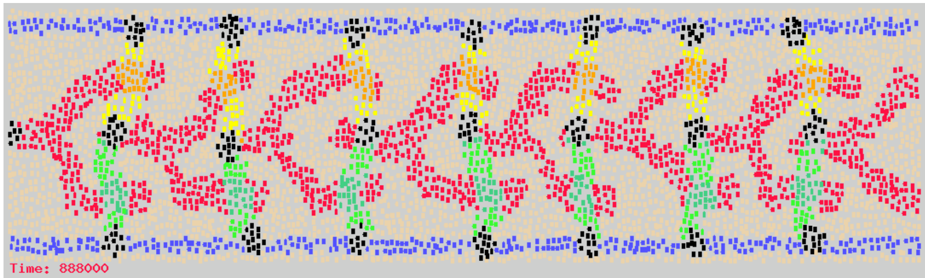


Figure 1-1: Each rectangle represents a computational element. Each one is coloured according to a value maintained in the element's state. The size of a neighbourhood is the thickness of one of the interior lines.

of such a pattern, which was obtained from a small program encoding about seventeen computational states. The initial conditions included the two shaded lines at the top and bottom of the pattern and a single point at the left edge.

By implementing an interpreter that translates system-level GPL commands to element-level instructions, we provide insight into the relationship between emergent macroscopic properties and microscopic actions. Our implementation makes minimalist assumptions about the computational capabilities of the system elements. Therefore, by exhibiting topologically constrained patterns like that in Figure 1-1 we demonstrate that the necessary conditions for controlling such global organization are few.

The GPL system translates a GPL program into code fragments that are executed on each computational element by a message-passing event driven loop. Since the method of translation is oblivious to the identity of the element that eventually runs the code, *every element inherits identical code fragments*. However, as can be seen from Figure 1-1 some elements must eventually assume different states from others.

The problem then is to generate code fragments that will differentiate, in a controlled way, the behaviour of each element they run on. That differentiation can be controlled in two ways: in response to changes in local state on the element, or in response to messages received by the element. Therefore, the behavioural differentiation at each element must be highly coordinated in order to meet the non-local constraints originally expressed by the GPL program.

Figure 1-2 illustrates a pattern whose essential feature is the connectivity of the lines in it. We call this pattern *topologically constrained*. Figure 1-3 illustrates what it means for the system to *draw* the pattern shown in Figure 1-2. Each disc represents a system element, and each colour represents a label that the element has assigned itself during the drawing computation. Connected regions of common labels can be mapped, via the colour codings shown, to regions in the given pattern in Figure 1-2 in a way that preserves its topology. The pattern in Figure 1-2 represents an example of the type of patterns that GPL is capable of expressing.

We make two claims:

- Strictly local communication and a common program are sufficient conditions to express non-local topologically constrained (differentiation) patterns. Such patterns are expressed by GPL programs.

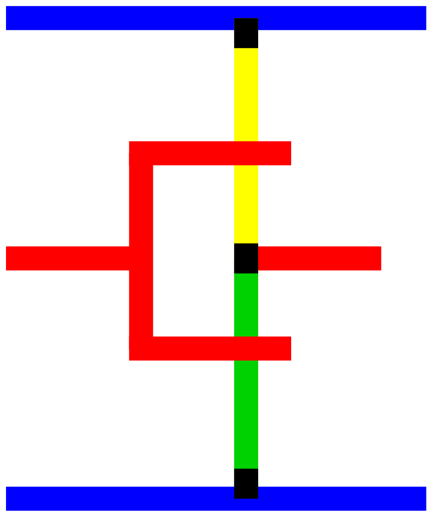

Figure 1-2: A topologically constrained pattern

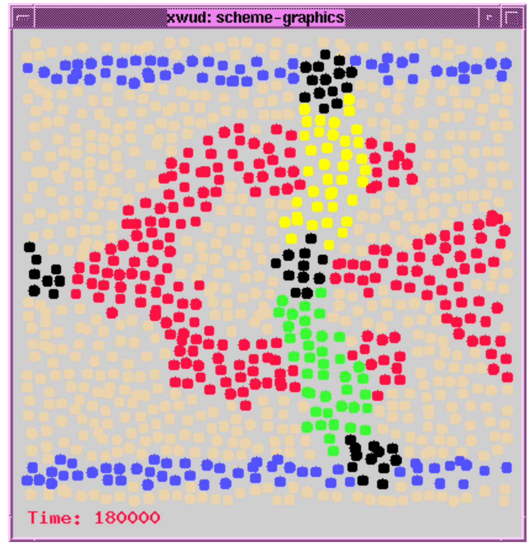


Figure 1-3: The realization of that pattern on the system of elements

- The interconnect topology of any planar graph is expressible as a GPL program.

Thesis:

> *Any planar interconnect topology can be drawn by a system of irregularly connected, identically programmed, asynchronous, locally-interacting computational elements.*

### 1.0.2 Context

One broad example of the endeavour to control macroscopic effects through the direction of microscopic action is genetic engineering. There, biologists attempt to obtain a specific phenotypic property (for example, improved resistance to disease) by genetic manipulations. A good specific example of work that shares our goals is Slack's theory of morphology. Slack [46] presents a description of how the morphology of an organism might be *designed* by manipulating its genes.

The point here is that we want to learn how to *actively* control global behaviours when we have access to only local manipulations. Under ideal conditions, we fully understand the effect of each local manipulation, and so we can focus on understanding the nature and origins of the emergent properties of the system. In the case of genetic engineering, we do not fully understand the effects of every primitive operation, so the problem of controlling emergence is compounded. We are hindered from gaining as much insight into our problem than would be the case if we had a complete understanding of all the local operations.

Unfortunately, there are few synthetic examples of research with similar goals to ours. One example from distributed control theory is research to show that the shape of a stressed beam can be controlled using sensors and actuators that are distributed along the length of the beam [21, 24, 7]. Other examples are in computational modeling of biological processes:

- A Lindenmayer system [43] hybridized with diffusion can be programmed (by local rules) to generate patterns that model axonal growth in mammals [26].

- A lattice model of protein folding has been used to investigate the complexity of the inverse protein folding problem [55]. The inverse folding problem attempts to find a sequence of residues that will fold to some given target native conformation.

In each of these three examples there is a global goal that must be achieved as a result of local actions. So in that respect, they are similar to the work we present.

There are at least two important differences between these examples and the approach described in this dissertation. The first is that all of those examples used regular discretizations of space. The second difference is that none of those examples had as strict locality constraints as we do. For example, in the beam controller, actuation decisions were made with complete global knowledge, and in the model for axonal growth, arbitrary angles could be specified at any location which implied an implicit global coordinate frame. In all of those examples, a physical system was being modeled, so it made sense to facilitate the analysis by making the discretization of space regular or permitting non-local computations when necessary. In this dissertation, we are primarily concerned with the organizational principles involved in controlling a large system of loosely coupled, locally interacting elements. We do not assume that these elements are regularly arranged (without precluding the possibility that they are), and we strictly enforce the locality constraints for communication and computation on the elements.

In this respect, our goals are more aligned with those of Distributed Computing [34, 31], where the primary concern is how to coordinate the operation of networks of computers. However, in Amorphous Computing, we retain the concept of space to limit the types of connections that can be made between the system elements. Therefore the setting for this dissertation is unique: we do not claim to model any particular physical system, yet we retain physical spatial constraints in determining the interconnect of the system. In general, we seek to understand how geometric constraints on a large system's elements may be exploited to effectively control the global behaviour of the system.

### 1.0.3 Motivations

Given the recent advances in Biotechnology and microfabrication technologies, we may soon have programming access to large systems of computational elements. The numbers of elements would be enormous, significantly larger than any system we currently have. These elements might be realised as biological cells or more familiar silicon-based hardware; but they will, of necessity, be small and therefore resource-limited. It is also prudent to assume that they will probably be unreliable and loosely coupled in irregular interconnection networks. If we were given such a system today, we would be hard-pressed to do its computational power any justice, not because we lack difficult tasks, but because we lack the understanding to deal with such a system.

The study of Amorphous Computing addresses this problem by researching techniques that enable us to engineer useful system-wide behaviours. In order to develop our ideas on differentiating local behaviour to obtain globally coherent goals, we needed to solve a more specific problem. We chose the generation of patterns of interconnect topologies as our model global behaviour because:

1. it is easy to verify when we have achieved our goal,
2. specific information must propagate non-locally to specific sites, so the problem is non-trivial,

3. it represents an unprecedented level of control over macroscopic properties derived from the manipulation of microscopic parameters.

Ultimately we would like to learn and understand general principles that will aid us in controlling immensely complex systems. We believe that such systems are imminent. Therefore it is extremely important for us to discover new ways of reasoning that will allow us to properly marshal their resources. Such novel reasoning techniques will undoubtedly involve abstractions that present a simple interface to the system. But these techniques must go one step further to include the automatic implementation of the interface through cooperation of the elements. By studying a non-trivial problem such as interconnect topology layout generation, we hope to gain some insights into the nature of that interface and its implementation.

## 1.1 The Amorphous Computing Model

An amorphous computer consists of myriad computing particles embedded in physical space. The particles have limited memory and modest computational resources; they run asynchronously, communicate with each other over a very limited physical distance, and are placed arbitrarily with respect to each other. We assume that the receivers on each particle are omni-directional and therefore a particle cannot immediately determine the relative location of a transmitting neighbour. The large number of particles precludes programming each one individually. Therefore, the programs executed by these particles are constrained to be initally identical (although the programs may contain multiple entry points). At the beginning of a computation, the only way to distinguish a particle from the rest is to give it a different entry point into the common program. Subsequent variation in behaviours among the particles must be self-induced: whether by communication with neighbours, by sensory input values or by random numbers.

The model considered here further constrains the particles to be static. So their interconnect, although unknown *a priori*, is fixed for the duration of any computation. The message of a single transmission must be shorter than a statically specified number of bits. Messages take time to deliver, during which a particle is allowed to continue computing. Each particle is built to encode and decode messages in exactly the same way. Therefore, the low-level communication protocol is not subject to software controlled variation (this does not preclude a particle from dynamically adjusting the manner in which it interprets the information in a message). The details of the programming language that we shall use for these devices are presented in Chapter 3.

The types of global behaviour that we consider will be restricted to those that specify a goal pattern for the system to assume. If the system were the collection of cells of an embryo, an example of this type of global behaviour would be the connectedness of the organism. Since the particles in this amorphous computing model are static, the computing particles assign themselves labels to correspond to features in the given specification of the goal pattern. Regions in the space of locations of the particles are determined by connected particles of common labels. A labeling is successful if there is a topology-preserving map from the regions obtained from labeling and the given pattern.

### 1.1.1 Justification for the Model

The physical realization of the computational elements determines the representation of our computations, and therefore the kinds of operations that will be convenient to perform. For example, if our elements are biological cells, then our computational states and data may be represented by chemical concentrations of various signal substances. If our devices are even smaller, say molecules, then our states and signals might be represented by electron distributions. It is already a bold assumption that we might be able to manipulate these microscopic elements in the way we manipulate computers. So it is only natural that we choose to restrict the resources available to each computational element.

We have taken a minimalist view of the capabilities of our computational particles so that our results will be general and robust. We do not assume that a particle can detect the direction from which a received message came, even though it is conceivable that that might be possible (for example, if our particles are cells and the messages are chemicals that attach to the cell's membrane). The assumption that particles have restricted resources encourages us to develop scalable algorithms that rely on only local cooperation among the particles. Regarding the initial conditions, while we assume that we can statically manipulate these particles arbitrarily, we do not want to extend that assumption to dynamic, real-time manipulation. For that reason, we constrain the model to accept external intervention only when the initial conditions need to be setup. Everything else should be autonomous, that is directed by the program.

Even the assumption that the particles are computationally equivalent to space-bound Turing machines may be too strong. Indeed, our results indicate that for the purposes of the problem addressed by this thesis, we may not need to assume as much computational power as a space-bound Turing machine. However, we chose to start with that assumption so that we could focus on the real issues at hand: moderating the interactions between the elements in an orderly manner, and doing so with a control specification that is initially identical on each device.

## 1.2 Encoding Patterns

Suppose that we are given the pattern in Figure 1-2 to draw on our amorphous computer. To simplify the problem, suppose also that the system has already been initialized so that the elements coloured blue (those comprising the two lines at the top and bottom of Figure 1-3) have already been labeled. We are also given one element on the left from which we would like to produce a pattern of labelings on the elements that has the topology of that in Figure 1-2. How do we think about organizing such a pattern, when we know that the elements have no (explicit) information about their locations, and that for them the concept of "direction" will need to be defined?

For the moment, let us ignore the second problem, and imagine how we would generate the pattern, obeying only the locality constraints. That is, we are allowed to draw a new point or line only if it is next to a point or line already drawn. We might proceed as follows:

1. Draw a short length of red line *parallel* to the given blue lines, which we will refer to here as "rails".

2. At the end of that red line, draw two red lines in opposite directions, *towards* the rails, taking care to make sure not to actually touch them.

3. From the end of each of those new red lines, draw another short red line *parallel* to the rails, and *away from* the original starting point on the left.

4. On each of the parallel red lines, near its end, draw four vertical lines, two in opposite directions towards the rails, and two in opposite directions towards each other. Use yellow for the segments near the top rail and green for the other two near the bottom rail.

5. Where each of the new vertical lines intersects with a rail, place a black rectangle there.

6. Where the yellow and green lines intersect, place a black rectangle there and draw a short red line *away from* the original starting point.

Upon even a cursory examination of the steps outlined, it is clear that the ability to draw a line in an arbitrary direction with a controlled length is a very useful subroutine to have. In fact, the only remaining problem given such a subroutine would be the matter of sequencing the calls to the subroutine so that the lines are drawn according to the plan. However those two steps represent significant issues that need to be addressed in this computational environment:

- What do terms like *parallel, towards* and *away from* mean to an element, for which there is only the concept of immediate connectivity to other elements?

- All the elements have the same program. So how will the ones that comprise a line that we draw be distinguished from those off the line?

- All the elements are executing asynchronously in parallel. So even if they are able to coordinate a line pattern, how do they ensure that those lines are connected appropriately?

A GPL program allows the user to express a pattern in a manner like the steps of the outline above. It hides the underlying issues from the user, so that the user can think in terms of the elements of the pattern being expressed. It can denote only points that have already been drawn or that were given in the initial conditions, thereby enforcing the locality constraint.

A line is readily expressed as the trajectory of a growing point that tries to take the most direct route to its destination. Growing points can cause the execution of instructions at the particles that house them. In this way growing points can invoke other growing points, and perform operations that affect other growing points. These properties can be used as a serializing mechanism to ensure that the patterns we express are connected appropriately. For example, to draw one line from the end of another, we cause the growing point responsible for the first line invoke the second growing point. Growing points can be named and invoked multiple times, so the user can effectively abstract commonalities in the given pattern.

## 1.3 How GPL works

Let us take a close look at the implementation of a simple program to draw a line segment between two points.

```
(define-growing-point (A-to-B-segment)
  (material A-material)
  (size 0)
  (tropism (ortho+ B-pheromone))
  (actions
   (when ((sensing? B-material)
          (terminate))
         (default
           (propagate)))))
```

The definition of a growing point has two parts: the attributes and the instructions. The attributes section describes the type of material (if any) that is deposited, the thickness of the curve produced and the rule for determining the next location for the growing point. In this example, the name of the growing point is `A-to-B-segment`, it deposits a material (i.e. sets a logical marker) called `A-material`, it has zero width (so the trajectory will be as thick as the size of a particle) and it grows towards increasing quantities of a signal called `B-pheromone` (the meaning of positive orthotropism for `B-pheromone`).

The instructions section of a growing point definition contains a sequence of commands that are executed at each location visited by the growing point. These commands control whether the growing point moves or not, whether it secretes any pheromones, and whether other growing points get initiated or not. In the example, the growing point will stop moving if there is any of a material called `B-material` deposited at its current location. Otherwise, it tries to move to some point in the immediate neighbourhood of its current location. Note that while the commands may control whether a growing point moves or not, it is the tropism declaration that controls *how* the growing point moves when it does.

As its name implies, the example is a fragment of GPL code for drawing a directed line segment between two points. If we label two points in the plane $A$ and $B$ (as the name suggests) then we would want to initiate `A-to-B-segment` at the point $A$. However, as it is written, `A-to-B-segment` will need some help to actually produce the desired line segment; there must be some cooperation from the point $B$. Here is some more of the GPL code:

```
(define-growing-point (B-point)
  (material B-material)
  (size 0)
  (for-each-step
   (secrete 10 B-pheromone)))
```

This code fragment defines another growing point named `B-point` which deposits markers called `B-material` and has zero width. Also quite importantly, `B-point` does not have a tropism declaration which means that it is stationary. The body of `B-point` introduces the `secrete` command. Its effect is to distribute a signal called `B-pheromone` radially symmetrically, with monotonic decreasing strength, for up to 10 units away from `B-point`'s current location.

Now we see how `A-to-B-segment` and `B-point` interact: since `B-point` is the source of all `B-pheromone`, any `A-to-B-segment` located within 10 units of `B-point` will grow towards it. Since the growth of `A-to-B-segment` is specified to be positively orthotropic to (ie. "directly towards") `B-pheromone`, and the pattern of distribution of pheromones is radially symmetric, the resulting path of the original active site of `A-to-B-segment` to `B-point` is a straight line.

In general, growing points are allowed to have parameters in their definitions and therefore take arguments when invoked. It so happens that both growing points in our example do not have any parameters, but we will see later how growing point parameters allow active sites to communicate information non locally so that we can produce curves of a particular length, or with varying sizes.

## 1.4 What GPL builds on

The GPL system is actually two systems at different levels. There is the lower level that models the system's particles and their interactions, and there is the higher level which implements the GPL interpreter.

The models for communications at the lower level were founded on models that were originally used for packet radio networks [38]. Our models evolved from explicit simulations of actual protocols like ALOHA (see [47] for a description) to the more abstract, layered model in the end (which was inspired largely by the organizational layers of the ISO OSI model and TCP/IP). Our model for computation is based on an event-driven that is closely coupled with the message-passing communications model [31].

The techniques for implementing secretions are based in part on approximations to simulations of diffusion. Our first implementations arose from our efforts to establish amorphous coordinate systems [13] and later from simulations of the Gray-Scott model of reaction-diffusion [42, 14]. Conversations with Norm Margolus and Raissa D'Souza about diffusion on cellular automata led to a better understanding of the time complexity involved and consequently a different time-accuracy tradeoff in our implementation. For analytical treatments of the problem, we turned to several sources [33, 11, 8], most notably Ahlfor's text [3].

At the higher level, much of our implementation of GPL is reminiscent of the meta-circular evaluator and Scheme compiler presented in Abelson and Sussman's text [1]. Unpublished course notes from the Programming Languages class at MIT, 6.821 [20] also contributed in important ways.

## 1.5 Outline

Chapter 2 is a detailed description of GPL and presents a more elaborate description of the program for expressing the pattern from Figure 1-2. Chapter 3 describes the message passing language used by the particles, and presents the translation process from GPL to that language.

Chapter 4 presents some analysis for GPL. The first part presents a framework for analysing GPL programs in general, and presents some analysis for two common building block growing points, namely rays and line segments. The next part of the chapter demonstrates that any planar graph is expressible in GPL, and discusses the resources involved in realising it. The last part of the chapter presents some empirical data for characterising the dependence of the success of GPL programs on the distribution of particles in the system.

Chapter 5 presents a range of patterns that are expressible in GPL. It also highlights some of the interesting programming paradigms within GPL. Finally, Chapter 6 presents our conclusions and outlines the future directions for this research.

# Chapter 2

# The Growing Point Language

This chapter describes the Growing Point Language in detail. Throughout the presentation, we use code excerpts from a single example pattern to illustrate the particular language feature under discussion. Finally, we try to characterise the types of curves that can be constructed in GPL. Complete code listings for the GPL interpreter are given in appendix C.

The Growing Point Language (GPL) is a programming language for specifying interconnect topologies in a coordinate-free way. The behaviour of a GPL program is determined by its text, an associated domain and a set of initial conditions. The *GPL domain* consists of a set of points, a metric for defining distances between them, and a characteristic distance called the *step-size*. The members of the GPL domain are referred to as *locations* within the language, and will be referred to similarly from now on. The initial conditions are specified as associations of locations (i.e. points from the GPL domain) with program commands. The specification is done in the program text with abstractly denoted locations whose actual values are supplied when the GPL program is evaluated.

## 2.1 Language Overview

Before plunging into the details of GPL, we present a brief overview of the language, and describe the principal concepts at an abstract level.

### 2.1.1 The Concepts

A *growing point* is a locus of activity that describes a path in the GPL domain. At any given moment, a typical growing point is housed at a single location in the GPL domain, called its *active site*. A growing point may "move" in discrete steps in the domain according to its *tropism* by changing its active site from one location to a neighbouring location. At its active site, a growing point may "deposit" material and it may "secrete" pheromones. (Both processes are, in fact, associations of locations in the GPL domain with values to represent either the presence of a material (deposition) or the quantity of a pheromone (secretion).) The tropism of a growing point is specified in terms of differences in pheromone levels in the neighbourhood of the active site. Specifically, the pheromone quantities form a scalar field whose gradient directs the motion of the growing point. The path that the growing point describes is the collection of locations that it has visited. The path is detectable only if the growing point has been specified to deposit material.

In the context of pattern generation, the post-execution presence of deposited material at a location represents that a fragment of the pattern has been established at that location's coordinates in the plane. The correspondence between types of materials and types of pattern fragments is established before the program is executed. In this way a GPL program draws a definite pattern that can be recognized based on only the proximity of common types of material deposits.

The definition of a growing point dictates the topology of its path, however the geometry of a particular path that a growing point produces—being dependent on the geometry of the domain—is determined only after the growing point has been invoked at some location. Invoking a growing point at a location may yield successor locations in the domain. If there are no successor locations, the growing point *terminates* the path at that location; otherwise each successor location in the domain is not farther than *step-size* from the current active site. In most cases, there is at most one successor location to an active site, so the path produced is simply a polygonal path. In cases where there are more than one successor locations, then the "path" is actually a tree in the GPL domain. Whether we have a tree or a polygonal path, each edge connects locations that are neighbours in the GPL domain. The step-size is the fundamental unit distance: it defines a growing point's neighbourhood; it is therefore an upper bound on the size of every growing point's step; and it is also used as the unit by which all distances in GPL are expressed.

### 2.1.2 Primitive Datatypes

There are two types of primitive objects in GPL: pheromones and materials.

**Materials** are used as markers to indicate where growing points have previously visited. The `color` command associates a material type with a colour which is used to highlight locations as they are visited. Materials can also be sensed by growing points so that decisions for growth may be based on what materials have already been deposited at a site. Ultimately, it is in the arrangement of the materials deposited that we will look to find the pattern we intended to generate.

**Pheromones** exist to guide the movement of growing points. Pheromones are produced when a `secrete` (or `secrete+`) command is executed. The extent of the surface to be covered by the pheromone is also specified in the command. The distribution of pheromone quantities is radially symmetric about the source, and monotonic decreasing in the distance from the source. The exact shape of the distribution is deliberately unspecified and should not be depended upon by GPL programs.

### 2.1.3 Execution

As part of their definition, growing points embody commands that must be executed at their active sites. Commands in the body of a growing point's definition are evaluated in sequence by the particle at the growing point's active site. Each command must *complete* before the subsequent command in sequence is executed. For most commands, completion simply means that the operations associated with that command have all been executed successfully. A few commands require communication between the active site and locations near it. For these, completion means that the operations associated with that command have been executed successfully not only at the active site, but also on every location that was involved in the processing of the command. These commands rely on completion messages from the neighbouring locations to the active site. So usually the active site must wait for an indication that the next command may be evaluated.

An active site executes until either it encounters a `halt` or `terminate` command, or it runs out of commands to evaluate. When a new active site is spawned, the original one may not have finished evaluating all of its commands. In this case, both active sites are considered to be processing commands in parallel. The spawning active site pauses for a short period after the new active site has been spawned. There are no commands to explicitly control the rate at which an active site processes commands. So this implicit pause is a mechanism to allow the second active site to begin execution and possibly affect the first active site's subsequent computation.

When a growing point moves, it requires the presence of at least one pheromone to guide its motion. If at such a time there is no relevant pheromone present at the active site, the active site is stalled until that pheromone becomes detectable. If however, the appropriate pheromones are present, but no acceptable location can be found to move the active site to, then the active site retries. After three failures to find an acceptable location, the active site *halts* execution, and the growing point is *stuck*.

## 2.2 Defining a Growing Point

Rather than give a formal definition and syntax for all the features of GPL, we will present small code examples that will exercise many of those features. Growing points are created with the `define-growing-point` command. Here is an example of its use.

```
(define-growing-point (orient-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- dir-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (actions
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

A growing point has a name, `orient-poly` in this case; it may have parameters, `length` is the only one here; and it may have any or all of several attributes, indicated by the keywords: *material*, *size*, *tropism*, *avoids*, *actions*. The name is used for referencing the growing point and is scoped globally since all growing point definitions are at the top-level. Any parameters, on the other hand, are scoped over the definition of the growing point. Parameter values may be either numbers or boolean values.

The *material* attribute designates the tag that is associated with each location that the growing point visits. The value of this tag must be a constant literal. Each location keeps a list of material tags that can be polled by growing points visiting that location. This "detection" of materials allows growing points to be sensitive to the presence of certain materials. It is important to note that a growing point does not deposit its material until after it has executed any commands given in its *actions* attribute. That feature allows a growing point to detect its own material without being encumbered by its own deposition.

The material tags are also used as keys for colour-coding the growing point when the program's pattern is displayed. If the *material* attribute is unspecified, then the growing point does not cause any tag to be associated with the locations it visits; consequently its path cannot be displayed.

The *size* attribute indicates the half-width, in *step-size* units, of the growing point; it can be specified as any integer valued expression. The expression is evaluated once per invocation of the growing point and retains its value for the lifetime of the growing point (i.e. until it terminates). That value is used as the radius of the neighbourhood, in step-size units, with which the material tag will be associated. In the example, `N-WIDTH` is a globally specified constant, which happens to have the value 1. This means that when `orient-poly` is invoked, every point in the 1-hop neighbourhood of its location will be associated with the material tag. From the standpoint of the display, the growing point will produce a curve whose width is twice the step-size. If the *size* attribute is omitted then the growing point assumes the default size of zero. This means that only the location of the growing point will be tagged by the material tag, but not any of its neighbours.

The *tropism* attribute determines the pheromones to which the growing point is sensitive, as well as in what manner. The specification of this attribute is a *tropism expression*. Since tropism expressions have slightly involved semantics, we defer their detailed discussion for now. In this example, the tropism expression indicates that there are three pheromones of interest, their names are: `dir-pheromone`, `vdd-long` and `vss-long`. When the growing point is ready to change locations, the most favoured destination is the one that is furthest from the source of `dir-pheromone` (i.e. *negatively ortho*tropic to `dir-pheromone`) yet also at distances from `vdd-long` and `vss-long` respectively similar to those of the current location (i.e. *dia*tropic to `vdd-long vss-long`). Omitting this attribute implicitly implies that the growing point is stationary. In that case, use of the `propagate` (see description in section 2.2.2) command in the *actions* attribute is forbidden.

The *avoids* attribute indicates which pheromones inhibit the growing point. A growing point will never visit a location where any of the pheromones mentioned in this attribute are present. If this attribute is omitted, then no pheromones inhibit the growing point.

The *actions* attribute describes the sequence of commands executed at each location visited by the growing point. These commands control whether the growing point moves or not, whether it secretes any pheromones, and whether other growing points get invoked. In the example, the growing point will stop moving if the value of the `length` parameter is smaller than 1. Otherwise it attempts to visit another location (determined by the tropism attribute) where the value of the `length` parameter will be one less than its value at the current location. Note that while the commands may control whether a growing point moves or not, it is the tropism declaration that controls *how* it moves when it does. Growing points that are specified without an *actions* attribute implicitly *terminate* themselves when invoked.

### 2.2.1 Tropism Expressions

A tropism describes how a growing point moves. It is always expressed in terms of one or more pheromones. A tropism expression is represented as a pair of a filter and a sorter that given a set of associations from neighbour ids to pheromone concentrations, returns a subset of acceptable associations, ordered from most acceptable to least acceptable. In order for a growing point to change its location, the values (concentrations) for the appropriate pheromones are polled from its location's neighbourhood, and filtered according

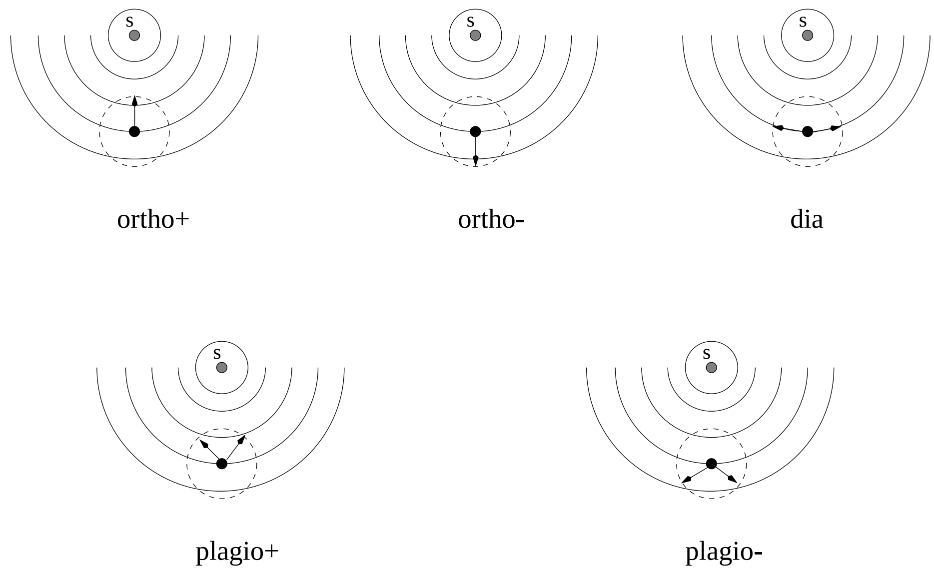


Figure 2-1: The directions, relative to the source of a pheromone, represented by each tropism expression. The particle marked ‘s’ is the source of secretion. The arcs represent contours of the pheromone concentration. The dotted circle represents the neighbourhood of the active site, which is represented as a black dot. The arrows should be interpreted relative to the line joining the source and the active site.

to its tropism. The location associated with the first of the remaining (ordered) values is selected as the growing point’s destination. If there are no values left after filtering, then the growing point is *stuck* and terminates at its current location. Tropism expressions have five primitive forms (*ortho+, ortho-, plagio+, plagio-* and *dia*) and four combining forms (*and, ∼and, or*, and *∼or*). The primitive forms are described below[1]. Figure 2-1 illustrates the directions that each tropism expression represents relative to the pheromone source.

**ortho+** : retain all pheromone concentrations higher than the current value, and give preference in decreasing order of concentration. Rough interpretation: move in the direction of maximal increase of pheromone concentration.

**ortho-** : retain all pheromone concentrations lower than the current value, and give preference in increasing order of concentration. Rough interpretation: move in the direction of maximal decrease of pheromone concentration.

**plagio+** : retain all pheromone concentrations higher than the current value; compute the average of the highest and the current concentration and give preference in increasing order of proximity of the concentrations to that value. Rough interpretation: move in a direction oblique to the direction of maximal increase of pheromone concentration.

**plagio-** : retain all pheromone concentrations lower than the current value; compute the average of the lowest and the current concentration and give preference in increasing

[1]The names for these tropisms was taken from the Encyclopoedia Britannica on line: ”tropism” Britannican Online. ¡http://www.eb.com:180/cgi-bin/g?DocF=micro/604/86.html¿

order of proximity of the concentrations to that value. Rough interpretation: move in a direction oblique to the direction of maximal decrease of pheromone concentration.

**dia** : compute the range of pheromone concentrations and retain all those that differ from the initial concentration by less than a small fraction of the range. Preference is given in increasing order of proximity to the initial concentration. Rough interpretation: move in a direction orthogonal to the direction of maximal increase of pheromone concentration.

The combining forms take arbitrarily many tropism expressions as arguments. They must combine both the filtering and the ordering operations of each of their argument expressions to produce a single filter and a single sorter. The ordering operations are combined in the same way for each combining form. Suppose the sorters obtained from the argument expressions of a combining form are $s_1$ and $s_2$. The combined sorter gives precedence to $s_1$, then uses $s_2$ to order elements whose ordering was indeterminate under $s_1$. The method is best described in terms of the predicates used by these ordering operations. Say `pred1` and `pred2` are the respective predicates corresponding to $s_1$ and $s_2$, then the predicate corresponding to their combined sorter is

```
(lambda (elt1 elt2)
  (or (pred1 elt1 elt2)
      (and (not (pred1 elt2 elt1))
           (pred2 elt1 elt2))))
```

Unlike sorters, filters may be combined in a number of ways, depending on the combining form specified. The compound filter for each combining form is described as follows:

**and** : compose the filters sequentially, using the output of one as the input to the next.

∼**and** : compose the filters sequentially, as in the case of *and*, but only use as many beyond the first as possible that still return a non-empty result.

**or** : apply each filter in turn to the input set and return the first non-empty result.

∼**or** : return the union of the result of each filter applied independently to the input set.

### 2.2.2 Growing Point Commands

All GPL commands begin with a keyword. It may be followed by appropriate arguments depending on the command. A command sequence is simply a list of commands, no special keyword is required to indicate one. Here is a description of all the commands[2] that may be used within the actions attribute. Recall that these commands are executed only when a growing point is invoked at a location, therefore it always makes sense to talk about the current location in the context of their explanations.

`(start-gp` < *gp-name* > < $arg_1$ > `...)` Invoke the growing point named by *gp-name* at the current location. The given arguments are supplied as the initial values of the parameters declared in the definition of *gp-name*.

[2]Actually there are three others, but their descriptions have been deferred until later.

`(propagate` $< arg_1 >$ `...)` Find a new location for the growing point, propagate all its dynamic state including the new values for the parameters given by the values of $arg_i$, and invoke the command sequence of the actions attribute at the new location.

`(terminate)` Terminate the growing point at the current location.

`(secrete` $<$ *range* $>$ $<$ *pheromone* $>$ `)` Secrete enough of the pheromone named *pheromone* so that its concentration is monotonic decreasing from the source and zero at a distance *range* from the current location. The concentration at the current location depends on the value of *range*. The pheromone concentration recorded at each affected site is taken to be the larger of its current value for *pheromone* and the quantity arriving there due to this secretion.

`(secrete+` $<$ *range* $>$ $<$ *pheromone* $>$ `)` Secrete enough of the pheromone named *pheromone* so that its concentration is monotonic decreasing from the source and zero at a distance *range* from the current location. The concentration at the current location depends on the value of *range*. The pheromone concentration recorded at each affected site is taken to be the sum of its current value for *pheromone* and the quantity arriving there due to this secretion.

`(when` $< clause_1 >$ `...(default` $<$ *default-seq* $>$ `))` Each clause consists of an expression followed by a command or a command sequence. Each clause's expression is evaluated in turn until one of them evaluates to a non-false value. When that happens, the associated command or command sequence within the containing clause is executed. The special expression `default` always evaluates to *true*. Upon completion of the execution of a command sequence of a clause, execution returns to the point after the `when` command.

### 2.2.3 Growing Point Expressions

All expressions must evaluate to either numbers or boolean values. They may be:

- literal numeric constants
- variables such as parameter names, growing point names and globally declared constants.
- arithmetic expressions combining any of the arithmetic operators `+`,`-`,`*`,`/` with numerically valued expressions.
- logical comparisons using one of the arithmetic comparators `<`, `=`, `>`.
- `(sensing?` *material-name* `)`. This expression returns *true* if the tag *material-name* has previously been associated with the current location.

## 2.3 Abstractions and Methods of Combination

The commands presented so far are sufficient to render any given topology in the GPL domain. However, as yet there are no ways of capturing common patterns in growing point definitions, thereby making it cumbersome to manually program patterns with even a moderate number of lines. This section presents commands that significantly improve the expressive power of GPL. These commands come in two categories: the first allows growing point definitions to be reused by and incorporated into other growing points; the second allows collections of growing points to be treated as a single entity. Both categories provide GPL code modularity and therefore facilitate manually implementing non-trivial patterns.

### 2.3.1 Linking growing points

With the commands presented so far, two growing points can be sequenced only by starting the second explicitly within the actions attribute of the first. This property forces compound growing points to be monolithic in the sense that their sub-pieces may not be reused in any other contexts. To alleviate that annoyance, we provide the `connect` command. The `connect` command allows growing points to be sequenced even though they are defined independently. The termination location of the first growing point becomes the location at which the second growing point is invoked. As an example, suppose we have three growing points `A`, `B` and `C` which we would like to combine in two different ways. Namely, we would like to produce one curve generated by `A` followed by `B` and another generated by `A` followed by `C`. Until we had the `connect` command, we would have had to repeat the definition of `A` because it was used in two different contexts. Specifically, now we may write `(connect (start-gp A) (start-gp B))` and `(connect (start-gp A) (start-gp C))` to produce both of those curves without having to define `A` for each instance of its use.

The `connect` command provides a powerful technique for composing primitive growing points to produce new ones. To be a little more concrete, here are two code fragments that demonstrate the use of the `connect` command in that context.

```
(define-growing-point (poly-with-inertia length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- poly-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))

(define-growing-point (horiz-poly length)
  (material poly)
  (size N-WIDTH)
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (connect (start-growing-point orient-poly 2)
            (start-growing-point poly-with-inertia (- length 2)))))
```

It is not too important, for now, to understand what path `poly-with-inertia` describes. Rather, the emphasis is on the manner in which `horiz-poly` is defined as a composition of `orient-poly` and `poly-with-inertia`. The purpose of `orient-poly` was to produce a path that grew in a certain predetermined direction, as dictated by the distribution of `dir-pheromone`. On the other hand, `poly-with-inertia` produces a path that continues to grow in the same direction as it was started. Combining these two growing points into the single entity `horiz-poly` allows us to produce a path that is oriented by `dir-pheromone` but does not depend on its presence throughout the entire growth process.

### 2.3.2 Aggregating growing points

Growing points need not be composed together in order for us to think of them collectively. Logical groupings of growing points are also provided for within GPL. Such groupings are called *networks*.

As mentioned earlier, a single growing point describes a path or a tree in the GPL domain. With the `start-gp` command, we can cause one growing point to invoke another, so in fact we can produce superpositions of trees that share at least one node. For patterns that contain more than one connected component, there is no way to capture the whole pattern with a single growing point, since each growing point produces only connected components. A description of any such pattern must supply at least as many locations in the initial conditions as there are connected components. This means that we have to look at the entire program to understand the pattern that will be produced. Our growing point abstractions allowed us to represent commonalities within the connected parts of our pattern, but not across connected components. Therefore if a pattern is composed from sub-patterns of more than one connected component, our program will be unable to properly reflect this structure across connected components. The network abstraction solves this problem.

A growing point can be viewed as a relation between the location of its invocation and the location(s) of its termination. A network is an extension of this idea: it defines a relation between two sets of locations called the *inputs* and the *outputs*. Inputs represent the locations where growing points may be invoked and outputs represent the subset of the termination locations of those growing points that we choose to advertise outside of the network.

Networks
are defined with the `define-network` command. Like the `define-growing-point` command, `define-network` may appear only at the top level of the program. Consider the problem of drawing two parallel lines. In principle, this could be achieved by two growing points. The first produces one line and secretes a pheromone to which the other growing point grows diatropically. (Note that we must invoke these two growing points at distantly separated locations, so the pattern of parallel lines is a good candidate for our network abstraction.) In the following code excerpt, two short parallel lines are produced using the general idea of the method just described (the code is mindful of a few technical details).

```
(define-network (rails-1 (vdd-in vss-in) (vdd-out vss-out))
  (at vdd-in
      (connect (start-growing-point vdd-rail-1 UNIT-LEN)
               (connect (start-growing-point vss-starter)
                        (connect (start-growing-point vss-rail-1 UNIT-LEN)
                                 (->output vss-out)))
               (->output vdd-out)))
  (at vss-in
      (start-growing-point vss-starter-beacon)))
```

The network is named `rails-1`, and it has two inputs called `vdd-in` and `vss-in` and two outputs called `vdd-out` and `vss-out`. The growing points `vdd-rail-1` and `vss-rail-1` generate the lines we see in the pattern. `UNIT-LEN` is a globally declared constant, and is used here to express the lengths of the lines drawn. The other two growing points, `vss-starter` and `vss-starter-beacon` will be discussed shortly.

Our method for drawing parallel lines does not treat the two lines equivalently. The second line depends on a secretion from the first in order to grow; in a sense it is *dependent* on the first. In order to ensure that the first is given a chance to grow before the second begins, we employ the following serialization technique. At the location at which we intend to invoke the second growing point, secrete a long range pheromone that acts as an attractor for a growing point that will be invoked at the end of the first line. This serialization is the function of the two growing points `vss-starter` and `vss-starter-beacon`.

The body of `rails-1` completely describes the relations between the inputs and the outputs in terms of growing points. The `at` command allows us to specify the sequence of commands to perform at each input. The `->output` form allows us to indicate which growing points terminate at which outputs. In this example, we see that we invoke `vdd-rail-1` at the input labelled `vdd-in` and `vss-starter-beacon` at the input labelled `vss-in`. The termination location of `vdd-rail-1` is labelled as `vdd-out`. The other network output `vss-out` is obtained from the termination location of `vss-rail-1`.

Under ideal conditions using the described method of drawing a pair of parallel lines, we could allow the first line to grow independently for as long as it needs to, and the second to follow all the way. However if we have an irregularly spaced domain, and the second growing point encounters an especially sparse region in the domain, it is likely to get stuck. By growing short segments at a time, alternating the independence of the lines, we hope to produce a more robust pair of parallel lines. Therefore, we define another network called `rails-2`, similar to `rails-1`, that reverses the roles of `vdd-rail` and `vss-rail`. The true value of the network abstraction is highlighted when we cascade the two networks to produce a single network that can itself be cascaded to produce arbitrarily long parallel lines. Here are the code fragments to demonstrate that:

```
(define-network (rails-2 (vdd-in vss-in) (vdd-out vss-out))
  (at vdd-in
      (start-growing-point vdd-starter-beacon))
  (at vss-in
      (connect (start-growing-point vss-rail-2 UNIT-LEN)
               (connect (start-growing-point vdd-starter)
                        (connect (start-growing-point vdd-rail-2 UNIT-LEN)
                                 (->output vdd-out)))
               (->output vss-out))))
(define-network (rails (vdd-in vss-in) (vdd-out vss-out))
  (==> (vdd-in vss-in) rails-1 rails-2 (vdd-out vss-out)))
```

The definition of `rails-2` is identical in form to `rails-1` and so needs no further explanation. The definition of `rails` indicates that it is a network with two inputs and two outputs, both sets named as they were for `rails-1` and `rails-2`. Since both input and output identifiers are scoped over the definition body of the network, that fact has no bearing on the semantics of `rails`. The body of `rails` contains a new symbol: `==>`. It should be read as "cascade". In the example, the `rails` network is defined by connecting its two inputs `vdd-in` and `vss-in` (in that order) to the inputs of the `rails-1` network. The outputs of `rails-1` are wired to the inputs of `rails-2` and finally the outputs of `rails-2` are wired to outputs `vdd-out` and `vss-out` of `rails`. So in a completely natural way, the cascade network command allows us to compose networks to obtain "compound" networks.

### 2.3.3 Network Commands

Like the growing point commands, network commands begin with a keyword that identifies the command. Network commands may be used only in the body of a `define-network` command. Multiple network commands in the body of a network definition, are interpreted as the superposition of their individual effects. In addition to the network commands, there are also a few new growing point commands that are associated with networks. These commands may be used in the same contexts as the previously presented growing point commands, but we deferred their presentation until now when networks have already been presented. Here are the network commands and their descriptions:

`(at` < *input* > < *gp-command-seq* > `)` Execute *gp-command-seq* at the location denoted by *input*.

`(==>` < *inputs* > < *net* > `...` < *outputs* > `)` The networks given, $net_i$, are evaluated so that the outputs of one serve as inputs of the next. The locations denoted in *inputs* serve as inputs to the first network given, and the last network labels its outputs with the list of location names provided in *outputs*.

`(let-locs ((`< *loc-names* > `)` < *loc-defn* > `)` < *body* > `)` Create new location identifiers as given in *loc-names*. The network command sequence, *loc-defn*, is first evaluated to obtain the locations that are denoted by the names in *loc-names*. Then *body* (also a network command sequence) is evaluated within the scope of this new binding.

The new growing point command is `->output`. It has the syntax `(->output` < *loc-name* > `)` and is used to associate actual locations with names. Since all locations, except for those provided as initial conditions, are obtained by propagation of growing points, it was natural for the assignment of locations to be done implicitly through a command. In this way, GPL escapes the need to explicitly denote locations, thus no coordinate systems are required.

### 2.3.4 Auxiliary Commands

At this point we have presented only commands that define the behaviour of our growing points, but we have not yet described how the a program is executed, and how to control the output. For these special operations, we need a few new commands. All of these commands may be used only at the top level. They are as follows:

`(color (` < *material list* > < *colour* > `) ...)` Associate a colour with each material combination that we anticipate needs to be observed. The specified colour is given as a string containing a valid X colour name. This command is really just a simulator directive for rendering the patterns that the program produces.

`(constant` < *name* > < *value* > `)` Declare a global constant called *name* to have the numerical value *value*. Constants may be shadowed statically by growing point parameters.

`(include` < *files* > `)` Include the GPL programs contained in *files* in the current file at the location of this `include` command.

| Keyword | Alias |
|---|---|
| `define-growing-point` | `define-gp` |
| `start-growing-point` | `start-gp` |
| `cascade` | `==>` |
| `connect` | `-->` |

Table 2.1: A list of aliases for GPL keywords

`(with-initial-locs` < *loc-names* > < *body* > `)` Define the entry point of the program. This command is very much like a network definition. The specification of *loc-names* is a list of names to denote locations, exactly like the *inputs* specification of an ordinary network definition. The *body* of this command follows the same syntax as the body of the `define-network` command. It also serves the same function, namely to describe the actions that take place at each input location given. In the case of this command, the input locations given are the initial conditions for the entire program. If multiple `with-initial-locs` are given, the last one is the only one used.

By far, the most interesting command in this section is the `with-initial-locs` command. It is part of the interface between GPL and the user. When the user evaluates a GPL program, he supplies a list of locations from the GPL domain that correspond to each denoted location in the `with-initial-locs` command. That is the only time when locations are ever explicitly named. These names may be coordinates or some other naming system that can be used to uniquely identify the points in the GPL domain. When a GPL program is executed with its supplied initial locations, the interpreter translates those named locations into elements of the GPL domain, and associates the location names from the `with-initial-locs` command with those GPL domain elements. In this way, GPL programs never actually manipulate the point names supplied by the user. Therefore, GPL programs may denote only those locations supplied as initial conditions, or neighbours of denotable locations.

At this point we have presented example GPL code fragments and introduced all of the important GPL constructs. Figures 2-2 and 2-3 present a summary of the syntax for all the commands, separated into growing point commands and network commands respectively.

### 2.3.5 Aliases

There are a few shorthand versions of the keywords presented. They are listed in Table 2.1

## 2.4 The Execution Model

GPL is essentially a parallel language in that command sequences are executed simultaneously at multiple locations. We model this parallelism on a single processor by maintaining an agenda for the locations that are currently executing commands. Whenever a growing point is activated at a location, that location is added to the agenda. It remains on the agenda until the commands of the growing point's actions attribute have all been executed.

If each location were computed independently of every other location, then the timing of the execution of the command sequences would be irrelevant. However locations can induce

```
(define-growing-point ( gp-name [ parameter ...])
  [(material  material)]
  [(size  numeric-exp)]
  [(tropism  tropism-exp)]
  [(avoids  pheromone ...)]
  [(actions  gp-command-seq)]
  )
 gp-name =  id
 parameter =  id
 pheromone =  id
 material =  id
 tropism-exp = (ortho+  pheromone) |
                    (ortho-  pheromone) |
                    (plagio+  pheromone) |
                    (plagio-  pheromone) |
                    (dia  pheromone) |
                    (and  tropism-exp ...) |
                    ( and  tropism-exp ...) |
                    (or  tropism-exp ...) |
                    ( or  tropism-exp ...)
 numeric-exp = L_N  |  id |
                    (+  numeric-exp  numeric-exp ...) |
                    (-  numeric-exp  numeric-exp ...) |
                    (*  numeric-exp  numeric-exp ...) |
                    (/  numeric-exp  numeric-exp ...)
 gp-command = (start-gp  gp-name [ numeric-exp [...]]) |
                   (propagate [ numeric-exp [...]]) |
                   (terminate) |
                   (secrete  numeric-exp  pheromone) |
                   (secrete+  range  pheromone) |
                   (when  clause ...) |
                   (connect  gp-command  gp-command-seq) |
                   (->output  net-terminal)
 gp-command-seq =  gp-command |
                       ( gp-command  gp-command-seq)
```

Figure 2-2: Syntax of growing point level commands

```
 net-name =  id
 net-terminal =  id
 net-terminals =  nil |
                       net-terminal .  net-terminals
(define-network ( net-name  net-terminals  net-terminals)
   net-command-seq)
(with-initial-locs  net-terminals
   net-command-seq)
 net-command = (at  net-terminal  gp-command-seq) |
                    (==>  net-terminals  net-name ... net-terminals) |
                    (let-locs ( net-terminals  net-command-seq
                               [ net-terminals  net-command-seq]
                               ...)
                       net-command-seq)
 net-command-seq =  net-command |
                        ( net-command  net-command-seq)
```

Figure 2-3: The syntax of network level GPL commands computations at neighbouring locations, and so we have to be careful about how the computations at those neighbouring locations are orchestrated. Observe from the command list that there is only one command that actually induces the execution of command sequences at neighbouring locations, that is the `propagate` command. Secretions also affect nearby locations, but only to update the state of the recorded pheromone concentrations, not to cause the execution of GPL commands. Secretions are also presumed to happen much faster than the time to complete the transactions between locations involved in evaluating a `propagate` command. Therefore, to process each location on the agenda, each command is evaluated in turn until either there are no more commands, or we encounter a `propagate` command. If there are no more commands, the next location on the agenda is processed. If a `propagate` command is encountered, the successor location is determined, based on the growing point's tropism, and placed on the agenda. The current location is replaced on the agenda in such a way that when it is next removed, it will resume execution at the command following the `propagate` command.

### 2.4.1 Justification for the Execution Model

The `propagate` command was designed so that a single growing point could have the flexibility of describing both branching structures and polygonal paths. The way this is accomplished is by multiple uses of `propagate`. For example, if we want to produce a tree with branch factor 3, we can accomplish this with a single growing point that is defined as follows:

```
(define-growing-point (tree init-life life)
  (material plant)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< init-life 1)
          (terminate))
         ((< life 1)
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3)))
         (default
           (propagate init-life (- life 1))))))
```

There are many interesting aspects of the code fragment given above, but for now we focus on the `when` command and its three clauses. Each clause represents one of the three things that can happen in the development of the tree. The tree may either:

1. stop growing,
2. branch to produce three sub-trees or
3. continue growing.

These three options are clearly reflected in the structure of the code fragment, therefore there is great metaphorical value in being able to express the growing point structures in this way. However, in order for the behaviour of this growing point to be what we would like it to be, the successor location resulting from one propagate must be given a chance to execute its commands before the current location is allowed to proceed with the next command. In a parallel execution, that circumstance would be quite natural, especially since we have presumed the `propagate` command to be slower than all other commands. However, with a single-threaded evaluation we have to orchestrate this, hence the execution model presented.

## 2.5 Examples: Simple CMOS Circuit Layouts

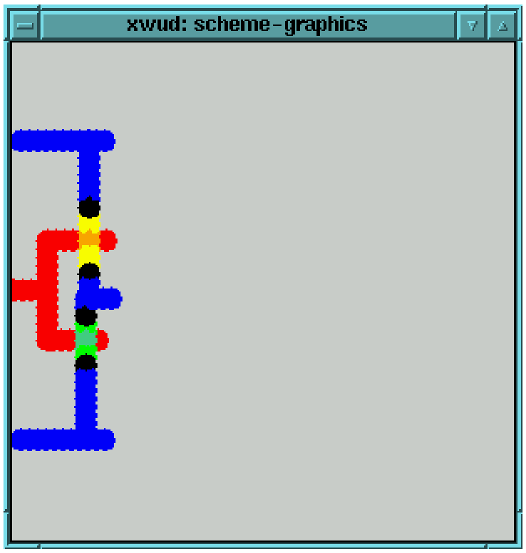


Figure 2-4: The CMOS Layout of an inverter

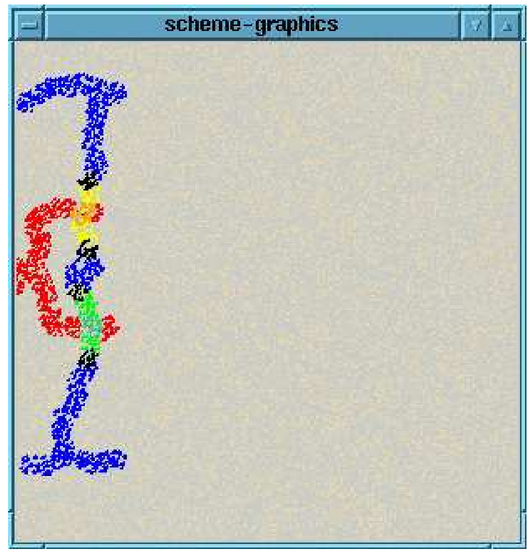


Figure 2-5: The CMOS Layout of an inverter on a randomly arranged domain

Now that the language has been presented piecewise, let us try expressing an example pattern that will exercise many of the commands we have seen. Let us suppose that we want to draw the CMOS layout of an inverter as shown in Figure 2-4. Incidentally, for the discussion in this chapter, whenever we illustrate the effect of a GPL code fragment, we shall show two pictures that both result from evaluating the same GPL code. The picture on the left is generated on a domain where the locations lie on a lattice, and we use the Euclidean metric to measure distances. These assumptions represent the ideal case of the domain, and are useful for debugging GPL programs in general. The first step of expressing the pattern given in Figure 2-4 in GPL is to identify lines that can be specified with as few inter-dependencies as possible (i.e. can be described in GPL with as few references to pheromones or materials produced by other lines). Then we define the remaining lines to run parallel, perpendicular or obliquely to those already defined.

Upon inspection of the layout, we observe that if we were able to specify the rails (blue lines at the top and bottom of the layout), we could then specify all the other lines as being either parallel or perpendicular to those lines. Let us imagine for the moment that we could somehow produce the rails[3], and focus on how we could specify the remaining lines. It is worthwhile mentioning here that we recognize, from the context of our target pattern, that

[3]The reader may have noticed that their definition was used as the example code for the discussion on networks

partitioning the rails from the rest of the lines is likely also to be useful for generating other circuit layouts.

### 2.5.1 The Primitive Pieces

Observe that there are three occurences of a red, horizontal line segment, so we shall begin with trying to write a definition of a growing point to produce such a line segment. In the context of CMOS layouts, the material of those line segments is known as polysilicon. Since the line is parallel to the rails, we can arrange for the rails to secrete long range pheromones, and then try to make this red line segment grow diatropically to those pheromones. The astute reader will realise that this simplicity is misleading: there are *two* directions that correspond to the diatropic growth with respect to the rails. If we assume that we start at one end of the circuit (say the left) and proceed to the other end, then we will need to somehow restrict those two choices to one during the production of these horizontal lines. To accomplish that we simply add another tropism expression to the tropism attribute: one that is negatively orthotropic to the place where we start (i.e. away from it). Here is the definition corresponding to the preceding discussion:

```
(define-growing-point (orient-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- dir-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (actions
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

This definition is a good start, but since growing points will stall when a relevant pheromone is absent from the current location, we see that we will need to secrete `dir-pheromone` for a considerable distance to allow the upper and lower horizontal polysilicon segments to grow. In order to make our growing point a little more flexible in its interaction with `dir-pheromone` we can express it as a combination of two pieces, each piece itself a growing point. The strategy is to define one piece almost exactly as `orient-poly` but use it only to get the poly segment started. We then use the second piece to continue the segment to whatever length we require, but so that it does not require the presence of dir-pheromone to grow. This second piece is essentially a ray, which can be achieved by defining a growing point that grows with negative orthotropism to a pheromone that it secretes. By using `secrete+` so that the secretions superpose, we can make sure that the growing point propagates in the same direction on each step. Here is the definition of the second piece:

```
(define-growing-point (poly-with-inertia length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- poly-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (actions
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

In the definition of `poly-with-inertia` we use the pheromone `poly-pheromone` to serve the propelling function described. We include the `vdd-long` and `vss-long` tropisms to make the definition robust, so that small initial displacements from parallel to the rails are not magnified unchecked. In order to get this growing point to grow in the direction that was started by `orient-poly` we add a command to secrete `poly-pheromone` in the actions attribute of `orient-poly`. Finally to put the two pieces together, we define `horiz-poly` as follows.

```
(define-growing-point (horiz-poly length)
  (material poly)
  (size N-WIDTH)
  (actions
   (connect (start-growing-point orient-poly 2)
            (start-growing-point poly-with-inertia (- length 2)))))
```

The vertical lines of Figure 2-4 are simpler to describe: they are orthotropic to the rail pheromones (positive or negative, depending on which rail we use as a reference). Here is the GPL code for each type:

```
(define-growing-point (up-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (or (ortho+ vdd-long) (ortho- vss-long)))
  (for-each-step
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))

(define-growing-point (down-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (ortho+ vss-long))
  (for-each-step
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

The diffusion lines are also defined similarly, except, of course, they deposit the appropriate type of diffusion material. Sometimes the vertical lines need to terminate when they intersect with the rails. For example, that is the case with the diffusion lines in the figure.

In that case we would not define our growing point with a length parameter, rather the growing point would terminate whenever it sensed the metal of a rail. The definition of this version of a vertical line is different enough that it is worth presenting explicitly. Here we use the example of the p-diffusion segment that connects the upper segment of horizontal polysilicon to the $V_{dd}$ rail.

```
(define-growing-point (p-diff->vdd)
  (material p-diffusion)
  (size P-WIDTH)
  (tropism ( and (ortho+ vdd-long) (ortho- dir-pheromone)))
  (for-each-step
   (when ((sensing? metal)
          (terminate))
         (default
           (propagate)))))
```

Technically, we need only the `(ortho+ vdd-long)` tropism expression, but we include a weak conjunction with the `(ortho- dir-pheromone)` to encourage left/right deviations to occur in only one direction.

### 2.5.2 Building Bigger Abstractions

At this point, we have what is necessary to construct all the growing point definitions required to produce the pattern between the rails. However, keeping in mind that we may want to produce more general circuits, we capture the patterns of both types of transistors as GPL networks. Here is the definition for the p-type transistor. It assumes that the rail pheromones will be present, but it does not assume what its source and drain will be connected to. We also defined `vert-n-fet` in an analogous way.

```
(define-network (vert-p-fet (gate) (src drain))
  (at gate
      (start-growing-point horiz-poly 2)
      (connect (start-growing-point up-p-diff FET-HEIGHT)
               (->output src))
      (connect (start-growing-point down-p-diff FET-HEIGHT)
               (->output drain))))
```

#### Defining the rails

We have now defined everything necessary for the inverter of Figure 2-4 except the rails. But rails are nothing more than parallel lines. So we can simply use the implementation from the discussion in § 2.3.2. Of course, there are alternate strategies to generating parallel lines. For example, we could produce a centre line and have it secrete a pheromone, to which the two rails would grow diatropically. This strategy has the advantage that the range of the secretion of the guiding pheromone is half of what it was in the first strategy. On the other hand, if we intended to reduce the range of the long range pheromones, then in particular, we would not have the pheromones `vdd-long` and `vss-long` extending all the way between the rails. We could probably get away with this as long as every location between the rails was covered by at least one of those two pheromones. However, it would mean that we would have to redefine all our other growing points to rely on only one of those pheromones, and not both as is currently the case.

### Putting it all together

We have now identified and defined the important pieces in the pattern of Figure 2-4. Now consider how the parts of the pattern are assembled. We see that if we begin from the input on the left of the figure, and we want to use our transistor abstractions, we are forced to manipulate two dynamically generated locations simultaneously: namely, the points where the transistors should start growing. So we use a `let-locs` command to denote those two locations, and we pass them to another network, `inverter-fets`, that connects the transistors as they ought to be for an inverter.

```
(define-network (inverter (poly-in) (poly-out))
  (let-locs
   ((p-fet n-fet)
    (at poly-in
        (--> (start-growing-point horiz-poly POLY-LEN)
             (secrete+ (* 2 POLY-LEN) dir-pheromone)
             (--> (start-growing-point up-poly
                                       (+ INV-HEIGHT N-WIDTH))
                  (--> (start-growing-point horiz-poly
                                            (+ POLY-LEN (* 2 N-WIDTH)))
                       (->output p-fet)))
             (--> (start-growing-point down-poly
                                       (+ INV-HEIGHT N-WIDTH))
                  (--> (start-growing-point horiz-poly
                                            (+ POLY-LEN (* 2 N-WIDTH)))
                       (->output n-fet))))))
   (==> (p-fet n-fet) inverter-fets (poly-out))))
```

The special symbol `-->` is used as a shorthand for the `connect` command. All the capitalised identifiers are global constants that were defined elsewhere. It should be noted that while we may specify distances in our GPL program, there is no guarantee that our growing points will actually travel a physical distance proportional to the values specified. The values specify the number of times a growing point's instructions will be executed, the actual distance it travels in the GPL domain will depend heavily on the distribution of points in the domain. This discrepancy between values and distance means that for a given domain, we sometimes have to tweak both the initial conditions and the distance specifications of our programs to obtain the layout we aimed for. Experience has shown that a good approach to writing GPL programs is to first debug the program with a regular GPL domain, and then fine-tune it on the target domain by adjusting distances and the initial conditions if possible.

The network `inverter-fets` applies the `vert-p-fet` and `vert-n-fet` networks at the respective input locations. The `src` output of the `vert-p-fet` network is connected via a metal depositing growing point to the `vdd-rail`. The `drain` outputs of both FET networks are connected via two metal-depositing growing points. The one starting from the `drain` output of the `vert-n-fet` network secretes an attracting pheromone for the other, and grows a small length that terminates in the output of the inverter. We have omitted the code listing for `inverter-fets` here since no new concepts are introduced in it; a complete code listing for the inverter and all the GPL programs presented can be found in Appendix A.

The final step in putting all the pieces together is to superpose the rails with the `inverter` network. We define this composite network, `inverter+rails`, essentially as the superposition of the rails with the `inverter` network. There is one detail that must be considered however: for robustness we ought to ensure that the rails are drawn before the `inverter` network is. The language already provides an automatic ordering scheme by stalling a growing point until all the pheromones upon which it depends are present. That means that none of the `inverter` network will actually get drawn until the `rails` network has begun and the pheromone secretions have reached the initial location of the `inverter` network. Under ideal conditions, this automatic blocking of growing points would be sufficient to produce the correct output because the growing points will all grow at the same rate, so the initial polysilicon growth would always be behind the rail metal growth. However, on a random layout, we have no guarantee that the rate of progress of one growing point will be equal to that of another growing point at a different location in the GPL domain. So to improve the robustness of our code, we implement a serializer with growing points.

```
(define-network (inverter+rails (vdd-in vss-in poly-in)
                                (vdd-out vss-out poly-out))
  (let-locs ((vdd-tmp vss-tmp)
             (==> (vdd-in vss-in) rails (vdd-tmp vss-tmp)))
    (let-locs ((ser-poly-in)
               (==> (vdd-tmp poly-in) input-serializer (ser-poly-in)))
      (at vdd-tmp (->output vdd-out))
      (at vss-tmp (->output vss-out))
      (==> (ser-poly-in) inverter (poly-out)))))
```

If we would like for there to be a definite precedence of events at two (usually non-neighbouring) locations, then we need to explicitly enforce that. GPL treats command execution at different locations as happening simultaneously. The idea is a relatively simple one: at the first location, instead of starting the usual operation, secrete a pheromone far enough that it can reach the other location. (That means that serialization only works over a limited range, but if we can characterise that distance, that limitation is usually not a real problem.) At the second location, we perform whatever operations need to precede those at the first, and then initiate a growing point that is attracted to the pheromone secreted by the first location. This growing point does not deposit any material, and terminates when it encounters the source of the pheromone (by sensing for a special material). Upon termination, the invisible growing point initiates the execution of the operations intended to be at the first location. The preceding operations are implemented by the `input-serializer` network. It takes as input locations, the two locations that need to be serialized, the first location is that of the earlier executing commands. It produces one output location, which will be the same as second input location, but by the time it would have been "produced" as a result, it would be safe to execute whatever instructions are supposed to be executed there.

Figure 2-6 shows a sequence of snapshots of the inverter as it was grown during the execution of the GPL program described. There were five initial locations supplied: two for each rail, and one for the inverter input. The form of the `with-initial-locs` command best describes the relationship between the given initial locations and the operations that are performed at each location.

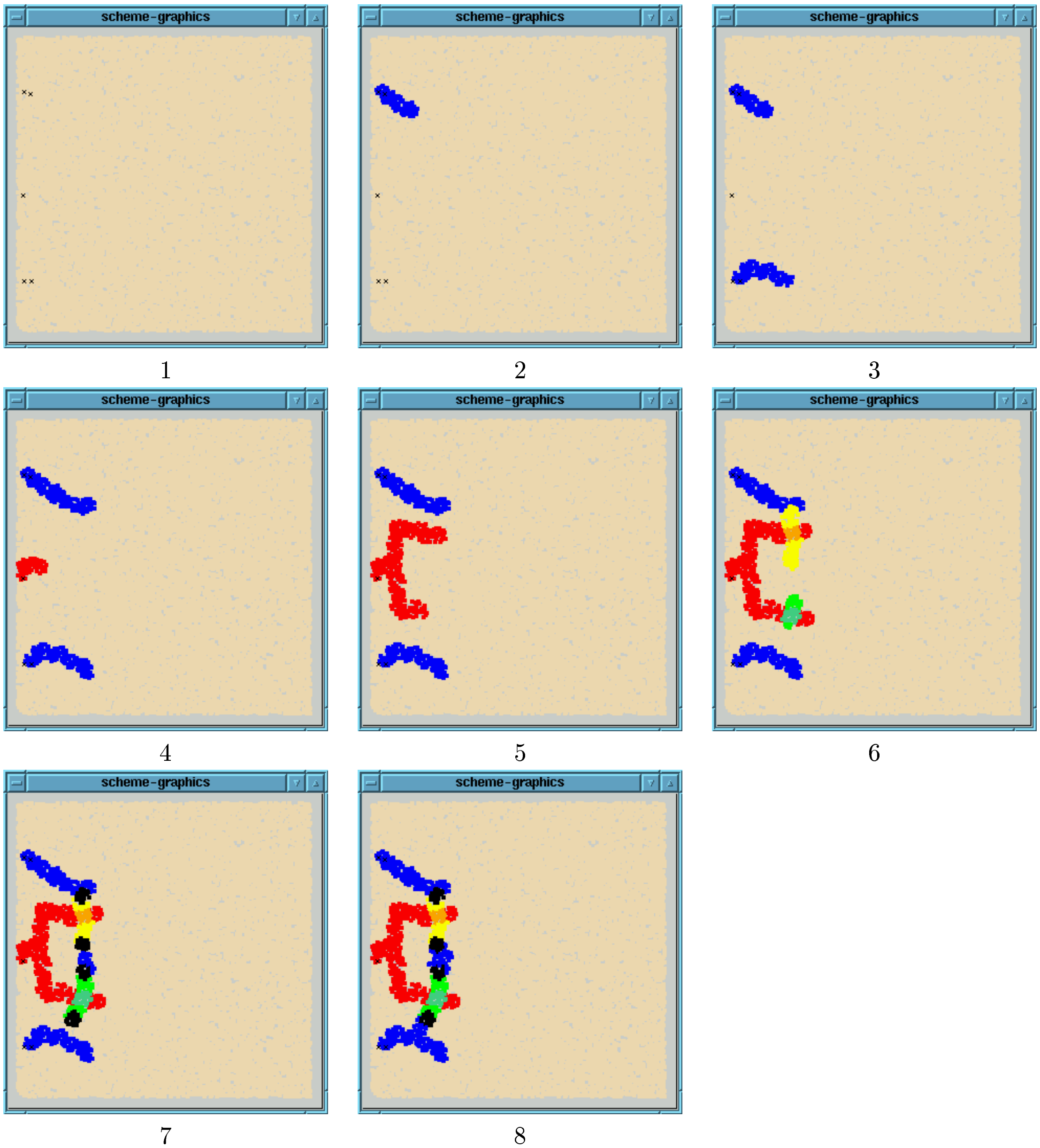


Figure 2-6: The Steps in the Formation of the Layout of a CMOS Inverter

```
(with-initial-locs (vdd-1 vdd-2 vss-1 vss-2 input)
  (==> (vdd-1 vss-1) init-rails ())
  (at input (secrete+ (* 3 POLY-LEN) dir-pheromone))
  (==> (vdd-2 vss-2 input) inverter+rails
       (vdd-ignore vss-ignore out-ignore)))
```

Finally, to illustrate that the robustness measures we discussed have paid off, Figure 2-5 shows the resulting pattern on a randomly arranged domain.

### 2.5.3 Generating Bigger Circuits

At this point, one might complain that the example circuit was too simple, that there are features of larger circuits that do not arise with the inverter, but need to be dealt with. To address this concern, we present a slightly more complicated circuit layout in this section. It is the layout of a two-input NAND gate, as shown in Figure 2-7. We chose this example because it is not too complex for exposition, but it captures some of the important issues that need to be dealt with when drawing layouts with multiple inputs. Specifically, we must be careful when we design growing points that have restrictions on which other growing points' paths they are allowed to intersect.

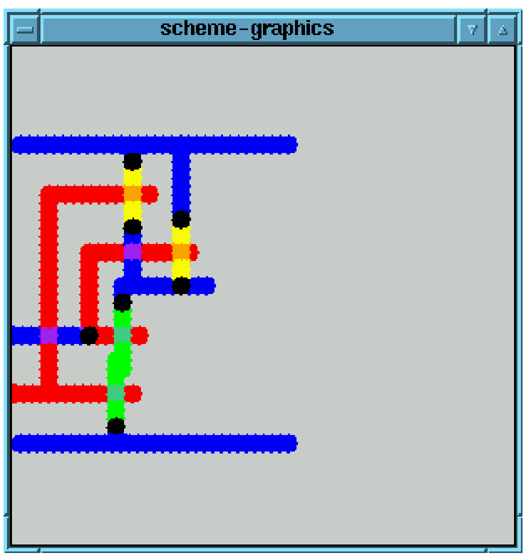


Figure 2-7: The CMOS Layout of a NAND gate

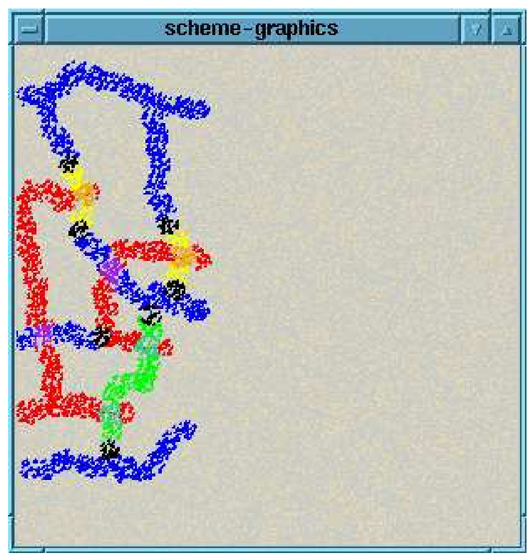


Figure 2-8: The CMOS Layout of an NAND gate on a randomly arranged domain

The *avoids* attribute is useful for keeping growing points away from each other, but it introduces a new problem, one that never had to be considered for the inverter. The *avoids* attribute indicates which pheromones a growing point dislikes. To exploit that so that two growing points repel each other, we cause each of them to secrete pheromones that the other avoids. This means that a growing point cannot avoid the same pheromone that it secretes: it would inhibit itself. The problem that arises is that now growing points that would otherwise be defined in the same way, will now need a new definition for each set of pheromones that they avoid.

The number of definitions that we need is determined by the number of lines that could have been defined by the same growing point, but which are not allowed to cross each other in our target layout. In principle, the number of different versions needed could be high since it is the chromaticity of a graph that encodes the crossing restrictions for the lines in the pattern. However, in practice, lines that are forbidden from crossing are usually in the same plane, and growing points that are identical except for their *avoids* attribute, are unlikely to deliberately intersect, except perhaps at their termination points. That means that we can usually implement just two versions of a growing point and alternate among them for adjacent lines so that each pair of adjacent growing points repel each other.

In the definition of the NAND gate, this issue arose because there are two inputs that have vertical segments of polysilicon, which should not touch each other. In this case, we simply use a different definition for each segment. The definitions of the polysilicon segments for the NAND gate are the same as for the inverter, except that they have an *avoids* clause, and an additional secretion to identify themselves. The two types of polysilicon defined for the NAND gate are complementary in the sense that the pheromone that one growing point secretes is the same that the other avoids. It is probably worthwhile mentioning here that adding *avoids* attributes to growing points are essentially robustness enhancement measures. They are not critical to the growth of the growing point, they simply improve our chances of achieving our target topology. That means that they need not be used to explicitly repel every pair of growing points that we do not want to touch each other. For instance, we can rely a little on the distances specified in GPL if those distances are sufficiently far apart. Moderation is called for in any practical situation. Figure 2-8 shows the results obtained when we evaluated the NAND code on a randomly arranged GPL domain.

Having defined both the NAND gate and the inverter, we would like to believe that we could now define an AND gate in terms of those two circuits. This turns out to be true. The definition of an AND gate is specified simply as the cascade of the NAND gate and the inverter: no additional growing point definitions are needed.

```
(define-network (via+rails (vdd-in vss-in input) (vdd-out vss-out output))
  (==> (input) via (output))
  (at vdd-in (->output vdd-out))
  (at vss-in (->output vss-out)))

(define-network (and+rails (vdd-in vss-in a b) (vdd-out vss-out output))
  (==> (vdd-in vss-in a b) nand+rails via+rails inverter+rails
       (vdd-out vss-out output)))
```

Since neither the NAND circuit nor the inverter assumed a contact at their inputs or outputs, we must include one when we compose the two circuits into an AND gate. We already defined a 1-input, 1-output network called `via` to produce such a contact. Here, the network `via+rails` promotes `via` to a 3-input, 3-output network so that it is compatible with the 3-output `nand+rails` and the 3-input `inverter+rails`.

Of course, the fact that the description of the AND gate is so straightforward was anticipated by the care with which the NAND gate and the inverter were implemented. The real message here is that GPL is expressive enough to describe a rich set of patterns, and it has the added feature that GPL programs resemble the patterns they generate. Figures 2-9 and 2-10 show the resulting patterns when the GPL program for the AND gate was executed on a regular square grid, and a randomised arrangement respectively.

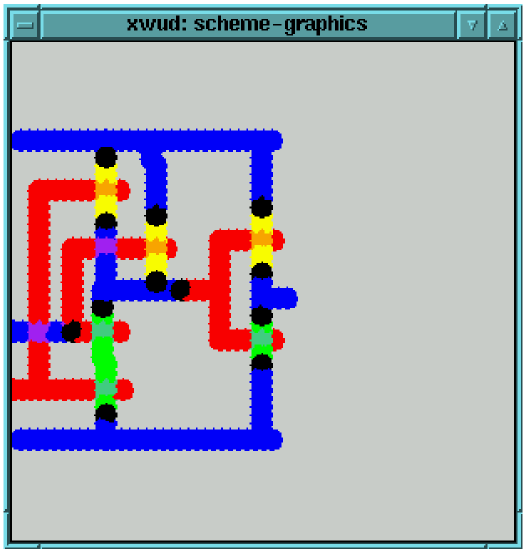


Figure 2-9: The CMOS Layout of an AND gate

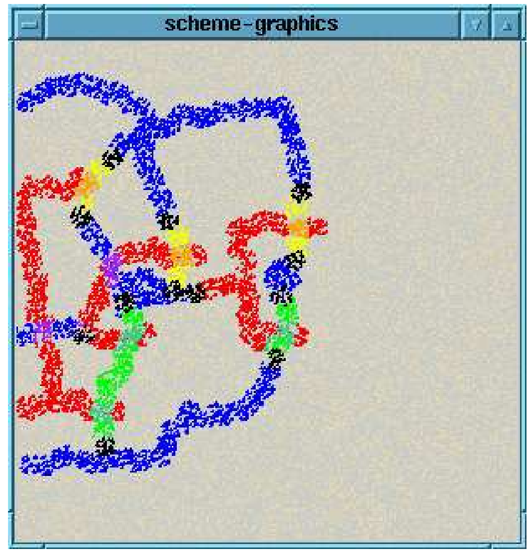


Figure 2-10: The CMOS Layout of an AND gate on a randomly arranged domain

## 2.6 Comments

There were many design decisions involved with the implementation of GPL. Some of them were made for aesthetic reasons, some for pragmatic reasons, and others for reasons that are hard to express. Below we discuss a few of the choices made, and what some of the alternatives are.

### 2.6.1 Expressing Distances in GPL

The Growing Point Language assumes a fundamental distance as part of its semantic definition. That distance defines the largest distance any two consecutive sites of a growing point are allowed to be. It also reflects the number of options for propagation there will be at a given site, since it determines the area of a site's neighbourhood.

The syntactic manifestation of that fundamental distance has at least two possibilities. One is to treat that distance as a unit, and express all other distances (such as extent of pheromone secretions) in terms of it. The other is to allow the unit distance to be set as part of the initial conditions. Although the second formulation allows patterns to be scaled very easily, it involves slightly more complicated initial conditions, it also does not obviate the need for specifying a fundamental distance, it only hides it: neighbouring points in a growing point will still have a particular length characteristic. The presentation of GPL made here opted for the first option in the interest of simplifying the descriptions of all the relevant features of the language.

### 2.6.2 Limited Tropisms

At the moment, GPL provides five primitive types of tropism and allows conjunctions and disjuntions of them. The set of tropism expressions was deliberately limited in this way, in order to reduce "feature creep" and to keep the characterization of tropisms simple. However, there are many other possible tropisms that will fit into the filter/sorter model of tropisms. For example, permitting tropisms that performed arithmetic computations on the pheromone levels is no more difficult to implement than the tropisms already implemented. The choice to forbid those types of tropisms was made in an attempt at minimizing the system requirements for an implementation of GPL. The current tropisms require only the ability to compare pheromone levels, the ability to perform arbitrary arithmetic operations on them would require far more computational power.

Variable tropisms are also readily implemented under the current implementation framework, but were avoided again for the sake of minimality. All that would be required is that the tropism expression be passed on along with all the other information that is passed during the evaluation of a `propagate` command. It seems that they would not add significant power to the expressibility of GPL though, since if the set of possible tropisms for a growing point were finite, one could in principle define as many growing points as there were tropisms. Then the `connect` command could be used to sequence the growing points in the same order that the tropisms would have been sequenced.

# Chapter 3

# GPL: A Particle's Perspective

In this chapter, we describe how the Growing Point Language is interpreted by each particle in an Amorphous Computer. First we present a computational model centred around particle-particle interaction. This model serves as the formalism for expressing the computations that an amorphous computing particle performs. It also makes explicit what we mean when we say "computation by local interactions".

The second part of the chapter is devoted to describing how GPL commands are interpreted in this model. The goal of the chapter is to illuminate how a GPL program, which regards the amorphous computer as a single programmable entity, can be translated into independently executing, distributed code fragments.

## 3.1 A Language of Local Rules

The Extensible Calculus Of Local Interactions (ECOLI ) is intended to be a language for programming an Amorphous Computing computational particle. Currently, ECOLI is more of a programming discipline applied to Scheme, than it is a language. Nevertheless, it is still useful in describing algorithms for a single particle.

In ECOLI , it is assumed that communication among particles is asynchronous and happens in parallel with their computations. The low-level details of the inter-particle communication are not specified by ECOLI , but it does assume that messages take a constant amount of time to deliver and at most one message can be received in one message-delay time interval. This assumption imposes a total ordering on all of a particle's incoming messages, and reflects the serial nature of the particle's processing abilities. Messages in ECOLI are interpreted as commands to be executed on the receiving processor. However the exact meaning of a given message can be changed during the execution of a program, and so it may differ from particle to particle; therein lies a particle's (behavioural) differentiability. Even though the particles are considered to be serial processors, it is often useful to decompose problems in terms of several, loosely coupled, concurrent tasks. ECOLI accomodates this type of problem decomposition by allowing a programmer to specify groups of instructions, called *actions*, that are executed atomically in some unspecified order.

### 3.1.1 The Model for a Computational Particle

There are three components of the processor model that may be directly affected by ECOLI expressions:

- The working **environment** maintains bindings for variables that were assigned or referenced by messages.
- The working **dictionary** maintains a set of bindings from message types to handlers.
- The **current agenda** maintains a list of *actions* that have been started, but have not yet been terminated. An action is a sequence of instructions that is atomically executed.

A fourth component of the model is a message queue, currently modeled as an infinite FIFO queue, whose primary function is to serialise the arriving messages and buffer them until they can be processed. The queue is distinguished from the other three components in that it cannot be modified directly by ECOLI expressions (except when the processor resets itself causing the queue to be flushed). In this way, the state of a processor's message queue at any given time represents the interactions of that processor with its neighbours. The fact that the queue is infinite means that incoming messages are never lost, and simplifies the formal description of the semantics of ECOLI . Clearly, in any physical implementation the queue would be finite, however this is not likely to be a real hindrance. Under the reasonable assumption that the computational throughput is higher than the message bandwidth, for most situations a queue size that is linear in the average number of neighbours will be sufficient.

Here is a list of ECOLI primitives. In general, both handlers and actions are implemented using these primitives as well as primitives from the extending language (Scheme in this case).

`(make-handler args body)`
Create a handler that takes arguments according to `args` and whose behaviour is specified by `body`.

`(def-response name handler)`
Bind `name` to `handler` in the working dictionary.

`(push-response` *dict template body*`)`
Bind a new handler as is done for `def-response`, however, shadow any previously bound handler instead of replacing it. When this new handler is invoked, it is removed from the dictionary, and the binding for the message type reverts to the previous handler.

`(ignore msg-type1 .  other-msg-types)`
Remove the binding for all the supplied message types from the working dictionary.

`(make-action exp1 .  rest-exps)`
Create an action from the sequence of expressions given.

`(action.start action)`
Add `action` to the current agenda.

`(delayed-perform` *delay exp* `...)`
Defer executing *exp-seq* until *delay* cycles from present. Does not return any useful value.

`(repeat)`
Terminate the current action, but do not remove it from the current agenda.

`(get-var id)`
Lookup `id` in the working environment.

`(set-var id exp)`
Cause `id` to be bound to the value of `exp` in the working environment.

`(ensure-binding id exp)`
If `id` is not bound in the working environment, bind it to the value of `exp`.

`(send msg-type . args)`
Broadcast a message of type `msg-type` and contents `args` to all neighbours.

`(delayed-send` *delay message*`)`
Defer sending *message* for *delay* cycles.

`(send-to-self` *message*`)`
Dispatch *message* as if it had been retrieved from the incoming queue.

### Execution

Execution proceeds by first removing a message from the queue and retrieving the handler for its type from the working dictionary. If there is no binding in the dictionary, the message is ignored; otherwise, the rest of the message is passed to the handler as arguments. The procedure `eval-msg`, extracted from the definition file of ECOLI formally expresses the lookup operation.

```
(define (eval-msg message dictionary)
  (if (pair? message)
      (let ((handler (get dictionary (car message))))
        (if handler
            (apply handler (cdr message))
            #f))
      (error "Improper message format: " message)))
```

After the handler is applied, the current agenda is executed which entails evaluating each statement in each member action of the current agenda. The ordering of actions within an agenda is not specified, however the instructions within an action are performed sequentially. Unless the last statement executed within an action was a `(repeat)` statement, the action is removed from the current agenda. At the end of agenda execution, another message is removed from the queue to restart the entire evaluation process.

These steps are implemented by the procedure `driver-loop`:

```
(define (driver-loop terminal? stop)
  (let loop ()
    (if (not (msg-queue/empty? *incoming-msg-q*))
        (eval-msg (msg-queue/remove! *incoming-msg-q*) *the-dictionary*))
    (agenda.exec *current-agenda*)
    (if (terminal? *incoming-msg-q* *the-dictionary* *current-agenda*)
        (stop *incoming-msg-q* *the-dictionary* *current-agenda*)
        (loop))))
```

### 3.1.2 A Simple Example

To further demonstrate the model of ECOLI, a simple example program that implements a "count-up-wave" will now be presented. A count-up-wave is simply a labelling of connected processors such that each processor records its best approximation to the hop count distance from one distinguished processor, called the source.

The first step is to identify and initialize any process state variables. The working environment is used to store variables that other processors may ask the value of. It is also used to store values obtained from other processors. In this example, it will be used only to store up-to-date information (ie. the distances from the source) from neighbouring processors.

```
;;;  Set up initial variable bindings
(ensure-binding 'myid (random 65536))   ; choose 16-bit id
(set-var 'mydistance #f)
```

The most interesting parts of an ECOLI program involve determining the bindings in the working dictionary and the conditions that will trigger the desired actions. It is usually at this point that interesting spatial properties of the ensemble that arise from the execution of a program manifest themselves in the program's text. In this simple example, the fact that a *distance*-typed message is *self-triggering* (ie. it can cause another message of its own type to be sent) indicates that *distance*-typed messages can propagate for as far as connected processors extend.

```
;;;  Install a handler for  distance messages in the working dictionary.
(def-response 'distance
  (make-handler (sender-id new-distance)
    (cond ((or (not (get-var 'mydistance))
                (< new-distance (get-var 'mydistance)))
           (set-var 'mydistance new-distance)
           (color-me (get-var 'mydistance))
           (send 'distance (get-var 'myid) (+ new-distance 1)))
          (else 'ignore))))
;;;  Declare the initial triggering action
(define source-action
  (make-action
    (set-var 'mydistance 0)
    (color-me (get-var 'mydistance))
    (send 'distance (get-var 'myid) 1)))
```

Finally, there has to be a way to get everything started. The procedure `run` resets the ECOLI system and then calls the driver-loop. It takes as arguments: an initial agenda, an action to perform just before termination, and optionally the maximum time (measured in clock ticks) to run for.

```
;;; Start the ECOLI engine with the appropriate agenda
(run
 (if (read-sensor 'special?)           ;interface with environment
     (make-agenda source-action)
     (make-empty-agenda))
 (make-action
   (color-me (get-var 'mydistance))))
```

```
(define (make-distribution-process name next-key next-val
                                   key= combiner store? propagate?)
  ;; combiner is a left combiner
  ;; (propagate? key val old-val . args) decides whether to continue
  (ensure-binding 'distribution-table (make-table 'distribution-results))
  (let ((key-assoc (g-association-procedure key= entry/tag))
        (dist-tbl (get-var 'distribution-table)))
    (make-process
     name
     (make-dictionary
      (('DIST key val . args)
       (let ((entry (key-assoc key (table/entries dist-tbl))))
         (if (not entry)
             (begin
               (if (apply propagate? key val #f args)
                   (apply send name 'DIST
                          (next-key key) (next-val key val)
                          args))
               (if (apply store? key val #f args)
                   (put dist-tbl key val)))
             (let ((stored-val (entry/value entry)))
               (let ((result (combiner stored-val val)))
                 (if (apply propagate? key result stored-val args)
                     (apply send name 'DIST
                            (next-key key) (next-val key result)
                            args))
                 (if (apply store? key result stored-val args)
                     (entry/replace-value! entry result)))))))
      (('ACTIVATE init-key init-val . args)
       (put (get-var 'distribution-table) init-key init-val)
       (apply send name 'DIST
              (next-key init-key) (next-val init-key init-val)
              args))))))
```

Figure 3-1: Definition of a distribution *process*. This process is somewhat like a list accumulation in Scheme.

### 3.1.3 Abstractions

The model presented up to this point does not have any way of abstracting over the kinds of processes we might imagine implementing as building blocks. For example, there is no way to take advantage of the count-up-wave program to produce a program that labelled processors with decreasing numbers, as their hop count from the source increased. Since the two programs differ only in the way they propagate values, it is quite reasonable to hope that such a facility existed within the language.

At the moment, there are only rudimentary means of abstraction and combination. In particular, one means of abstraction is the notion of a *process* which encapsulates the behaviour induced by a group of handlers bound in the dictionary. Processes are named and can be produced as results of procedures, so ordinary Scheme procedures can be used to make higher-level abstractions over processes. The process name is used to identify the bindings placed in the working dictionary with the process.

Listing 3-1 shows an example of the use of the process abstraction. The process described there is called a distribution process and is somewhat like accumulation over lists in Scheme. The general idea is that each processor maintains a value locally, which is computed from information given to it by its neighbours. On receipt of each message, the processor decides

```
(define count-up-wave
  (let ((propagate? (lambda (key new-dist old-dist)
                      (or (not old-dist)
                          (< new-dist old-dist)))))
    (make-distribution-process
     'count-up
     (lambda (key) key)                 ;  key transformer
     (lambda (key dist) (+ dist 1))     ;  value transformer
     (lambda (k1 k2) (eq? k1 k2))       ;  key comparator
     (lambda (stored-val new-val)       ;  combiner
       (let ((result (min stored-val new-val)))
         (color-me result)
         result))
     propagate?                         ;  store?
     propagate?)                        ;  propagate?
    ))
;;; Declare the initial triggering action
(define source-action
  (make-action
    (set-var 'mydistance 0)
    (color-me (get-var 'mydistance))
    (process.start count-up-wave (get-var 'myid) 1)))
(run
 (if (read-sensor 'special?)
     (make-agenda source-action)
     (make-empty-agenda))
 (make-action
  (color-me (get-var 'mydistance))))
```

Figure 3-2: Count-up waves implemented using the distribution abstraction whether to update its current value, and whether to propagate any information to its neighbours. The distribution process can be used to implement either count-up or count-down waves, and even a model for diffusion. Listing 3-2 shows how the count-up-wave code, previously presented, is recast as a distribution process.

Two ways in which processes might be combined are to run them in parallel, or to have one trigger the activation of the other. The model naturally facilitates the execution of two processes in parallel, since that is equivalent to ensuring that the two processes use disjoint message types. Sequencing two processes is accomplished by creating a special binding in the working dictionary with a handler that activates the second process. By ensuring that the message type for the binding is generated only when the first process is completed, the two processes can be guaranteed to sequence correctly.

## 3.2 Translating GPL to ECOLI

We are now at the stage where we can carefully describe what a GPL program looks like from the perspective of a single Amorphous Computing particle. We shall describe the processes that each particle must be able to execute when a GPL program is to be evaluated on a collection of them. These processes can be divided into two categories: those that depend upon the cooperation of neighbouring particles, and those that do not. The first category is implemented on top of a universal protocol that is independent of the specific GPL program to be interpreted. It turns out that the only GPL primitives that give rise to processes in this category are `propagate` and `secrete`. Consequently, we shall devote considerable effort to explaining the interpretation of these two primitives. In contrast, we will spend little or no time on the other primitives because they reduce to single-processor computations with which we are already very familiar. In our exposition, we use two levels of description: one for each category of processes.

The first level of description is concrete, in the sense that we present actual ECOLI code fragments that result from translations of the cooperation-dependent GPL primitives. The resulting ECOLI program is executable by the HLSIM Amorphous Computing simulator [2]. HLSIM pays attention to all the details involved in a particle simultaneously handling local computation and communication. This aspect of it is particularly important for the translation of pheromone secretion and growing point propagation. The HLSIM simulator gives each particle in the Amorphous Computer a time slice within which it executes a portion of its program (written in ECOLI ). The particles are given similar clock frequencies, but have randomly determined relative phases. Although a few test runs were made on simulations that used faulty communications, most of the development, and all of the results presented were produced assuming error-free communications. (We address some of the issues that will have to be taken care of when communications are faulty in Chapter 6.) HLSIM strictly enforces the locality constraints of an Amorphous Computer, therefore a correctly behaving ECOLI translation of a GPL program is strong evidence for the soundness of our ideas.

Because the translation of GPL programs to ECOLI is highly detailed, the resulting programs can be unwieldly. So for the exposition of the GPL commands that involve only computation on a single particle, we switch to a higher level of description. This second level of description is presented in terms of a GPL interpreter, called the *GPL illustrator*. The GPL illustrator guarantees that each command on a single particle is completed before the next one in sequence is evaluated. For the `propagate` and `secrete` commands, that policy of the illustrator only approximates what actually happens in the HLSIM simulator. In the

HLSIM implementation, handshaking between particles is used to force an active particle to wait for the cooperation-dependent commands to terminate. For example, in a secretion, the source particle waits for a special signal before proceeding with the subsequent command. That signal indicates that the pheromone values at locations within range of the secretion's extent have settled. Under poor communications conditions, it would be possible for the source particle to be deceived into proceeding prematurely because of delayed completion signals. Therefore, the GPL illustrator is not an exact model of the HLSIM implementation, but under normal operating conditions it is a very good approximation.

The GPL illustrator maintains an agenda of "active" particles. Particles are considered active if they are currently evaluating the body of some growing point. Since fewer particles get a time-slice, simulations run faster with the GPL illustrator than with the HLSIM implementation. The GPL illustrator glosses over some of the communication details involved with implementing secretions and propagation, but it ensures that any information that a particle uses could have been available under the standard Amorphous Computing assumptions described in Chapter 1 § 1.1. The GPL illustrator is a useful tool for exposition, and it is also a convenient test bed for exploring new language extensions and abstractions, such as networks.

## 3.3 The Representations of the Primitive Datatypes

We now describe the data structures that are used to represent the key aspects of GPL. By presenting the concrete representations of pheromones, materials, tropisms, and growing points, we hope to make the subsequent discussion on the processes that support GPL easier for the reader to follow.

### 3.3.1 Pheromones

Pheromones are represented as an association of the pheromone name and a number representing its concentration level. Each particle maintains a table of all the pheromones that it has ever detected.

### 3.3.2 Materials

The materials at a particle's location are represented as a set of the material names (symbols). When a particle's location becomes the active site for a growing point, the growing point's material is added to the set after all the instructions in the body have been evaluated. The colour used to display a particle is determined from the composition of its set of materials.

### 3.3.3 Growing points

There are two classes of properties associated with growing points that we care about. The first class is the static information that can be determined from the syntax of a growing point's definition. The second class is the dynamic information that is generated when a growing point is invoked and may evolve as the growing point's active site moves from particle to particle.

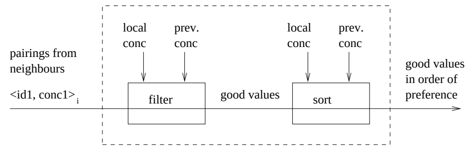


Figure 3-3: Tropism expressions are "compiled" into configurable filters and sorters whose behaviour depends on the run-time values of the pheromone levels at and around the growing point's active site.

### Static Information

The static information for a growing point is represented as a vector of attributes: its name, parameters, material, tropism, poisons, and body. For each growing point, these attributes are shared across all the particles in the GPL domain because they are derived from the GPL program.

### Dynamic Information

For each instance of a growing point invocation, the parameter values, current size, previous pheromone result and the current (growing point) continuation are all part of the dynamic information that particles pass around.

## 3.4 Translating Tropisms

Approximately, each tropism is implemented as a combination of a filtering process and a sorting process (see Figure 3-3). A tropism expression is actually translated into a procedure that takes two continuations and returns a filter generator. A filter generator is a procedure that takes the pheromone values at the predecessor's and the current location's site, and returns a procedure that actually does the filtering and the sorting of its given list of pheromone values. Here is the top level code for translating a tropism expression:

```
(define (analyse-tropism tropism-exp targets)
  (let ((filter-gen-maker (tropism->filter tropism-exp targets)))
    (let ((sort-gen (pred-gen->sorter-gen
                     (tropism->pred-gen tropism-exp targets))))
      (lambda (empty-case non-empty-case)
        (lambda (my-result last-result)
          (let ((sorter (sort-gen my-result last-result)))
            ((filter-gen-maker
              empty-case
              (lambda (results)
                (non-empty-case (sorter results))))
             my-result last-result)))))))
```

Each primitive tropism expression is implemented as a composition of an appropriate filter and sorter. The procedures `tropism->filter` and `tropism->pred-gen` do most of the work involved in the translation from tropism expression to filter and to sorter. The parameters `empty-case` and `non-empty-case` are continuations that determine the behaviour of the filter depending on whether the result of filtering is an empty list. These continuations permit composition of filters and sorter predicates to represent compound tropism expressions. The next layer of parameters, `my-result` and `last-result` contain information about the pheromones at the particle's location and those at the growing point's previous site, respectively. These parameters are used to define the predicates that characterise the filter and the sorter.

### 3.4.1 Filtering

The procedure `tropism->filter` dispatches on the tropism expression to create a filter. In the case that the expression is a primitive tropism, the procedure `D:make-primitive-filter` is called on the same arguments as were given to `tropism->filter`. There are two of these arguments: the expression (`exp`), and a mutable list of pheromone names discovered so far (`targets`). Assuming the expression was legal, `D:make-primitive-filter` returns the primitive filter corresponding to the tropism expression and updates `targets` to contain any new pheromone names discovered. As seen below, `D:make-primitive-filter` is also a simple dispatch on the tropism expression.

```
(define (D:make-primitive-filter exp targets)
  (let ((index (get-index (cadr exp) targets)))
    (case (car exp)
      ((ortho+)
       (make-extreme-filter > index))
      ((ortho-)
       (make-extreme-filter < index))
      ((plagio+)
       (make-filter > index))
      ((plagio-)
       (make-filter < index))
      ((dia)
       (make-eqv-filter index))
      (else (error "Unknown primitive tropism")))))
```

Let us now consider a specific example so that we can discuss the conversion process in more detail. In the case of the `ortho-` expression, the real work gets done by the `make-extreme-filter` procedure, presented below. We first compute an index to represent the pheromone name in the current expression. (A primitive tropism expression has the pheromone name as its second element.) The procedure `get-index` searches the list of known pheromones, called `targets`, to see if the pheromone has already been assigned an index. If not, a new index is used, and `targets` is updated.

Since pheromone values are monotonic decreasing in the distance from the source of secretion, negative orthotropism to a pheromone translates to finding a neighbour with a smaller pheromone value than the value at the current active site. That statement is implemented by passing the less-than predicate and the pheromone's index to `make-extreme-filter`. The other filter makers are similar in structure to `make-extreme-filter` but differ in function. (See Appendix C for their implementations.)

```
(define (make-extreme-filter comparator index)
  (define ref (make-ref index))
  (lambda (empty-case non-empty-case)
    (lambda (my-result last-result)
      (let ((1-predicate (make-1-predicate comparator index my-result)))
        (lambda (result-list)
          (let ((good-results (g-filter 1-predicate result-list)))
            (if (null? good-results)
                (empty-case)
                (non-empty-case good-results))))))))
```

In this case, the pheromone values in `my-result` are used as references to build a predicate of one argument that returns true when its argument pheromone values are less than those given by `my-result`. That predicate is then used to filter the incoming list of neighbouring pheromone values in the call to `g-filter`[1]. If after filtering, there are any elements in `good-results`, then the `non-empty-case` continuation is invoked; otherwise the `empty-case` continuation is invoked instead.

### 3.4.2 Sorting

The sorter is built in two stages. First an appropriate predicate is generated, and then it is passed as an argument to a stable sort procedure. (A stable sorter is one that leaves equivalent objects in the same order as given in the original list.) Here is the code for generating a sorting predicate from a tropism expression.

```
(define (tropism->pred-gen exp targets)
  ;; returns a result of the form:
  ;; (lambda (my-result last-result)
  ;;   (lambda (result-list) ... ))
  (if (not (pair? exp))
      (error "Bad TROPISM syntax: " exp)
      (let ((op (car exp)))
        (case op
          ((ortho+ plagio+)
           (make-pred-gen > (get-index (cadr exp) targets)))
          ((ortho- plagio-)
           (make-pred-gen < (get-index (cadr exp) targets)))
          ((dia)
           (make-eqv-gen (get-index (cadr exp) targets)))
          (else
           (cond ((null? (cdr exp))
                  (error "Ill formed tropism" exp))
                 ((null? (cddr exp))
                  (tropism->pred-gen (cadr exp) targets))
                 (else
                  (compose-pred-gen
                   (tropism->pred-gen (cadr exp) targets)
                   (tropism->pred-gen (cons op (cddr exp))
                                      targets)))))))))
```

Notice that for compound tropisms, the sorter-predicate generator is indifferent to the method of composition of the primitive tropism expressions. Sorting predicates are always

[1] The behaviour of `g-filter` is exactly that of `list-transform-positive` as described in the MIT Scheme manual [22]

composed in the same way: precedence is given to the predicates in left to right order, and subsequent sorting predicates are used only to break ties reported by the preceding predicates.

Continuing with the `ortho-` example, the result of our translation is a call to `make-pred-gen` with the same arguments that we passed to `make-extreme-filter` when we were constructing a filter. Since we are looking for small pheromone values, we pass the less-than predicate to `make-pred-gen` so that neighbour pheromone values will eventually be arranged in ascending order. The implementation of `make-pred-gen` is fairly straightforward.

```
(define (make-pred-gen comparator index)
  (define ref (make-ref index))
  (let ((predicate (make-2-predicate comparator ref)))
    (lambda (my-result last-result)
      predicate)))
```

The index is used to build a referencing procedure that will extract the appropriate pheromone value out of the set of pheromone values that each neighbour returns. The procedure `make-2-predicate` simply composes the given comparator (less-than in this case) with the referencing procedure. It happens that in the case of `ortho-` the result of `make-2-predicate` is the predicate that we eventually want, but we cannot simply return it straight away; we must first wrap it in a procedure of the appropriate type. In general, the predicate we form may need to refer to the values in `my-result` or in `last-result`, as is the case in the `dia` tropism.

Once a sorting predicate has been generated, the final step, as described in the body of the top-level procedure `analyse-tropism`, is to produce a procedure that will actually do the sorting. Its implementation is also straightforward:

```
(define (pred-gen->sorter-gen pred-gen)
  (lambda (my-result last-result)
    (let ((2-predicate (pred-gen my-result last-result)))
      (lambda (result-list)
        (g-sort result-list 2-predicate)))))
```

So in the end, if we peel away all the layers of procedures returned in the process of translating the `ortho-` tropism, we obtain a procedure that simply sorts its given list in ascending order of the appropriate field.

As mentioned earlier, the reason for the rather involved layers of procedures is that they facilitate composition for both filters and sorters. This implementation of tropisms is quite general and can represent complicated directives such as "take the value of pheromone mumble[2] into account only if it does not render the list of acceptable neighbours empty." The continuations allow for specifiable complex control flow as the values in the list are being processed. The current and previous pheromone values are necessary to construct some filtering and sorting predicates. Since the predicates are simply procedures of one or two arguments, it is easy to compose predicates to yield other predicates. The system is therefore general enough to accommodate many possible tropism forms—many more than have actually been implemented.

[2]insert your favourite pheromone name here

To demonstrate the power of the representation, we now discuss the implementation of the tropism `and` form. First we show how the sorting predicates are combined.

```
(define (compose-pred-gen gen1 gen2)
  (lambda (my-result last-result)
    (let ((pred1 (gen1 my-result last-result))
          (pred2 (gen2 my-result last-result)))
      (lambda (elt1 elt2)
        (or (pred1 elt1 elt2)
            (and (not (pred1 elt2 elt1))
                 (pred2 elt1 elt2)))))))
```

We mentioned briefly that the sorting predicates are taken into account in left to right order, and that we only care about predicates that appear late in the ordering if the earlier ones considered elements of the list equivalent. The definition of `compose-pred-gen` above is the precise implementation of that policy. In general the two predicates given, need only be partial orderings. So if the first predicate returns *false* for two elements in both possible orderings, then we conclude that they have no ordering with respect to the first predicate and we resort to the second. The compound predicate formed in this way, is itself composable in the same way with a third predicate, and so our method of composition generalises to arbitrarily many predicates.

It happens that the method of combination for sorters is the same for all the compound tropisms implemented. That decision was arbitrary, and in principle, one could implement alternative methods of composition of sorters, as simply as `compose-pred-gen` was generated.

Now we turn to the filter part of the compound tropism representation. When we use the `and` form, we want all the filters from the argument tropisms to cascade, one after the other. Here is the implementation:

```
(define (make-and-filter clauses targets)
  (if (null? clauses)
      (error "No clauses in TROPISM (and) form: " exp)
      (let ((filters (map (lambda (clause)
                            (tropism->filter clause targets))
                          clauses)))
        (let loop ((my-filters filters))
          (if (null? (cdr my-filters))
              (car my-filters)
              (let ((rest-filter (loop (cdr my-filters))))
                (lambda (empty-case non-empty-case)
                  (lambda (my-result last-result)
                    (let ((cont ((rest-filter
                                  empty-case non-empty-case)
                                 my-result last-result)))
                      (((car my-filters) empty-case cont)
                       my-result last-result))))))))))
```

First, each tropism clause is transformed to a filter-generator-generator (misnamed a *filter* in the code), and the set of them is maintained in the list `filters`. (Although it is misleading to refer to the filter-generator-generators as filters, we will persist in doing so for the remainder of this section in the interest of word economy, with many apologies to the astute reader.)

We can best understand the loop inductively. If there is only one filter in `filters` then that filter is the result we want (this corresponds to the degenerate case when only one tropism was given to the **and** combining form). In the case that there are more than one filter in `filters`, we compute the result of combining all the filters after the first and call the result `rest-filter`. (This is where we make use of our inductive hypothesis that the code is correct, also known as "wishful thinking".) Assuming that `rest-filter` is actually correct, constructing the composition of all the filters reduces to composing two filters.

To compose two filters, we simply apply the first to the given list, and then if the list is not empty, we apply the second. This is the notion that is captured by the code beginning with the binding for `cont`. Its essence is contained in the expression `(((car my-filters) empty-case cont) ...)`. The reason that the code looks more complicated than its explanation is the unfortunate differences in shape (of the procedure types) between the continuations and the filters that take them as arguments.

## 3.5 Cooperation Dependent Commands

As mentioned earlier, only secretions and growing point propagation require the cooperation of particles. We also mentioned that there was a universal protocol for implementing these commands. In this section, we present the ECOLI translations of secretions and growing point propagation and explain how they work. The protocol was invented for the purposes of interpreting these two commands, so rather than present it independently, we will describe it with respect to each command below.

### 3.5.1 Secretions

At the heart of the secretion implementation is a very simple idea: a breadth-first walk of a graph can be done in a highly parallel and local manner. The result of that breadth-first walk can be used to measure the shortest-path-length of any particle from some distinguished source particle. The essential feature of a secretion is that the pheromone values decrease with the distance from the source of that secretion. Here, we take "distance" to mean the shortest-path-length from the source, where paths are composed of edges that represent the communications connections. Our ability to label particles with their shortest-path-length from some distinguished particle allows us to assign a distribution of values that is consistent with the requirements of a secretion.

#### The Protocol

There are three message types in the secretion protocol that a particle may receive: `SECRETE`, `UPDATE` and `RESTORE`. To announce its current value for a specific pheromone, a particle sends a `SECRETE` message. If at some time later, that particle receives information that invalidates the information it previously sent, then it sends an `UPDATE` message.

Since a particle may need to broadcast its pheromone value more than once in response to correcting messages, there is a `RESTORE` message to indicate that the current secretion is done. A particle detects that it is at the boundary by checking if its shortest-path-length from the source is equal to the extent of the secretion. The extent of the secretion is included in the messages responsible for distributing the pheromone. When a particle determines that it is at the boundary it initiates the `RESTORE` message. Eventually, the source receives

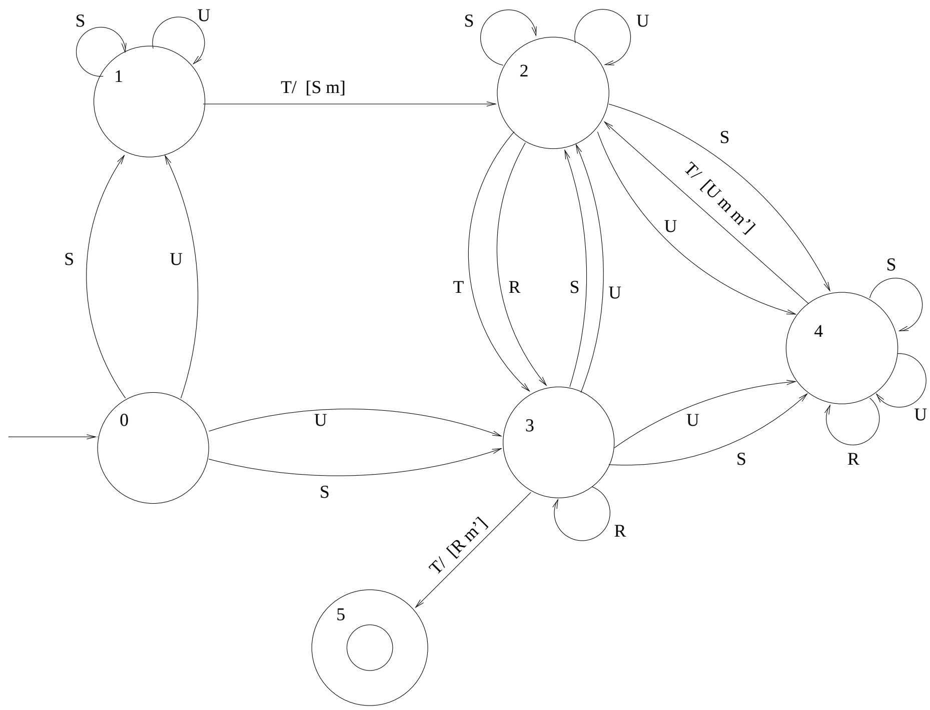


Figure 3-4: FSM for controlling a particle's reaction to messages generated during the secretion process.

a `RESTORE` message, which indicates that the secretion has reached its extent, and that the source particle may proceed with any pending computations.

### Auxiliary Data Structures

Each particle maintains a pheromone table that stores the current pheromone levels for all the pheromones that have reached that particle. This is the table that is consulted when a particle wants to know what the level of some pheromone is at its location. There is also some "scratch space" that each particle maintains to do the necessary bookkeeping during a secretion. Among other things, That scratch space contains the current list of neighbour values, the id number for the secretion, and the pheromone name. The neighbour values in the scratch space are used to compute the contribution of the current secretion which is then committed to the table.

That is, after a secretion source has sent a `SECRETE` message, but before it has received a `RESTORE` message, each affected particle must store its neighbouring values as well as maintain its state in the protocol.

### How it Works

Although the basic idea behind the implementation of secretions is relatively simple to describe, the actual implementation is complicated by the need to deal with simultaneous secretions, and uncertain termination conditions. We elaborate on the issues:

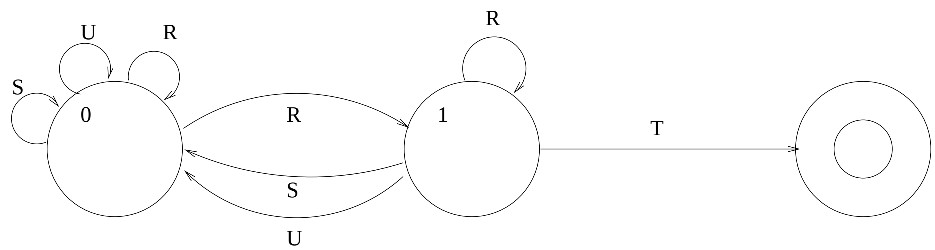


Figure 3-5: FSM for controlling the pheromone source's reaction to messages generated during the secretion process.

- Particles do not know who their neighbours are *a priori*, or how many there are, so there has to be a policy for deciding when to stop listening for messages.
- The possibility of corrections introduces the possibility of an update for the current secretion being confused with an initiation of a new secretion.
- The completion of a secretion is a non-local phenomenon. So there has to be some mechanism by which particles updating their pheromone concentrations know that the values have stabilized. This is particularly important to the source particle since it may have computation pending the completion of the secretion.

Instead of showing all the code involved in the implementation here, we present in Figure 3-4 an FSM depicting the computations that occur on a single particle in response to the messages it receives. Upon receipt of a SECRETE or UPDATE (from state 0), a typical particle transitions to state 1. There the particle collects the pheromone levels of all its neighbours. These are consolidated into a single value (states 2, 3 and 4) that will then be used by the particle (state 5) to represent its own value.

This approach is general enough to accommodate update rules that are not based on only the current value and a single incoming value. The transitions in the FSM in Figure 3-4 have not been labeled completely. Where multiple arcs leaving a state have the same label, there are conditions that must be met before the arc is taken. For example, if a particle in state 0 receives a SECRETE or UPDATE and realises that it is at the boundary of the secretion, then it transitions to state 3 instead of state 1. Also, on some arcs, side effects can occur to change the current information about the set of neighbour values. Appendix B has a complete ECOLI listing of the secretion FSM.

Since a particle *a priori* does not know the number of neighbours it has, it relies on a timeout mechanism in order to make progress. Specifically, upon receiving an initial message, it sets a timer for indicating when to stop waiting. If another message is received in the interim, it resets the timer. It decides that there are no more messages to receive only after the timer reaches zero. We call this mechanism a *cascaded timeout*. In the figure, each state with a 'T' arrow leaving it (i.e. states 1–4) implements a cascaded timeout. The 'T' arrow indicates the new state when the timer runs out.

State 2 is essentially a collection state, where a particle typically is waiting for a RESTORE message so it can transition to state 3. If while in state 2, a particle receives a message that changes its idea of its distance from the source, then it will eventually need to retransmit an update in its status, so it transitions to state 4.

State 3 is the state before committing the set of values that are being maintained for the secretion. When a particle receives a RESTORE and it has no broadcasts pending, then it transitions to state 3. If it manages to timeout of state 3, then it transitions to state 5. Any new information received in a SECRETE or UPDATE message while in state 3 causes a particle to leave it. When a particle transitions to state 5 all the values currently stored for each neighbour are committed. A pheromone concentration based on those values is computed and used to update the total concentration of the pheromone for that particle. The particle then returns to state 0, when the computation is complete in preparation for the next secretion.

Figure 3-5 is the FSM used by the secretion source for its own secretion. It is much simpler, since the primary task for the source is to determine when the secretion is complete. Note that the source particle has the option of using the pheromone concentrations of its neighbours to make modifications to its own, but we do not do that in our implementation.

### 3.5.2 The Propagate Command

The process of propagating a growing point is broken down into two communication phases: collecting pheromone values, and notifying the successor. In addition to those two processes is the computation that corresponds to the implementation of the tropism of the growing point. We shall now describe each of those processes in detail.

#### The protocol

When a particle needs to propagate a growing point's active site, it broadcasts a request (REQ) for pheromone values, and listens for results. Normally, neighbours acknowledge (ACK) the request with values for the pheromones requested. Neighbours may send back a response asking for extra time, this causes the first particle to reset its timeout counter to give the neighbour more time before it stops accepting responses. If all of the pending responses have arrived by the time the timeout triggers, the particle terminates the process, otherwise, it waits for one more timeout period, after which it terminates the process regardless of whether there are pending neighbours or not.

Upon termination of the collection process, the results are filtered and sorted by the tropism code, and one neighbour is selected as the successor. To notify that particle of its new status, the first particle sends a confirming (CONF) message that contains its own id, that of the succeeding neighbour, and all the current dynamic data for the growing point (e.g. the current values of the growing point's parameters).

#### Collecting Neighbouring Pheromone Values

The process that gathers a particle's neighbouring pheromone values is generated by the procedure `make-collector-process`, where the word *process* here is used in the ECOLI technical sense of the word. The procedure takes four arguments: the name of the generated process, a predicate for deciding how to respond to a request for pheromone values, the delay of the timeout mechanism, and the message tag whose handler is the continuation for the result of the process.

```
(define (make-collector-process name predicate delay exit-msg)
  (let ((result-buffer (list 'buffer))
        (pending-list (list 'pending-ids))
        (timeout-action #f))
    (define (restart-send-timer n) ...)
    (define (req-handler dict sender-id . args) ...)
    (define (make-specialised-req-handler special-id) ...)
    (define (ack-handler dest-id nbr-id result) ...)
    ;; initial instructions for all processors
    (ensure-binding 'id (random 65536))  ; 16-bit id
    (make-process
     name
     (make-dictionary
      (('REQ sender-id . args)
       (apply req-handler this sender-id args))
      (('WAIT-RETRY dest-id sender-id)
       (if (= (get-var 'id) dest-id)
           (begin
             (restart-send-timer 3)
             (set-cdr! pending-list (cons sender-id (cdr pending-list))))))
      (('ACTIVATE . args)
       (dict.ignore this 'REQ)
       (dict.bind this 'ACK ack-handler)
       (apply send name 'REQ (get-var 'id) args)
       ;; setup timeout to send
       (restart-send-timer 2))))
    ))
```

There are three variables local to the generated process. That is, the handlers and procedures used by the collector process returned are closed over these three variables. The first is the buffer for temporarily storing the pheromone values as they are reported by the neighbours. The second variable is a registry of the neighbours that have asked for an extension to respond. This is used to determine whether any extra time ought to be granted after a send timeout. The third variable provides a handle on the countdown action that is initiated after the particle sends its request for values. This handle permits the countdown to be aborted.

There are four message types recognized by the generated process: `REQ`, `ACK`, `WAIT-RETRY`, `ACTIVATE`. The first three collectively implement the protocol. The `ACTIVATE` message type is generated by the particle and sent to itself when the process is invoked. As described in Chapter 3, every process assumes the existence of an `ACTIVATE` message type. When a particle invokes the process, first it deactivates the `REQ` handler, thereby preventing another neighbour from simultaneously recruiting it for the same growing point. It then installs the `ACK` message handler, and sends a `REQ` message that contains its id number and the pheromone names whose values it needs; finally it sets up the timeout mechanism.

Each neighbour subsequently invokes the `REQ` handler. The ultimate result of the `REQ` handler is that the neighbour responds with either an `ACK` message containing its pheromone levels, or with a `WAIT-RETRY` message, asking for more time. The original particle then reacts appropriately to each message type: either terminating and returning the collected results, or resetting its timeout mechanism to wait a little longer. Having presented the overview, we now give detailed explanations for each handler.

**The timer** is managed by the procedure `restart-send-timer`. It aborts the current countdown and restarts a new countdown action. It takes one argument which counts the number of times that the timer has been restarted. Metaphorically, this parameter is a "patience" meter.

```
(define (restart-send-timer n)
   ;; n is the number of timeouts before give up on pending ids
  (action.stop timeout-action)
  (set! timeout-action
        (delayed-perform
         delay
         (make-action
          (if (null? (cdr pending-list))
              (begin
                (set! timeout-action #f)
                (send-to-self exit-msg result-buffer))
              (if (= n 0)
                  (begin
                    (report 'collection 'timeout 'at 'time: (e-time)
                            'no 'response 'from: (cdr pending-list))
                    (set-cdr! pending-list '())
                    (set! timeout-action #f)
                    (send-to-self exit-msg result-buffer))
                  (restart-send-timer (- n 1))))))))
```

Upon invocation (i.e. when the timeout triggers), this new action first checks if any responses are known to be pending (by checking `pending-list`). If there are no pending responses, then there is no need to restart the timer again, so the process terminates, returning the results to the current continuation. On the other hand, if there are pending responses, and the particle has not run out of patience (i.e. `n` is still positive), it will reset the timer by recursively invoking `restart-send-timer`. If the particle is out of patience, then it simply terminates the process and returns whatever responses it has at that point, after making sure to clean up the `pending-list` variable so that it will not be confused the next time it invokes this process.

**Requests** for values are handled (on each neighbouring particle) by application of the procedure `req-handler` as implied by the code fragment taken from the `make-dictionary` expression above:

```
(make-dictionary
 (('REQ sender-id . args)
  (apply req-handler this sender-id args))
  ...
 )
```

Each request message contains the id number of the originating particle, and the names of all the pheromones for which it needs the concentrations. Those pheromone names are contained in the dotted list parameter `args`.

```
;; REQuest handlers
(define (req-handler dict sender-id . args)
  (let ((result (apply predicate sender-id args)))
    (dict.ignore dict 'DISABLE)
    (cond ((eq? result 'WAIT-RETRY)
           (send name 'WAIT-RETRY sender-id (get-var 'id))
           (delayed-perform
            (/ delay 2)
            (make-action
             (apply send-to-self name 'REQ sender-id args))))
          (result
           (send name 'ACK sender-id (get-var 'id) result))
          (else #f))))
```

The first operation performed by the handler is to invoke the predicate on `sender-id` and the pheromone names. The predicate may return one of three possible values: the list of pheromone values, *false*, or the symbol `WAIT-RETRY`. In the first case, the behaviour of the handler is straightforward. In the case of a `WAIT-RETRY` result, the particle sends a `WAIT-RETRY` message to the original sender and sets up a timeout action to poll itself at some later time (much shorter than the timeout period of the original sender). A result of *false* means that the request should not be answered, as would be the case if there were a detectable amount of poisonous (to the current growing point being propagated) pheromone at the particle's location.

**Acknowledgements** are processed on the original particle by the handler `ack-handler`. Acknowledgement messages are expected to contain the ids of both the intended target particle and the sending particle, as well as the pheromone values requested. The three parameters to `ack-handler` below represent, respectively, those three pieces of information.

```
;; ACKnowledgement handler
(define (ack-handler dest-id nbr-id result)
  (if (= (get-var 'id) dest-id)
      (begin                          ; add result to buffer
        (set-cdr! result-buffer
                  (cons (make-entry nbr-id result)
                        (cdr result-buffer)))
        (let ((nbr-pending (memv nbr-id (cdr pending-list))))
          (if nbr-pending
              (begin
                (set-cdr! pending-list (delv nbr-id (cdr pending-list)))
                (if (null? (cdr pending-list))
                    (begin
                      (action.stop timeout-action)
                      (set! timeout-action #f)
                      (send-to-self exit-msg result-buffer))
                    (restart-send-timer 2)))))))
```

A response to an `ACK` type message happens only if the destination id in the message matches the receiving particle's id. This ensures that if there happen to be two particles in the same neighbourhood trying to propagate the same growing point at approximately the same time, then the responses from the common neighbours will go to the appropriate propagating particles. Although the consequences of confused pheromone results are probably not very significant (since the names will be the same, but the values may have changed between the two polling times), the id number check serves as an extra measure of protection against a renegade particle falsely propagating a growing point in response to an acknowledging message.

### Generating the Collector Predicate

We saw from the definition of `make-collector-process` that a particle applies the predicate to the pheromone names in a request message to determine how it should respond. We now describe how that predicate is generated. The procedure `make-sel-predicate` is a procedure that takes the name of the process the predicate will be used in, and the dictionary for the process, and returns the predicate. The predicate itself is a procedure that takes the same arguments as the request message handler does.

```
(define (make-sel-predicate name dict)
  (lambda (sender-id target-names)
    (let loop ((results '()) (reqd-pheromones target-names))
      (if (null? reqd-pheromones)
          (begin
            (dict.bind dict 'CONF
                       (make-conf-handler sender-id name))
            (reverse results))
          (lookup-pheromone pheromone-table (car reqd-pheromones)
                            (lambda (level)
                              (if level
                                  (loop (cons level results)
                                        (cdr reqd-pheromones))
                                  'wait-retry))
                            (lambda (locks)
                              'wait-retry))))))
```

The predicate does two important tasks. The first is to try to produce a list of pheromone levels for each pheromone name given. If it fails to produce such a list, it returns the symbol `WAIT-RETRY`. The second task is performed only in the case when all the pheromones asked for are present. That task is to bind in the dictionary of the collection process, a handler for a confirming message. The confirming message is sent by the original sending particle when it has decided which particle will succeed it. By binding the handler for the confirming message only after a request message is successfully answered, we automatically ensure that only neighbours that responded with pheromone values may become the next successor. In other words, in the rare case that more than one neighbour happen to assign themselves the same id number, the originating particle either knows about it, or activates at most one of them.

### Obeying the Tropism

Respecting the tropisms is a fairly straightforward process since most of the work was done in the preprocessing of the tropism expressions. Here we present the application of the tropism representation, and describe how we use the empty and non-empty continuations to make a particle repeatedly try to propagate before deciding that the growing point is stuck.

The following output was obtained by compiling a GPL program for producing a tree. The name of the growing point for which this code was generated is `stem`. The compiler automatically defines a procedure called `stem-loop` that implements the complete process of applying the tropism to the buffer returned from the collect process, and checking to see if the resulting list is empty, repeating up to three times if necessary until a neighbour is found to propaagate to. Here we present the interesting part of `stem-loop`. The procedure `repeat-proc` is a procedure that checks to see if the number of repetitions have exceeded the limit to decide whether to try the collection process again. The procedure `stem-chooser` is quite straightforward: when given a list of entries, it selects the first one and extracts the neighbour id out of the entry. It then broadcasts a message to notify that neighbour that it has become the successor.

```
(let ((find-best
       ((stem-tropism (lambda ()
                        (report (quote is) (quote stuck))
                        repeat-proc)
                      stem-chooser)
        my-result
        last-result)))
  (report (quote my-result:) my-result (quote last-result:) last-result)
  (def-response
    'stem-collect-exit
    (make-handler
     (result-buffer)
     (let ((results (convert-results (cdr result-buffer))))
       (reset result-buffer)
       (report (quote responses:) results)
       ((find-best results) stop-name last-result args)))))
```

Recall that the result of parsing a tropism expression is a procedure that takes continuations as arguments, one for the empty case, and one for the non-empty-case. The result of that is a procedure that takes the current pheromone readings and those from the previous active site and returns a procedure that takes the input buffer of associations from neighbours to pheromone readings. So `find-best` is a procedure that expects a list of associations between neighbouring ids and pheromone readings. As defined, it will call `repeat-proc` if its list is empty after filtering or `stem-chooser` if there is at least one surviving entry.

The procedure `stem-loop` does not in fact directly execute the operations described so far. It merely sets them up as potential commands to be executed, and installs a handler for the exit message from the collection process. If invoked, that handler applies `find-best` to the list of results from the collection process. If at least one successful entry remains, it is passed to `stem-chooser` so that the successor can be notified.

### Notifying the Successor

Once a neighbour has been selected as a successor, all that remains is that it be notified. When notified, a particle receives a CONF message. To be able to respond appropriately, it must have the proper hander installed. Here is the code for the other half of the communications involved in propagation, namely the implementation of the handler for successor notification.

```
(define (make-conf-handler src-id name)
  (lambda (dest-id sender-id last-result . args)
    (and (= sender-id src-id)
         (= dest-id (get-var 'id))
         (apply send-to-self name 'ACTIVATE last-result args))))
```

## 3.6 Communication-Independent Commands

The commands discussed in this section do not require a particle to converse with its neighbours; so in the interest of exposition, we will discuss the implementation of these commands with reference to the GPL illustrator.

All the commands that do not require communication are implemented by straightforward interpretation. Since the most interesting non-communicating commands in GPL are the `connect` command and those involved in networks, we shall limit our discussion to those only.

In GPL, the body of a growing point is implemented as a procedure, but since growing points cannot be nested, closing environments for growing points never get more than one level deep. The significance of this fact is that continuations can be represented simply as the bindings for the parameters of the current growing point, together with a pointer into the text of the program. This means that continuations are small enough to be sent in messages, and can plausibly be used to implement commands that can defer the invocation of growing points, such as `connect` and `cascade`.

### 3.6.1 Deferring Growing Points with Continuations

Particles actually pass around a list of continuations, which resemble a stack on a conventional processor. In this way, the maximum number of deferred growing point evaluations is determined by the message length. There are two types of growing point continuations: one for growing points and the other for networks. Network continuations take an argument when they are invoked that indicate which terminal of a network the current particle is emulating.

#### Growing point continuations

Once growing point continuations exist, implementing a connect command is fairly straightforward. All that is involved is interpreting the first command in the sequence given to `connect` and wrapping up the rest in a continuation.

```
(define (illustrate-connect exp env point cont stop id last-result size gp)
   ;; create a gp-continuation of rest of exps and pass it along
  (let ((start-command (car exp))
        (end-commands (cdr exp)))
    (illustrate-command start-command env point cont
                        (make-gp-continuation end-commands env
                                              process-next-point
                                              stop id last-result size gp)
                        id last-result size gp)))
```

The value of `exp` is the list of commands given in the rest of the `connect` command. The value of `env`, despite its name, is not really a full-blown environment, but simply an extension of the common global env by the bindings of the current growing point. The parameter `point` represents the active site. It's value is used only by the scheduler and auxilliary display routines, but never by the instructions that a growing point body represents.

The parameters `cont` and `stop` are continuations. `cont` captures the computation that is supposed to follow the current command (`connect` in this case). The `stop` parameter is either a growing point continuation or a network continuation. It is invoked when a growing point terminates.

The parameters `id`, tt last-result, `size` and `gp` represent values that would be passed around by the particles in their implementation of the GPL interpreter. The value of `id` (almost surely) uniquely identifies the instance of the growing point, whose name is given by `gp`. The value of `size` is the size parameter for the current instantiation of the growing point. `last-result` contains pheromone information about the previous active site and is used in the interpretation of tropism clauses.

A growing point continuation will be invoked when the active site terminates the current growing point.

```
(define (illustrate-terminate point cont stop)
  (cond ((eq? stop 'stop)
         (cont))                          ; not connected to another gp
        ((net-cont? stop)
         ((call-pt-cont stop point) #f)  ; not an network output terminal
         (cont))
        (else
         (write-line `(CONNECTING at ,point))
         (call-pt-cont stop point)
         (cont))))
```

The procedure `call-pt-cont` invokes continuations. The special symbol `'stop` is used to indicate that there were no deferred growing points waiting for the termination of the current one. If the current growing point was invoked because of a network invocation, then `stop` will be a network continuation. In this case the current active site is not the location of a network output, and therefore should ignore the continuation. (A network continuation is implemented as a procedure that takes a terminal name, so it needs an application to actually do anything. The special case of `#f` as a terminal name causes the continuation to be effectively ignored.) If indeed `stop` is a valid growing point continuation, then it should be invoked since it contains the computation that is to be executed upon termination of the current growing point.

### Network Continuations

Network continuations are invoked by evaluating a `->output` command. The `->output` command works like the `terminate` command for growing points, but it also supplies the name of the terminal to emulate.

```
(define (illustrate->output exp env point cont stop)
  (cond ((eq? stop 'stop)
         (cont))
        ((net-cont? stop)
         (for-each (call-pt-cont stop point) exp)
         (cont))
        (else
         (error `(->output ,@exp) "appears in illegal context"))))
```

A particle interprets a network by "projecting" the network over the input terminal that the particle is currently emulating. The procedure `project-seq` is used to interpret the body of a `define-network` command, where sequences of network commands may be encountered. The implementation of `project-seq` is straightforward: project the first command over the given terminal, and save the rest of the computation in the `cont` parameter to that projection.

```
(define (project-seq terminal seq env point cont stop id)
  (cond ((null? seq) (cont))
        ((null? (cdr seq))
         (project terminal (car seq) env point cont stop id))
        (else
         (project terminal (car seq) env point
                  (lambda ()
                    (project-seq terminal (cdr seq) env point cont stop id))
                  stop
                  id))))
```

The parameter `terminal` is the name of the current terminal of the current network being invoked. All the other parameters are as they were for `illustrate-connect` (except that `seq` here replaces `exp` previously). The `project` simply finds the appropriate procedure to invoke depending on the type of the command given to it.

```
(define (project terminal exp env point cont stop id)
  (cond ((not (pair? exp))
         (error "Unrecognised network command" exp))
        ((not terminal)
         (illustrate-terminate point cont stop))
        ((at-exp? exp)
         (project-at terminal (cadr exp) (cddr exp) env point cont stop id))
        ((location-binding? exp)
         (project-location-binding terminal exp env point cont stop id))
        ((cascade? exp)
         (project-cascade terminal (cdr exp) env point cont stop id))
        (else
         (project-seq terminal exp env point cont stop id))))
```

The projection operation simply teases out the parts of a command that are relevant to `terminal`. The purpose of each of the `project-`*mumble* procedures is to parse a network command and find any instructions that are associated with `terminal`. These instructions will then be executed when the location that corresponds to `terminal` is instantiated – typically when either a `cascade` or `at` command is evaluated.

The simplest case of a projection is of the `at` command. Here is the implementation of `project-at`.

```
(define (project-at terminal at-terminal at-body env point cont stop id)
  (if (eq? terminal at-terminal)
      (illustrate-seq at-body env point cont stop id #f 0 #f)
      (cont)))
```

An `at` command is how instructions for a terminal are explicitly given within the definition of a network. So projecting an `at` command over a terminal is simply a matter of matching the terminal with the expression to see whether the terminal is the one being referenced by the `at` command. If the match fails, the particle gets on with what it was doing before by invoking its continuation that encodes all of its pending operations.

As mentioned earlier in the discussion of `illustrate-terminate`, we ignore a network continuation if it was encountered by terminating the current growing point. In fact, we cannot simply leave it at that. Once we have gotten rid of the useless network continuation from the wrapped sequence of pending continuations, we need to propagate the fact that the current growing point terminated to the next continuation on the list. So, when `project` is handed `#f` as a terminal name, it hands back control to the `illustrate-terminate` procedure so that the next growing point continuation can be handled appropriately.

We could have implemented `illustrate-terminate` to simply unwrap the next growing point continuation from `stop`, but then we would have lost the flexibility of taking special action with terminated growing points that were invoked from within networks. Such a feature is potentially useful if, for example, we wanted to add a notion of serialised networks that waited for all their invoked growing points to terminate before performing some other action.

Now that we have seen how network continuations and network commands are implemented, we are equipped to understand how `cascade` is implemented.

```
(define (project-cascade terminal exp env point cont stop id)
  (let ((inputs (car exp))
        (rest-cascade (cddr exp)))
    (if (memq terminal inputs)
        (if (null? rest-cascade)
            (illustrate->output
             (list (alias terminal inputs (cadr exp)))
             env point cont stop)
            (let ((first-net (eval-exp (cadr exp) env point)))
              (if (network? first-net)
                  (project (alias terminal inputs (net.inputs first-net))
                           (net.body first-net)
                           env point cont
                           (make-net-continuation
                            (make-cascade-exp (net.outputs first-net)
                                              rest-cascade)
                            env process-next-point stop id)
                           id)
                  (error "The object" first-net
                         "passed to ==> is not a network"))))
        (cont))))
```

The best way to parse the behaviour of `project-cascade` is to think of a network cascade recursively. The expression

$$\texttt{(==> (}i_1 \ldots i_n\texttt{) net-a net-b ...(}o_1 \ldots o_m\texttt{))}$$

can be interpreted as the two step sequence:

1. Draw `net-a` at the locations specified by $i_1 \ldots i_n$ to produce the outputs $a_1 \ldots a_k$, where $k$ is the number of output terminals with which `net-a` was defined.

2. Evaluate $\texttt{(==> (}a_1 \ldots a_k\texttt{) net-b ...(}o_1 \ldots o_m\texttt{))}$

The sequencing of operations is accomplished by making a network continuation to capture the second operation, and passing it to the `project` command to carry out the first. The procedure `alias` takes a terminal name, a source list of names and a target list of names. The target list is a renaming of the names in the source list. `alias` searches the source list for an occurrence of the given terminal name and returns the corresponding entry in the target list. The terminals have to be renamed during the cascade because each terminal location is first an output of a network (with some name from that network) and then an input to a different network (and therefore with a new name).

# Chapter 4

# Theoretical and Practical Limitations of GPL

At this point, we have demonstrated that it is possible to implement GPL programs on a computational system that must be programmed uniformly and permits only local interactions among its elements. Given the rich set of patterns that GPL can express, that demonstration is a convincing indicator that we can exercise remarkable control over such a system. We shall now explore the limitations of GPL, both theoretically and practically in terms of our translation to ECOLI.

In this chapter we support the claim that any planar graph can be drawn by a GPL program. We carefully define what it means for a GPL program to "draw" a graph. Then we present a method for generating an appropriate GPL program, based on a given planar graph. As a part of our discussion, we present a general framework for analysing GPL programs, outlining the resources that we consider important, and how they are measured. The success of GPL programs is highly dependent on the connectivity properties of the GPL domain. In the last section of the chapter, we describe an experiment designed to investigate that dependence.

The method used to draw a planar graph is based on a result by de Fraysseix, Pach and Pollack [16] where they showed that any planar graph can be embedded on a grid, and still have every edge represented by a straight line. The importance of that result for our purposes is that planar graphs need only a finite resolution to be rendered. Therefore, if we have sufficiently many locations in our domain, we can always make room for many edges that converge at vertices by scaling the computed coordinates of the vertices.

In our approach, we first establish regions in the domain that correspond to the vertices of the given graph. The establishment of these regions is guided by the coordinates computed for the vertices. The second step is to draw line segments between vertices that are connected by edges in the given graph. To ensure that unintended connections are not made accidentally, particles in the vertex and edge regions secrete short range pheromones that poison growing points that ought not to get too close. Figure 4-1 illustrates the approach.

It is important to keep in mind that for any particular graph, its GPL implementation is likely to be able to take advantage of properties of the graph that we are not allowed to assume in general. The constructions presented in this chapter are general and they serve to show what is possible with GPL; they may not be the best approach to implementing a particular pattern. Chapter 5 presents a range of problems and demonstrates how each type of problem is tackled in GPL.

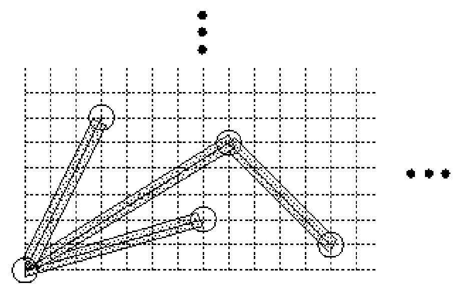

Figure 4-1: Vertices and edges are represented by regions. Regions for edges are indistinguishable when they overlap. Therefore a vertex with high degree must be represented by a large region so that there is enough room around that region to represent all the edges incident to that vertex.

## 4.1 A Framework for Analysing GPL Programs

In this section we describe how one goes about analysing a GPL program: how to determine growing point trajectories and what the relevant resources are. We will also analyse two basic subroutines, namely line-segments and rays, that recur under numerous guises in the implementations of many of the patterns produced throughout this dissertation.

### 4.1.1 Some Notation

For this chapter we shall use the following notational conventions. The GPL domain is represented as $D$ and its metric is written $d(p, q)$ for $p, q$ in $D$. Without loss of generality, we shall assume that the step-size of the GPL domain has been set to 1. The neighbourhood of a point, $p$, in $D$ is represented as $N(p)$ and is defined by

$$N(p) = \{q \in D \mid d(p, q) \leq 1\}$$

(since the step-size has been set to 1). Sometimes we also refer to $D$ as a graph; then it should be understood that the vertices are all the points in $D$ and the edges connect neighbouring points.

In the context of graphs, if $u$ and $v$ are two vertices with an edge between them, we denote that edge as $uv$. All the graphs discussed here are undirected graphs, so $uv$ represents the same edge as $vu$. In general, if $G$ is a graph, we write $V(G)$ and $E(G)$ to denote the vertices and edges of $G$, respectively.

### 4.1.2 Denoting GPL Computation

Recall from chapter 2 that in order to evaluate a GPL program we need a GPL domain as well as a set of initial conditions. The initial conditions are presented as locations in the domain that have been associated with code fragments to be executed on them. In order to carefully represent the effects of an evaluation of a GPL expression at a location in the domain, we present here a notation for describing the relevant aspects of GPL computation.

**Definition 4.1** *A* labeling *over a given alphabet, $M$, of the GPL domain, $D$, is a function* *that maps $D$ to the power set of $M$, denoted $2^M$.*

We assume that the GPL program has been evaluated, and that the result was a labeling of the particles in $D$. We define the predicate $\mathcal{Q}$ so that $\mathcal{Q}(\texttt{exp}, p)$ asserts that the GPL expression `exp` was evaluated at the point $p$ during the computation.

**Definition 4.2** *A growing point $g$ is said to be* invoked *at a point $p$ in $D$ if the particle at $p$ evaluates the body of $g$.*

**Definition 4.3** *A growing point $g$, is said to* propagate *from $p$ to $q$ if $q \in N(p)$ and an evaluation at $p$ of a* `propagate` *expression in the body of $g$ results in the invocation of $g$ at $q$.*

A growing point can be invoked at a location in one of two ways: either it is propagated from a neighbouring point, or a `start-gp` command was evaluated at the point.

**Definition 4.4** *A growing point, $g$, is said to be* stuck *at a point $p$ in $D$ if the particle at $p$ evaluates a* `propagate` *command in the body of $g$, but there are no neighbours of $p$ that satisfy the constraints of the* `tropism` *and* `avoids` *clauses.*

In order to refer to the collection of points that a growing point visits, we define the following two sets:

$$\mathcal{P}_1(g,p) = \{q \in N(p) \mid g \text{ propagated from } p \text{ to } q\}$$

$$\mathcal{P}(g,p) = \begin{cases} \{p\} \cup \bigcup_{q \in \mathcal{P}_1(g,p)} \mathcal{P}(g,q) & \text{if } g \text{ was invoked at } p \\ \emptyset & \text{otherwise} \end{cases}$$

Intuitively, we may think of $\mathcal{P}_1(g,p)$ as the set of neighbours of $p$ to which the growing point $g$ propagated after it was invoked at $p$. The set $\mathcal{P}(g,p)$ represents the set of points that form the trajectory of $g$ resulting from the invocation of $g$ at $p$.

**Definition 4.5** *The* embedded trajectory of $\mathcal{P}(g,s)$ *is the directed graph in the plane obtained by taking as vertices the locations in $\mathcal{P}(g,s)$. The edges are obtained by drawing a straight line from each vertex $p$ to each $q$ in $\mathcal{P}_1(g,p)$.*

The following theorem tells us that growing point trajectories are always connected components in $D$, and that this property is a direct consequence of the locality of growing point propagation.

**Theorem 4.1** *For any growing point $g$, if $g$ is invoked at $p$, then $\mathcal{P}(g,p)$ is a connected component in $D$.*

**Proof:** The proof is a fairly straightforward use of induction on the definition of $\mathcal{P}$, but just to be careful, we present it in its entirety.

Since $g$ was invoked at $p$, then by definition $p \in \mathcal{P}(g,p)$. So if $g$ terminates immediately at $p$, the theorem statement is trivially true.

By hypothesis, for $q \in \mathcal{P}_1(g,p)$, $\mathcal{P}(g,q)$ is a connected component. So for each $q \in \mathcal{P}_1(g,p)$, $\{p\} \cup \mathcal{P}(g,q)$ is a connected component (since $\mathcal{P}_1(g,p) \subseteq N(p)$ by definition of $\mathcal{P}_1$). Therefore, $\mathcal{P}(g,p) = \{p\} \cup \bigcup_{q \in \mathcal{P}_1(g,p)} \mathcal{P}(g,q)$ is a connected component. ■

Theorem 4.1 will be useful for establishing the proof of the main result. Although we will assume that our growing points never got stuck, Theorem 4.1 still holds if $g$ gets stuck. We also have the following corollary:

**Corollary 4.1.1** *For any given pattern, the number of initial locations given to a GPL program designed to achieve that pattern must be at least as many as the number of connected components in the pattern.*

Recall that $D$ is embedded in the two dimensional Euclidean plane. Ultimately we view the colour coding of the labeling produced by a program as a pattern in the plane, whose shape is determined by the Euclidean metric. Intuitively we have a good idea of what it means for our GPL program to draw a given pattern, but we shall need a precise definition in order to substantiate any claims about the success of a GPL program.

**Definition 4.6** *A k-ball, denoted $\mathcal{B}_k(p,r)$, is the set of points in $\mathbf{R}^k$ that are at most a distance $r$ from $p$. $\mathcal{B}_k(p,r) = \{q \in bfR^k \mid d(p,q) \leq r\}$.*

**Definition 4.7** *The* planar interpretation*, $\mathcal{I}$, of a labeling $\mu$ on $D$ is the union of the set of 2-balls of radius 1 (step-size), centred at locations that have been assigned non-empty labels by $\mu$. Formally, $\mathcal{I}(\mu) = \cup_{p \in D, \mu(p) \neq \emptyset} \mathcal{B}_2(p,1)$.*

The planar interepretation of a labeling on the GPL domain is a region in the plane. It gives us a way to relate growing point trajectories, which are logical in nature, to physical regions in the plane. However, the growing points that we design are modeled on curves in the plane, so we must have a method of relating regions in the plane to curves in the plane. We will then be able to evaluate the success of a growing point's definition by applying that method to the planar interpretation of its trajectory and then comparing the result to the curve on which the growing point was modeled.

We turn to Algebraic Topology in search of such a method of relating regions to curves. The following definitions of terms, which have been slightly paraphrased to avoid notational clutter, were provided by Maunder [35].

**Definition 4.8** *A subspace $A$ of a topological space $X$ is a* retract *of $X$ if there exists a map $r : X \mapsto A$ (called a* retraction*), such that $r(a) = a$ for all $a \in A$. If $i : A \mapsto X$ denotes the inclusion map and $ir$ is homotopic to the identity map on $X$ then $r$ is a* deformation retraction *(and $A$ is a* deformation retract *of $X$).*

Restated in more accessible terms, $A \subset X$ is a deformation retract of $X$ if there is a continuous map $F : X \times [0,1] \mapsto X$ such that for every $x \in X$, $F(x,0) = x$ and $F(x,1) \in A$ and for all $a \in A$ and all $t \in [0,1]$, $F(a,t) = a$.

We can regard a simply connected closed region, $R$, in the plane as a topological space. So we can regard a curve $C$ lying in $R$ as a subspace. In this way, we may talk of finding a deformation retraction that maps $R$ to a curve $C$, thereby "relating" the region $R$ to the curve $C$. Stated in terms more accessible to readers unfamiliar with Topology, we "relate" a region $R$ with a curve $C$ by finding a curve inside $R$ to which $R$ can be "warped" (i.e. continuously deformed). Once we obtain such a curve, we can then evaluate it on its topological merits (i.e. are its end points correctly placed, and is it representative of the curve we had in mind when we designed the growing point to render it). We are now ready to carefully define what we mean when we say that we can produce a GPL program that "draws" a line.

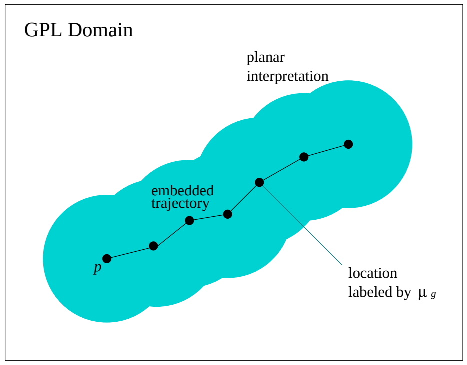


Figure 4-2: Growing point $g$ was invoked at location $p$; the result was $\mu_g$. The locations with non-empty labels assigned by $\mu_g$ are indicated by small black disks (we assume that $g$ has width 0). The embedded trajectory is shown with the connecting lines between its adjacent locations. The planar interpretation of $\mu_g$, $\mathcal{I}(\mu_g)$ is shaded.

**Definition 4.9** *We say that a GPL program* draws *a line, $AB$, if the program produces a labeling such that a curve homeomorphic to the unit interval is a deformation retract of the planar interpretation of the labeling.*

**Proposition 4.1** *Given growing point $g$ and a location $p$ in $D$, let $\mu_g$ be the labeling that results from invoking $g$ at $p$. The embedded trajectory of $\mathcal{P}(g,p)$ is a deformation retract of the planar interpretation of $\mu_g$.*

**Proof:** First, it is clear that the embedded trajectory of $\mathcal{P}(g,p)$ lies entirely within $\mathcal{I}(\mu_g)$. Two consecutive points of the embedded trajectory are less than 1 step-size unit removed from each other. So the straight line joining them is contained in the union of the 2-balls (disks), each of radius 1 step-size, centred at those points. Since $\mathcal{P}(g,p) \subseteq \{q \in D \mid \mu_g(q) \neq \emptyset\}$ it follows that the union of 2-balls centred at the points in the path is a subset of $\mathcal{I}(\mu_g)$. So we conclude that the embedded trajectory must be contained within $\mathcal{I}(\mu_g)$ (see figure 4-2. Now we need to show that the embedded trajectory is a retract of the planar interpretation, $\mathcal{I}(\mu_g)$.

Edelsbrunner [17] uses a "decomposition by join" technique to construct a deformation retraction map from the union of balls to a simplicial complex within the union of balls. Maunder [35] offers a variation on that approach. He provides a useful theorem, which states that if $f, h : X \mapsto Y$ are two maps on a topological space $X$ and if for every $x \in X, f(x)$ and $h(x)$ can be connected by a straight line lying entirely within $Y$, then $f$ and $h$ are homotopic.

In this context, we can regard the two maps as each mapping $\mathcal{I}(\mu_g)$ to itself. The first map is the identity and the second maps $\mathcal{I}(\mu_g)$ into the embedded trajectory of $\mathcal{P}(g,p)$. There is a straight line that connects the boundary of $\mathcal{I}(\mu_g)$ to its image on the embedded trajectory of $\mathcal{P}(g,p)$. Since this line lies entirely within $\mathcal{I}(\mu_g)$, that polygonal path is a deformation retract of $\mathcal{I}(\mu_g)$. ■

Proposition 4.1 tells us that the path described by $\mathcal{P}(g,p)$ is always a good candidate curve to which the planar interpretation of the labeling resulting from the invocation of $g$

"relates". We shall use this proposition to prove that we are able to draw line segments with GPL programs.

### 4.1.3 Some Basic Assumptions

- The pheromone distribution resulting from a secretion is such that the concentration values decrease with distance from the source. The pheromone values are zero at and beyond the range of the secretion. Every particle within range of the source obtains a non-zero contribution to the concentration value of the secreted pheromone.

- Pheromone values are always non-negative.

- Propagation occurs only after pheromone values have stabilized. So for the purposes of analysis, we assume that when a particle evaluates a `propagate` command, the concentration values for the relevant pheromones at that particle are no longer changing.

- If a pheromone has zero concentration at a particle that is trying to propagate a growing point that depends upon that pheromone, then the growing point stalls, waiting for the presence of that pheromone. This state is different from a stuck state, since there is no way to be certain that the propagation step has failed. We shall assume that secretions

### 4.1.4 Watching Resources

There are three resources that we shall be concerned with when analyzing the performance of a GPL program: particle memory, time, and domain space. The first two resources are quite familiar, but the third needs a little explanation: One of the possible outcomes of a GPL program is that the growing points may fail to form the pattern we hoped for by becoming "stuck". That happens when a growing point attempts to propagate from a particle, but there are no eligible neighbours to which the growing point may be passed. This concept leads us to regard the domain of particle locations as a resource.

#### Memory

In GPL, the principal resource is the particle memory. We distinguish between the static and the dynamic memory requirements. The static memory requirements are determined by the size of the GPL program. The dynamic memory requirements refer to the amount of memory required during the execution of a program. For a single particle, the dynamic memory requirements are determined by: the longest message it sends or receives, the number of pheromones it detects and the number of materials with which it gets labeled. The length of the message a particle sends is in turn determined by the number of deferred commands (by either the `connect` form or the `==>` form) and the length of the parameter list for the current growing point.

#### Time

In GPL, the operations performed by a single particle are relatively simple and take little time compared with the time to communicate with its neighbours. Furthermore, a growing point's body is evaluated on a particle at most once per invocation of the growing point. So the running time of GPL programs is measured not by the time for a single particle to complete its relevant tasks, but rather as the time for all growing points to terminate—this is in contrast to the memory usage described above. The unit of time is the time needed to exchange a single message between particles. The messages will either be pheromone updating transactions (in the production of secretions) or growing point arguments (in the propagation of growing points).

In our model, we assume that there is a fixed upper bound on the lengths of these messages (fixed sized frames), and that if a computation requires a longer message to be sent, it must be fragmented and sent in pieces. Since this upper bound is unspecified, we assume here that it is large enough to allow any of the messages to be sent by our program to fit in a single frame. This assumption simplifies the description of the running time, and alters it only by a constant factor, since all message sizes can be statically bounded given the GPL program that is to be evaluated.

The time required by a GPL program to complete its evalutation is determined by the largest total length of growing point trajectories that were drawn end-on-end, and by the extent of the secretions used. Each propagation step takes a constant amount of time under the assumptions above (notwithstanding a stall due to a missing pheromone). Note that the time costs that we have been referring to here are parallel time costs, not the time for a sequential simulation of the program. This distinction is particularly important when we treat the time taken by secretions. The time for pheromone secretion depends upon the implementation of secretions that we choose to use. If we use a diffusion-like implementation, then the time is quadratic in the extent, however if we use a shortest-path type of computation, as was shown in Chapter 3 then the parallel time required is linear in the extent of the secretion.

#### The Domain

The property of the domain being measured is the region that the pattern occupies, measured in step-size units.

The amount of space a growing point occupies is determined by its length in steps and its width (i.e. `size` attribute). It may also potentially consume more space by secreting pheromones that other growing points avoid. If those pheromones are secreted for mutual exclusion purposes, then the extent of the secretions may be the more appropriate measure to use instead of the `size` attribute.

### 4.1.5 Analysing Rays and Line Segments

As an example of the application of this framework, we shall now analyse two very commonly used routines: rays and line-segments. Key to our methods is the ability to generate a line: either as a ray from a point and in an optionally specified direction or as a line segment between two given points. The two types of lines have different applications based on the information they require. Rays are useful for invoking growing points at locations in the domain that are distantly removed from the initial locations. Their limitation is that their path is heavily dependent upon the underlying geometry of the domain. Line segments, on the other hand, are much more robust to the underlying geometry, but both of their end points must be established before they can be used. In this section, we present the simplest versions of these two line drawing techniques and analyse their behaviour. Many parts of our construction are based on the same principles, so our analysis here will also serve as a basis for some of our statements made in the body of the proof of the proposition.

### Line Segments

We present line segments first because they are simpler in concept, although they require more code. The principle is quite simple: we establish a pheromone gradient centred at one end, and invoke an attracted growing point (i.e. one with positive orthotropism to the pheromone) at the other end. Here are the definitions of those two growing points.

```
(define-growing-point (AB-segment)
  (material A-material)
  (tropism (ortho+ B-pheromone))
  (actions
   (when ((sensing? B-material)
          (terminate))
         (default
           (propagate)))))

(define-gp (B-point)
  (material B-material)
  (size 0)
  (actions
   (secrete+ radius B-pheromone)
   (terminate)))
```

In the definition of `B-point`, the variable `radius` refers to a globally declared constant (using the `constant` GPL form). In what follows, we shall write `[radius]` to denote its value at run time. Assuming we evaluate the above growing points in a GPL domain, $D$, that is devoid of `B-pheromone`, we can assert the following Lemma.

**Lemma 4.1 (Segment Lemma)** *For $s, t$ points in $D$ if $d(s,t) \leq$ [*`radius`*] and $\mathcal{Q}($*`(start-gp AB-segment)`*$,s)$ and $\mathcal{Q}($*`(start-gp B-point)`*$,\ t)$ then either* `AB-segment` *got stuck or it draws a line from $s$ to $t$.*

**Proof:** For brevity, let $P = \mathcal{P}($`AB-segment`$, s)$. Let $\phi_B : D \mapsto \mathbf{R}$ denote the distribution of `B-pheromone` in the domain. From $\mathcal{Q}($`(start-gp B-point)`$, t)$ it follows that $\mathcal{Q}($`(secrete+ radius B-pheromone)`$, t)$. So $\phi_B$ is a maximum over $D$ at $t$. Also since $d(s,t) \leq$ [`radius`] then $\phi_B(s) > 0$, so `AB-segment` will be able to propagate from $s$.

$\mathcal{Q}($`(start-gp AB-segment)`$,s) \Rightarrow s \in P$

If $s = t$ then `(sensing? B-material)` evaluates to true at $p$ and `AB-segment` terminates there. In this case, $P = \{s\} = \{t\}$ and the theorem statement is true.

In general, since $t$ is the only point at which `(sensing? B-material)` evaluates to true, if `AB-segment` terminates (i.e. does not get stuck) then $t \in P$.

If $s \neq t$ then `(sensing? B-material)` evaluates to false and the `propagate` expression is evaluated at $s$. Since $\phi_B(s) > 0$, there are non-zero values of $\phi_B$ in the neighbourhood of $s$ and so the particle at $s$ does not block waiting for the presence of `B-pheromone`. So in propagating from $s$, the `ortho+` filter is applied to the set $\{\phi_B(p) \mid p \in N(s)\}$.

Let $C$ be the resulting set of candidate neighbours for propagation. i.e. $C = \{p \in N(s) \mid \phi_B(p) > \phi_B(s)\}$. If $C$ is empty then `AB-segment` is stuck, so we can assume that there is at least one element in $C$.

Since $C$ is non-empty, it is possible for `AB-segment` to propagate from $s$, say to $p \in C$ and so $p \in \mathcal{P}_1($`AB-segment`$,s)$. Furthermore, $\phi_B(s) < \phi_B(p)$, so $d(s,t) > d(p,t)$.

Since there were no assumptions about $p$ except that $p \neq t$, the preceding reasoning applies to any of the points in $P$ except for $t$. That is, in general, $q \in \mathcal{P}_1($`AB-segment`$,p) \Rightarrow$

$d(p,t) > d(q,t)$. It then follows from the definition of $\mathcal{P}$ as a union of $\mathcal{P}_1$ results that $P = \{s = p_0, p_1, p_2, \ldots, p_n = t\}$ such that for $0 \leq i < n, d(p_i,t) > d(p_{i+1},t)$. Clearly all of the $p_i$'s are distinct points in the plane. It also follows that the embedded trajectory of $P$ is the path $\Pi = p_0 p_1 \ldots p_n$. By Proposition 4.1 $\Pi$ is a deformation retract of the planar interpretation of the labeling produced by `AB-segment`, so we now need to show that $\Pi$ is homeomorphic to the unit interval [0,1].

The map $f : \Pi \mapsto [0,1]$ whose rule for $t \in [0,1]$ and for integer $i$, $0 \leq i < n$ is given by:

$$f((1-t)p_i + tp_{i+1}) = (i+t)/n$$

is a bijective continuous map. Therefore the path $\Pi$ is homeomorphic to the unit interval [0,1]. Hence, the growing point `AB-segment` draws a line from $s$ to $t$. ■

In this example, the line-segment uses $O([\texttt{radius}])$ domain space, takes $O([\texttt{radius}])$ time steps (assuming linear time for the secretion) and uses constant space (both static and dynamic).

### Rays

Only one growing point is required to produce (an approximation to) a ray. The idea is that the growing point secretes self-repelling pheromone for a short distance. The superposition of two such secretions biases the choice of the succeeding location to be made in a consistent direction determined by the ordering of the two secretions. Here is the growing point definition for a ray:

```
(define-growing-point (ray life)
  (material ray-mat)
  (tropism (ortho- self-pheromone))
  (size 0)
  (actions
   (secrete+ 3 self-pheromone)
   (when ((< life 1)
          (terminate))
         (default
           (propagate (- life 1))))))
```

The required initial condition is that one location effects a secretion of `self-pheromone` for an extent of 3 (any value large enough to influence a particle two hops away is enough), and another location within the range of influence of the first secretion invokes the `ray` growing point. Here is the code to effect this:

```
(with-initial-locs
 (base-pt source)
 (at base-pt
     (secrete+ 3 self-pheromone))
 (at source
     (start-growing-point ray 20)))
```

Let us now analyse the effect of evaluating the ray program with two neighbouring locations $p_0, p_1$ supplied as `base-pt` and `source` respectively. Suppose that there is no appreciable lag between when commands are invoked at $p_0$ and at $p_1$ so that we can assume that by the time the `propagate` command is being evaluated, the secretion at $p_0$ has already

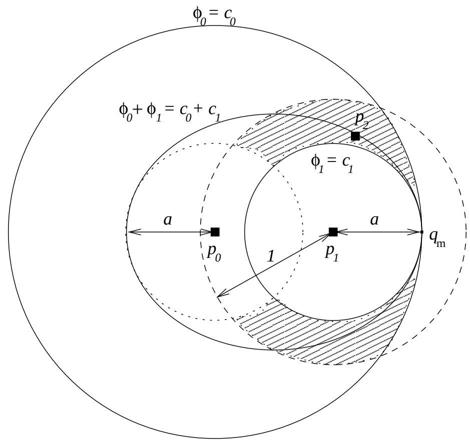


Figure 4-3: Propagation of the `ray` growing point. Pheromones have already been secreted from locations $p_0$ and $p_1$. The functions $\phi_i$ are abbreviations for $\phi[p_i]$, and the $c_i$ are constants. Contours of $\phi_0, \phi_1, \phi_0 + \phi_1$ are shown as well as the region from which the successor location $p_2$ might be chosen.

occurred. (In order to guarantee that this actually happens, we would have to introduce a few extra growing points that would act as serializers, but they would unnecessarily obscure the discussion here, so we ignore this detail for now.)

The initial conditions represent an abstract point in the domain (`source`) and a direction from that point in which the ray proceeds. The direction is specified by supplying a second point (`base-pt`) and it is interpreted as the direction from `base-pt` to `source`. Given $p_0$ and $p_1$ as the instantiations of those two points, we want to analyse the pattern that results from invoking the `ray`. In particular, we want to try to understand the range of options for propagation available to each active site as the `ray` growing point moves.

Let $u : \mathbf{R} \mapsto \mathbf{R}$ be the continuous decreasing function that characterizes a secretion with extent 3. That is, $u(l)$ represents the contribution of a secretion of extent 3 to a location that is at a distance of $l$ step-size units from the source of the secretion. Let $\phi[p](q) = u(d(p, q))$ denote the concentration value of `self-pheromone` secreted from $p$, and detected at $q$.

Let $\phi_T : D \mapsto \mathbf{R}$ denote the pheromone levels after secretions have been produced at $p_0$ and $p_1$, so that $\phi_T(q) = \phi[p_0](q) + \phi[p_1](q)$. Figure 4-3 illustrates the locations $p_0$ and $p_1$ and some of the contours of the the pheromone distributions $\phi[p_0], \phi[p_1] and \phi_T$.

Assuming $n \geq 1$, let $p_2$ be the point selected when `ray` propagated from $p_1$ (there must have been such a $p_2$, otherwise `ray` would have been stuck at $p_1$). Recall that the tropism expression `ortho-` is interpreted as a search for the neighbour with the minimal value of the specifed pheromone. It follows then that $p_2 \in N(p_1)$ and $\phi_T(p_2)$ is the minimum value of $\phi_T$ in $N(p_1)$.

Let $C$ denote the set of locations in $D$ that had the same pheromone value as $p_2$. So $C = \{q \in N(p_1) \mid \phi_T(q) = \phi_T(p_2)\}$. The set $C$ represents the set of candidate points that could have been chosen instead of $p_2$. Let us now investigate the properties of $C$ to see how the geometry of $D$ affects the trajectory produced by `ray`. We shall assume that $C$ has more than one location so that we can study what the geometric relationships between the alternative locations to $p_2$ are (even if in fact, $C$ contains only $p_2$, pretending that $C$

has more elements will tell us something about the permissible points in the plane where $p_2$ could be).

For any $q_1, q_2 \in C$,

$$\phi_T(q_1) = \phi_T(q_2) = \phi_T(p_2)$$
$$\Rightarrow u(d(p_0, q_1)) + u(d(p_1, q_1)) = u(d(p_0, q_2)) + u(d(p_1, q_2))$$
$$\Rightarrow u(d(p_0, q_1)) - u(d(p_0, q_2)) = u(d(p_1, q_2)) - u(d(p_1, q))$$

and

$$\begin{aligned} d(p_0, q_1) > d(p_0, q_2) &\Leftrightarrow u(d(p_0, q_1)) < u(d(p_0, q_2)) \\ &\Leftrightarrow u(d(p_0, q_1)) - u(d(p_0, q_2)) < 0 \\ &\Leftrightarrow u(d(p_1, q_2)) - u(d(p_1, q_1)) < 0 \\ &\Leftrightarrow u(d(p_1, q_1)) > u(d(p_1, q_2)) \\ &\Leftrightarrow d(p_1, q_1) < d(p_1, q_2) \end{aligned}$$

Suppose that $q_m$ is the point in $N(p_1)$ that minimizes $d(p_1, q)$ for $q \in C$, then $q_m$ simultaneously maximizes $d(p_0, q)$ over $q \in C$. So for each $q \in C$, we have

$$d(p_1, q_m) \leq d(p_1, q) \leq 1$$

$$d(p_0, q) \leq d(p_0, q_m) \leq d(p_0, p_1) + 1$$

In other words, all the points of $C$ are at least as far from $p_1$ as $q_m$ is, yet they are closer than $q_m$ is to $p_0$.

Furthermore, there are no points in $N(p_1)$ that are simultaneously farther from $p_0$ and $p_1$ than $q_m$. For,

$$\exists q \in N(p_1) : d(p_0, q) > d(p_0, q_m) \wedge d(p_1, q) > d(p_1, q_m)$$

$$\Rightarrow u(d(p_0, q)) + u(d(p_1, q)) < u(d(p_0, q_m)) + u(d(p_1, q_m))$$
$$\Rightarrow \phi_T(q) < \phi_T(q_m) = \phi_T(p_2)$$

which is a contradiction, by the minimality of $\phi_T(p_2)$.

Figure 4-3 is a diagram of the relevant pheromone loci around $p_0$ and $p_1$. The shaded region represents the region where the points of $C$ must lie. Although the diagram biases our thinking to assume a Euclidean metric, it is still useful in indicating the role of $q_m$. Since the set $C$ is always at least as far from $p_1$ as $q_m$ is, if $D$ has the property that $q_m$ is always located near the boundary of $N(p_1)$ then there will be few points in $C$, and furthermore, if $\phi[p_0]$ is not attenuated too much (a property we try to ensure in the implementation of secretions) they will all be located near the desired direction.

When $D$ is a regular grid, except at the edges, there are always locations in $D$ near a location's neighbourhood boundary. So each location propagated to by a growing point is as close to being in the direction required by the growing point's tropism as the grid will allow.

Using $n$ as the lifetime as in the statement of the lemma, the domain space occupied by `ray` is $O(n)$, the time taken is also $O(n)$ and it uses constant memory throughout its growth.

## 4.2 Implementing Arbitrary Networks with GPL

In this section, we formalise the notion of "drawing" a graph, and we establish the main proposition outlined at the beginning of the chapter.

### 4.2.1 The Main Proposition

Before we can present the statement of the main proposition, we must define a few concepts.

**Definition 4.10** *For a given labeling, $\mu$, on $D$, the* label induced subgraph *of $\mu$ is the graph, $D_\mu = (V, E)$, where*

$$\begin{aligned} V &= \{p \in D \mid \mu(p) \neq \emptyset\} \\ E &= \{pq \in V \times V \mid d(p,q) \leq 1\} \end{aligned}$$

**Definition 4.11** *Given graphs $G$ and $H$, we say $G$* resembles *$H$ if there is a map $f : V(H) \cup E(H) \mapsto 2^{V(G)}$ that has the properties:*

1. *$\forall x \in (V(H) \cup E(H)), f(x)$ is a connected component in $G$.*

2. *$(u \in V(H)$ incident to $e \in E(H)) \Leftrightarrow f(u) \cup f(e)$ is a connected component of $G$.*

**Definition 4.12** *We say that a GPL program* draws *a graph, $G$, if the program produces a labeling $\mu$ with the following properties:*

1. *$D_\mu$ resembles $G$,*

2. *there is a planar embedding of $G$ that is the retract of the planar interpretation of $\mu$.*

The claim of the thesis statement, and the focus of this chapter is the following proposition:

**Proposition 4.2** *Given a planar graph $G_P$, there is a GPL program that when evaluated on a square lattice, either draws $G_P$ or decides that it has failed to do so.*

A planar embedding of an abstract graph is necessarily isomorphic (in the graph sense) to the given graph. So in order to support the claim that any planar graph can be drawn by a GPL program, we need to show that there is some curve that is "related" (by retraction) to the planar interpretation of the labeled particles, and is also isomorphic to the given graph. Since the conditions for a map to be a retraction are rather weak, the first condition is included to ensure that the planar interpretation is not trivial (needs to be reworded).

We shall devote a considerable amount of this chapter to proving Proposition 4.2. Our proof holds in the case that $D$ is a regular lattice. We subsequently discuss the shortcomings of our constructions when $D$ is an arbitrary point set in the Euclidean plane.

We also provide upper bounds on the memory requirements and the running time of implementing a given graph, based on the constructions for our main result.

### 4.2.2 Overview of the Method

The construction is logically divided into two phases. For each vertex in the input graph, the first phase identifies a region in the GPL domain that will represent it. The second phase produces paths in the GPL domain to represent the edges of the graph.

The main idea is that points representing vertices secrete a long range and a short range pheromone. The long range pheromones will direct the formation of edge representing paths, while the short range pheromones prevent those paths from forming spurious connections. The edge representing paths are generated by growing points that are invoked and terminate at the regions corresponding to the edge's incident vertices. Since the input graph is planar and our output must resemble it, we have to ensure that our edge representations are individually discernible. To accomplish that, edge particles secrete short range pheromones that disable other potentially interfering paths, thereby maintaining a minimum distance between paths.

The constraint of minimum separation between paths introduces a new problem: paths need to converge when their corresponding edges share an incident node. We solve this problem by expanding the region that represents a vertex. Each path that represents an edge is divided into three parts. Two parts near the terminating nodes have no constraints on their proximity to other paths, but they label points in the domain as members of the node's region. The remaining part of the path lies in between the other two and labels points as members of the edge's region. That is the part that must maintain a minimum separation from other paths. So in fact, only the middle part of the path represents an edge, the other two parts represent the expansion to the node's region.

The approach of the vertex placement phase relies on a result by de Fraysseix, Pach and Pollack [16], which states that every planar graph can be embedded on a grid and still have every edge represented by a straight line. (They call such an embedding with straight lines for edges a Fáry embedding, recognizing that Fáry first proved that every planar graph has an embedding in the plane where every edge is represented with a straight line.) The first step of this phase uses the method of de Fraysseix et al. [16] to find coordinates for each vertex. The space a vertex's region might occupy is accounted for by scaling the computed coordinates by at least 2.

Since the definitions of many of the necessary growing points are specific to the given graph, we have parameterized them through the use of code generators. There are three principal types of code generators. The growing point produced by DEFNODE$[u, l]$ establishes the location of the vertex $u$. The parameter $l$ specifies the extent of its long range pheromone secretion. Vertex regions are expanded in directions of neighbouring (as determined by the edges) vertex regions. The growing point generated by NN-SEGMENT$[u, v, s(uv)]$ expands the region for vertex $u$ in the direction of the region for vertex $v$, for a distance $s(uv)$. Finally, the growing point generated by SEGMENT$[c, u, v]$ produces a path from the region for vertex $u$ to that of vertex $v$ and "colours" the edge with colour $c$. The colours are used to impose the minimum separation requirement for edge-representing paths.

There are also support definitions for establishing the initial vertex locations. These growing points produce a lattice with the origin at the given initial point. When the coordinates for a vertex, $u$, are established at a particle, the growing point generated from DEFNODE$[u, \cdot]$ is invoked at that particle.

### 4.2.3 Proof Outline for Main Proposition

The proof of Proposition 4.2 rests upon demonstrating key properties of several construction steps. We shall first present the construction in detail, along with the appropriate code fragments to implement them. Then we show that if our program successfully executes (i.e. no growing points are stuck) then it draws the given planar graph.

**Construction:** This is an off-line process. The steps described would be taken by a compiler that took the description of an input graph, $G$, and produced a GPL program as output.

1. Compute a grid layout for $G$.
2. Colour the edges of $G$, using a minimal set, $C$, of colours.
3. Compute $l(uv)$, the length of the line that represents edge $uv$ for each $uv \in E(G)$.
4. Compute $s(uv)$, the distance along the edge $uv$ for which it is closer than 2 step-size units to some other edge incident to $u$.
5. Compute the grid-spacing based on $s(uv)$ and $l(uv)$.
6. For each vertex $u$, let $l_u = \max\{l(vu) \mid vu \in E(G)\}$ and generate $\textsc{DefNode}[u, l_u]$.
7. For each edge, $uv$ generate both $\textsc{nn-segment}[u, v, s(uv)]$ and $\textsc{segment}[c_{uv}, u, v]$ where $c_{uv}$ is the colour assigned to edge $uv$ from step 2.
8. For each lattice $x$-coordinate where lies at least one vertex, generate $\textsc{v-ray}[x]$. Also generate any support code for producing a lattice.
9. To create the entry point, generate code to establish the directions of increasing lattice coordinates, and invoke `h-ray`.

### 4.2.4 Code Definitions

The following meta-level definitions describe GPL code fragments that are generated for the key parts of the construction. The input graph is called $G$, and is assumed to be globally available to all the code generators.

In the following definitions, it should be understood that $u$ and $v$ represent syntactic tokens that are substituted into the definition to produce the expression that is eventually evaluated. Strictly speaking, these syntactic tokens should be distinguished from the abstract graph nodes that they name. The operator $\bigoplus$ represents the in-place insertion of a concatenation of all its given terms. For example the expression (foo $\bigoplus_{i \in [1,3]} i$ bar) represents the expression (foo 1 2 3 bar).

**Definition 4.13** *Given a specified graph $G$, for a node $u \in V(G)$, and an integer $l$, let* $\textsc{DefNode}[u, l]$ *denote the code fragment:*

```
(define-growing-point (def-node-u)
  (material node u)
  (size 1)
  (actions
   (secrete l u-long-p)
   (secrete 2 u-short-p)
   $\bigoplus_{v:uv\in E(G), v\prec u}$ (start-gp nn-segment:u-v 0)
   $\bigoplus_{v:uv\in E(G), u\prec v}$ (connect (start-gp nn-segment:u-v 0)
                                              (start-gp col-$c_{uv}$-segment:u-v))
   (terminate)))
```

*where $\prec$ is some total ordering on the vertices, say the lexicographic ordering of the coordinates for $v \in V(G)$. $c_{uv}$ denotes the precomputed colour of edge $uv$.*

The exact meaning of $\prec$ is not important, so long as it is a total ordering. That way only one of the nodes at the endpoints of an edge will initiate a path representing that edge (i.e. evaluate `(start-gp col-`$c_{uv}$`-segment:`$u$-$v$`)`).

**Definition 4.14** *Let* NN-SEGMENT*$[u, v, s]$ denote the following growing point definition:*

```
(define-growing-point (nn-segment:u-v length)
  (material node u)
  (size 1)
  (tropism (ortho+ v-long-p))
  (avoids $\bigoplus_{w\in V(G)-\{u,v\}}$ w-short-p)
  (actions
   (secrete 2 u-short-p)
   (when ((> length s) (terminate))
         (default (propagate (+ length 1))))))
```

In the following, $C$ refers to the set of edge colours determined by step 2 of the construction.

**Definition 4.15** *Let* SEGMENT*$[c, u, v]$ denote the following growing point definition:*

```
(define-growing-point (col-c-segment:u-v)
  (material edge u v)
  (tropism (ortho+ v-long-p))
  (avoids $\bigoplus_{c'\in C-\{c\}}$ colour-c'
          $\bigoplus_{w\in V(G)-\{u,v\}}$ w-short-p)
  (actions
   (secrete 2 colour-c)
   (when ((sensing? (node v)) (terminate))
         (default (propagate)))))
```

Since trajectories representing edges are discernible only when their material labels are spaced at least 1 step-size unit apart, we make all poison pheromones be secreted for an extent of 2 step-size units.

### Code definition for Generating the Lattice

We now provide the essential GPL code definitions necessary to place the vertices, and we show that all the necessary conditions for proper vertex placement are met. For a vertex, $v \in V(G)$, $x(v)$ and $y(v)$ represent the $x$ and $y$ coordinates of $v$ as computed by step 1.

**Definition 4.16** *Let* V-RAY*[x] denote the following GPL definition:*

```
(define-gp (v-ray-x spacing length)
  (material v-stuff)
  (tropism (and (ortho- row-p) (ortho- v-ray-p)))
  (actions
   (secrete+ (- grid-size 1) v-ray-p)
   (when ((< spacing 1)
          (when ((> length h) (terminate))
                ⊕_{v∈U_x}((= length y(v))
                 (start-gp def-node-v)
                 (propagate (- spacing 1) length))
                (default (propagate grid-size (+ length 1)))))
         (default (propagate (- spacing 1) length))))
```

*where $U_x \subset V(G)$ is the set of vertices on a vertical lattice line (i.e. having the same $x$-coordinate), as determined by step 1 of the construction. The value of $h$ is set to the maximum $y$-coordinate of the vertices supplied in $U$.*

The code generated by V-RAY$[x]$ is used to produce the vertical grid line at the given $x$-coordinate. The principle behind V-RAY$[x]$ is very simple: a ray-like growing point keeps track of how many steps it has taken so far. If it arrives at a location that would be a lattice point, it checks the value of the $y$-coordinate to see if a vertex belongs there. If such a vertex does belong at that location, then the appropriate `def-node` growing point is invoked.

Each growing point produced by V-RAY$[x]$ is named with the value of the common $x$-coordinate to all the vertices that it establishes. So we define a dispatching growing point, `dispatch-v-ray`, that invokes `v-ray-`$x$ when given $x$ as an argument. Of course if $U_x$ (as defined above) is empty, then it simply does nothing.

```
(define-gp (dispatch-v-ray x)
  (actions
   (when ⊕_{n:U_n≠∅}((= x n) (start-gp v-ray-x 0))
         (default (halt)))))
```

We will also define `hray`, a growing point for spacing the vertical lines appropriately. The growing point, `dispatch-v-ray` provides an interface for `h-ray` to conveniently invoke each `v-ray-`$x$ growing point.

```
(define-gp (h-ray length count turn)
  (material h-stuff)
  (tropism (ortho- h-ray-p))
  (actions
   (secrete+ 3 h-ray-p)
   (secrete h_m row-p)
   (when ((< length 1) (terminate))
         ((< count 1)
          (when ((= turn 0)
                 (connect (start-gp v-starter-a)
                          (start-gp dispatch-v-ray length))
                 (start-gp make-intersection-b)
                 (propagate (- length 1) grid-size 1))
                (default
                  (connect (start-gp v-starter-b)
                           (start-gp dispatch-v-ray length))
                  (start-gp make-intersection-a)
                  (propagate (- length 1) grid-size 0))))
         (default (propagate length (- count 1) turn)))))
```

Here $h_m$ is the largest $y$-coordinate among all the vertex coordinates. The idea behind `h-ray` is that a long range pheromone, `row-p` is secreted to direct the growth of `v-ray`, which is invoked at regular intervals. The secretion of `row-p` by `h-ray` alone is not enough to enforce a consistent direction for each vertical line drawn. For simplicity we assume, as a part of the initial conditions specifications, that some `row-p` has been secreted along a line parallel and near to the x-axis, so that the "up" direction is properly determined.

In order to establish a short length of the `h-ray` trajectory before invoking `v-ray`, we use serializers `v-starter-a` and `v-starter-b`. These serializers are simply growing points that grow towards the previously established lattice point to terminate, and thereby establish that location as the place of invocation of `dispatch-v-ray`. We need two types so that adjacent lattice points do not secrete the same serializer pheromone (and confuse the serializer growing points). The definitions for the serializers are as follows.

**Definition 4.17** *Let* V-STARTER*[type] denote the definition:*

```
(define-gp (v-starter-type)
  (tropism (ortho+ intersection-type-p))
  (actions
   (when ((sensing? intersection) (terminate))
         (default (propagate)))))
```

**Definition 4.18** *Let* MAKE-INTERSECTION*[type] denote:*

```
(define-gp (make-intersection-type)
  (material intersection)
  (actions
   (secrete (+ grid-size 1) intersection-type-p)
   (terminate)))
```

The constant `grid-size` defines the spacing (in step-size units) between the grid lines that will be used to support the vertices. The grid spacing determines the minimum angle that is achievable between edges. The larger the grid spacing, the smaller the angle that can be represented on the grid. However, increasing the grid spacing also makes the edges longer, thereby requiring more time to render because of the wider secretions and increased number of locations per edge trajectory. So there is a tension between those two factors in choosing the size of the grid spacing. The value of the grid-spacing is determined by the highest ratio of $s(uv)$ to $l(uv)$ over all edges $uv$ in the graph.

### 4.2.5 Discussion of the Main Result

Before we get into the details of the argument for why the code should work as specified by Proposition 4.2, we state and prove a useful lemma about the locations of the regions representing the vertices of $G_P$.

**Lemma 4.2 (Unique Vertex Lemma)** *For each vertex $u \in V(G)$, there is a single point $s_u \in D$ for which $\mathcal{Q}($*`(start-gp def-node-`*$u$*`)`*,$s_u)$.*

**Proof:** Let $x_u$ be the $x$-coordinate of an arbitrary vertex $u$. Then by step 8 of the construction, there is a growing point called `v-ray-`$x_u$, which invokes `def-node-`$u$. From the definition of V-RAY$[x_u]$, `def-node-`$u$ is invoked at most once from within the body of `v-ray-`$x_u$. Now each V-RAY$[\cdot]$ growing point is invoked only once by `hray`, which in turn is invoked once by the entry point. It therefore follows that `def-node-`$u$ is invoked at most once during the entire execution of the program.

Since no growing points got stuck, we know all the V-RAY[·] growing points terminated normally. It follows that each `def-node-`$u$ growing point was invoked at least once. We therefore conclude that each `def-node-`$u$ growing point was invoked exactly once, implying that there is a unique point at which that invocation occurred.

■

**Corollary 4.1.2** *For each vertex $u$, $\mathcal{P}$(`def-node-`$u$, $s$) is empty unless $s = s_u$.*

**Proof:** The growing point `def-node-`$u$ is stationary (i.e. has no tropism attribute) and so it can never be invoked by a `propagate` command. Since $s_u$ is the only point for which $\mathcal{Q}$(`(start-gp def-node-`$u$`)`,$s_u$) then by the definition of $\mathcal{P}$, for any $s \neq s_u$, $\mathcal{P}(s)=\emptyset$. ■

### Proof of the Main Proposition

**Proof:** Suppose we evaluate the code from our construction step and no growing points got stuck (if any got stuck, then we are trivially done, because a stuck growing point can always report back to its originating location to decide that the algorithm failed).

Let $\mu$ be the resulting labeling from the execution, and $G_\mu$ the label induced subgraph of $\mu$. We need to show that it is possible to obtain a well-formed $G_\mu$ (i.e. no growing points got stuck) and when we do, it resembles the given planar graph $G_P$.

Define the map $f : V(G_P) \cup E(G_P) \mapsto 2^{V(G_\mu)}$ such that

$$f(u) = \{p \in D \mid \{\texttt{node}, u\} \subseteq \mu(p)\}$$

$$f(uv) = \{p \in D \mid \{\texttt{edge}, u, v\} \subseteq \mu(p)\}$$

We shall now show that $f$ satisfies the conditions stated for the map described in Definition 4.11.

**Claim 4.2.1** *$\forall x \in V(H), f(x)$ is a connected component in $G$.*

**Proof:** The only growing points that label particles `node` are those generated by DEFNODE[] and NN-SEGMENT[]. So for an arbitrary node $u \in V(G_P)$

$$f(u) = (\bigcup_{s \in S} \mathcal{P}(\texttt{def-node-}u,s)) \bigcup (\bigcup_{s' \in S'} \mathcal{P}(\texttt{nn-segment:}u\texttt{-}v,s')) \qquad (4.1)$$

where

$$S = \{s \in D \mid \mathcal{Q}(\texttt{(start-gp def-node-}u\texttt{)},s)\}$$

and

$$S' = \bigcup_{v:uv \in E(G)} \{s \in D \mid \mathcal{Q}(\texttt{(start-gp nn-segment:}u\texttt{-}v\texttt{ 0)},s)\}$$

By Theorem 4.1 each of the terms of each set union on the right hand side of Equation 4.1 is a connected component. Now for each vertex $v$ at the other end of an edge incident to $u$, the only point of invocation of `nn-segment:`$u$`-`$v$ is in the body of `def-node-`$u$. So in fact, $S' \subseteq S$. From corollary 4.1.2, $S$ contains only one point. So the right hand side of Equation 4.1 reduces to yield:

$$f(u) = \mathcal{P}(\texttt{def-node-}u, s_u) \bigcup \mathcal{P}(\texttt{nn-segment:}u\texttt{-}v, s_u)$$

where $s_u$ is the unique point in $D$ where `def-node-`$u$ was evaluated (as used in the statement of the Unique Vertex Lemma. It follows immediately that $f(u)$ is a connected component. ■

**Claim 4.2.2** *$\forall uv \in E(H), f(uv)$ is a connected component in $G$.*

**Proof:** The only growing points that label particles `edge` are those in the SEGMENT$[\cdot,\cdot,\cdot]$ family. Therefore for any edge $uv \in E(G_P)$,

$$p \in f(uv) \Leftrightarrow \exists s \in D : p \in \mathcal{P}(\texttt{col-}c_{uv}\texttt{-segment:}u\texttt{-}v, s).$$

So $f(uv)$ is a union of connected components by Theorem 4.1. By construction, there is exactly one point of invocation (in the code) for `col-`$c_{uv}$`-segment:`$u$`-`$v$, and that is in the body of `def-node-`$u$. So by the unique vertex lemma (Lemma 4.2) $f(uv)$ is not empty and it is a single connected component. ■

By claims 4.2.1 and 4.2.2 $f$ satisfies the first condition for the map described in definition 4.11.

**Claim 4.2.3** *$u \in V(H)$ incident to $e \in E(H) \Leftrightarrow f(u) \cup f(e)$ is a connected component of $G$.*

**Proof:** Since `col-`$c$`-segment:`$u$`-`$v$ is invoked at the point of termination of `nn-segment:`$u$`-`$v$ then that point, call it $p$, belongs to both $f(u)$ and $f(uv)$. To see that $f(v) \cup f(uv)$ will also be a connected component, observe that there is a single point $q$ such that $\mathcal{Q}$(`(secrete` $l_v$ $v$`-long-p)`,$q$) and $\mathcal{Q}$(`(start-gp nn-segment:`$v$`-`$u$ `0)`,$q$), since both expressions are in sequence in the body of `def-node-`$v$ which is evaluated in order to establish the region for $v$. Also by construction, we have that $l_v \geq d(u, v)$. Fourthly, we have from the first sentence that $\mathcal{Q}$(`col-`$c_{uv}$`-segment:`$u$`-`$v$,$p$). So by the segment lemma, we have that $p$ and $q$ are in a single connected component. So we have the forward direction of the second condition.

From the vertex placement phase of the constructions, the spacing of the vertices ensures that for any two vertices $u, v$ in $V(G_P)$, the points of evaluation of DEFNODE$[u, l_u]$ and DEFNODE$[v, l_v]$ are not within 2 step-sizes of each other. The avoids clause of NN-SEGMENT$[u, v, s(uv)]$ ensures that nodes do not spread into each other. So $f(u) \cup f(v)$ is not connected in $G_\mu$.

The colouring of the growing points generated by `col-`$c$`-segment:`$u$`-`$v$ guarantees that they do not come within one step-size of each other. So for any two edges $e_1, e_2 \in E(G_P), f(e_1) \cup f(e_2)$ is not connected in $G_\mu$.

It follows that for $x, y \in (V(G_P) \cup E(G_P))$ if $f(x) \cup f(y)$ is connected in $G_\mu$ then one of $x$ and $y$ is a vertex and the other is an edge in $G_P$. Without loss of generality, let $y$ be the edge. It remains to be shown that $x$ is incident to $y$ in $G_P$.

Suppose the contrary, that is, suppose that $f(x) \cup f(y)$ is a connected component and no growing points were stuck, but $x$ is not incident to $y$ in $G_P$. The growing points `def-node-`$x$

| Growing Point Family | Instance Size | Total Size |
|---|---|---|
| $\textsc{DefNode}[u, l_u]$ | $O(\deg(u))$ | $O(\|E(G)\|)$ |
| $\textsc{nn-segment}[u, v, s(uv)]$ | $O(\|V(G)\|)$ | $O(\|V(G)\|^2)$ |
| $\textsc{segment}[u, v]$ | $O(\|C\| + \|V(G)\|)$ | $O(\|E(G)\| * (\|C\| + \|V(G)\|))$ |
| $\textsc{v-ray}[x]$ | $O(\|U_x\|)$ | $O(\|V(G)\|)$ |

Table 4.1: The code sizes for the growing point definitions used in the construction.

and those generated by $\textsc{nn-segment}[x, \cdot, \cdot]$ secrete a short range pheromone, extending for 2 step-size units, that disables any $\textsc{segment}[\cdot, \cdot]$ growing point not of the form $\textsc{segment}[x, \cdot]$ or $\textsc{segment}[\cdot, x]$.

In particular, say $y = uv, x \neq u$ and $x \neq v$, since no growing points were stuck, then every location labeled by $\textsc{segment}[u, v]$ is at least 2 step-size units from any point labeled $\{\texttt{node}, x\}$. By Definition 4.10 of label induced subgraph, points labeled by $\textsc{segment}[u, v]$ are not connected to those labeled by `def-node-`$x$ or by $\textsc{nn-segment}[x, \cdot, \cdot]$ in $G_\mu$. This contradicts our hypothesis.

So we conclude that if $f(x) \cup f(y)$ is connected in $G_\mu$ then $x$ is incident to $y$ in $G_P$, thereby proving the right-to-left direction of the second condition of definition 4.11. ■

Since $f$ satisfies the conditions of Definition 4.11 then if we obtain a $G_\mu$, it resembles $G_P$. By the segment lemma, the combination of $\textsc{nn-segment}[u, v, s(uv)]$, $\textsc{segment}[u, v]$ and $\textsc{nn-segment}[v, u, s(vu)]$ draws each edge $uv$ of $G_P$. It therefore follows that the planar interpretation of $\mu$ draws the graph $G_P$.

■

## 4.3 Analysis of Resource Usage

All the steps of the vertex placement phase are performed off-line and can be done efficiently. If $V$ is the set of vertices, the time to find grid coordinates for $V$ is $\mathcal{O}(|V| \log |V|)$ (see [16]), so the setup phases of the construction are efficient. We now discuss our real concern: how the particle resources are consumed.

### 4.3.1 Static Memory

The total program size can be computed from Table 4.1.

Since the graph is planar, $|E(G)| \leq 3|V(G)| - 6$ (see [45]). Also, we now know that the graph is 4-colourable. So, we can regard $|C|$ as a constant, and we can regard $|E(G)|$ as $O(|V(G)|)$. The size of `dispatch-v-ray` is dependent upon the number of $x$-coordinates that are used by all the vertices taken altogether. But we do not include it in the table because its size is strictly upper-bounded by $|V(G)|$ and it therefore does not consume any more space than $\textsc{v-ray}[\cdot]$ does. The row for $\textsc{v-ray}[\cdot]$ can be taken to include all the auxilliary growing points that help establish the lattice. Summing up the right hand column of Table 4.1, we see that the construction takes $O(|V(G)|^2)$ space to store the program.

### 4.3.2 Dynamic Memory

No growing point uses more than one parameter, so very little space is consumed that way. The number of materials assigned to any one node is also constant since particles are labeled as either nodes or edges, and never with more than one such label. All of the dynamic space goes towards maintaining the pheromone levels during the execution of the program. At worst, every long-range pheromone will be detectable by some vertex region, so there is a $O(|V(G)|)$ contribution of cost there. The short range pheromones that impact any particular site are few since nodes secrete their own short range pheromone around their region, thereby keeping others out; and the only short range pheromones that can be tolerated by the edges are the ones they secrete themselves. So the overall dynamic space required by each particle is $|V(G)|$.

### 4.3.3 Time

The determining factors for the total time taken to draw the graph are: $\gamma$; $l$, the maximal value of $l_u$ over all nodes $u$; and the sum of $l(uv)$ over all the edges $uv \in E(G)$. Assuming that secretions take time linear in their extents, the total time spent in secretions is $O(\gamma\, l)$. The secretion from the last node established is delayed by at most $O(|V(G)|)$ time units since the particles are laid out on the grid sequentially (in order of $x$-coordinate).

The time taken to draw the edges (in parallel) is $\max_{uv \in E(G)} \gamma\, l(uv)$ which is no worse than $O(\gamma\, l)$. Therefore, the total amount of time required to generate the graph is $O(\gamma\, l + \gamma\, l + |V(G)|) = O(\gamma\, l)$. Theorem 4.2 (presented below) tells us that $\gamma = O(|V(G)|)$. Also, since the value of $l$ is upper bounded by the lattice diagonal, $l = O(|V(G)|)$. So, stated in terms of the graph properties, the time required to draw the given graph $G$, is $\mathrm{O}(|V(G)|^2)$.

### 4.3.4 Domain Space

The amount of domain space consumed is ultimately determined by the size of the grid-spacing used in the algorithm. Since the grid-spacing is computed by comparing $s(uv)$ with $l(uv)$, the actual value used will depend on the layout computed for the graph. The following theorem places an upper bound on the grid spacing.

**Theorem 4.2** *The grid spacing is at most* $20(|V(G)|-1)$ *in order to accommodate a spacing of at least 2 step-size units between the trajectories of the* SEGMENT*[·, ·, ·] growing points.*

**Proof:**

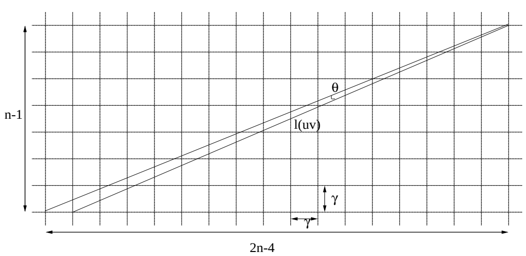


Figure 4-4: Smallest possible angle attainable on an $n-1$ by $2n-4$ grid.

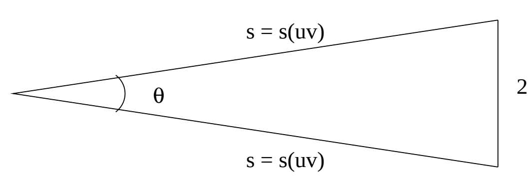


Figure 4-5: Computation of $s(uv)$ given $\theta$

Let $\gamma$ be the grid spacing. Then the actual length of the line representing edge $uv$ becomes $\gamma\, l(uv)$. In order to ensure proper spacing between the edge-representing growing point trajectories, we need that for every $uv \in E(G)$, $s(uv) < \frac{1}{2}\gamma\, l(uv)$. The value of $s(uv)$ is determined by the size of the angle between two adjacent edges incident at node $u$. So to find the worst case value of $\gamma$, we suppose that we have the smallest possible angle at a node bounded on one side by the shortest possible edge. Figure 4-4 shows the relevant dimensions involved in the computation. For economy, we abbreviate $s(uv)$ by $s$, and $|V(G)|$ by $n$.

The angle $\theta$ represents the smallest angle obtainable on an $n-1$ by $2n-4$ grid, the size grid required to layout the given graph $G$. By the cosine rule, $\theta$ is given by:

$$\cos(\theta) = \frac{2(n-1)^2 + (2n-5)^2 + (2n-4)^2 - 1}{2\sqrt{((n-1)^2 + (2n-5)^2)((n-1)^2 + (2n-4)^2)}}$$

Given $\theta$, the value of $s(uv)$ is the length of the symmetric side of an isosceles triangle whose base of length 2 subtends the angle $\theta$ (see Figure 4-5). Again, by the cosine rule, we have:

$$\cos\theta = \frac{s^2 - 2}{s^2}$$

Equating the two expressions, and solving for $s$, we get:

$$s^2 = \frac{4ab}{1 - (a-b)^2}$$

where $a$ and $b$ are given by:

$$\begin{aligned} a &= \sqrt{(n-1)^2 + (2n-4)^2} \\ b &= \sqrt{(n-1)^2 + (2n-5)^2} \end{aligned}$$

Multiplying the numerator and denominator of $s^2$ by $2ab + (a^2 + b^2 - 1)$, we get the equivalent statement:

$$s^2 = 4\frac{ab((a+b)^2 - 1)}{2(a^2 + b^2) - (a^2 - b^2) - 1}.$$

Substituting in the denominator for $a$ and $b$, we get

$$s^2 = \frac{ab((a+b)^2 - 1)}{(n-1)^2} \tag{4.2}$$

Now observe that

$$\begin{aligned} a &= \sqrt{(n-1)^2 + (2n-4)^2} \\ &= \sqrt{(n-1)^2 + 4((n-1)-1)^2} \\ &= \sqrt{5(n-1)^2 - 8(n-1) + 4} \end{aligned}$$

and

$$\begin{aligned} b &= \sqrt{(n-1)^2 + (2n-5)^2} \\ &= \sqrt{(n-1)^2 + (2(n-1)-3)^2} \\ &= \sqrt{5(n-1)^2 - 12(n-1) + 9} \end{aligned}$$

So clearly both $a$ and $b$ are upper bounded by $\sqrt{5}(n-1)$ for $n \geq 2$. Substituting these bounds into Equation 4.2, we obtain an upper bound for $s^2$

$$s^2 < 5 \cdot (20(n-1)^2 - 1)$$

and so

$$s < 10(n-1)$$

In the worst case, such an $s(uv)$ would be computed for an edge of length only 1 step-size (i.e. $l(uv) = 1$). In order to make $s(uv) < \frac{1}{2}\gamma\, l(uv)$, we set

$$10(n-1) = \frac{1}{2}\gamma \cdot 1$$

implying that at worst $\gamma$ is $20(n-1)$. ■

Theorem 4.2 tells us that if we happen to compute a bad layout for the given graph, that the area required to ensure that our GPL program will have a chance to succeed can be quite large. However, we see that we never require any more than quadratic length in the number of vertices for each grid dimension, even with the straightforward analysis presented. It may be the case that the bound on $\gamma$ could be reduced by applying some cleverer analysis, but it is unlikely to be as low as a constant since there is no limit on the degree of the graph, which is ultimately responsible for the amount of space required to render the graph.

## 4.4 The Effects of Random Distributions

In general, there are three important concerns that are directly affected by the geometry of the GPL domain. First, we care whether growing points can propagate at all, so the domain must be connected. Second, we care that growing point propagation occurs in the intended direction. For the domain, that means that from the location of any particle, every direction is represented (within some tolerance) by at least one neighbour lying in that direction. Third, we would like that when pheromone levels are monotonic decreasing in their shortest-path-lengths from the source of the secretion, that they are also monotonic decreasing in their Euclidean distances from the source. Otherwise, the patterns we generate will likely be unrecognizable.

The density of the particles in the domain affects all three of our concerns. There are two aspects of the density that can have negative effects: very low average densities or relatively high variances in the density. In a way, it is misleading to identify the density of the domain as the principal property of concern. It turns out that the truly important properties are the mean and variance of the neighbourhood size; these are closely related to the average density of the domain.

Kleinrock and Silvester [28] and Chandler [10] showed that low neighbourhood sizes are likely to result in disconnected domains. Another bad feature of low densities is that there are fewer directions represented from each particle. That means that a growing point is more likely to get stuck when trying to propagate because there are fewer prospective successor sites. We cannot say much more about the effect of the domain on the ability for growing points to propagate without specific programs in mind, so we shall now leave the discussion on our first concern.

In the rest of the section, we first provide a brief framework for analysing the properties of the domain. Then we discuss, within that framework, the effect the domain has on the ability for growing points to propagate in the "correct" directions; and we discuss the domain constraints that are thereby implied. Finally, we show that the variance in the neighbourhood sizes affects how well correlated pheromone levels are with their Euclidean distances from the source of a secretion.

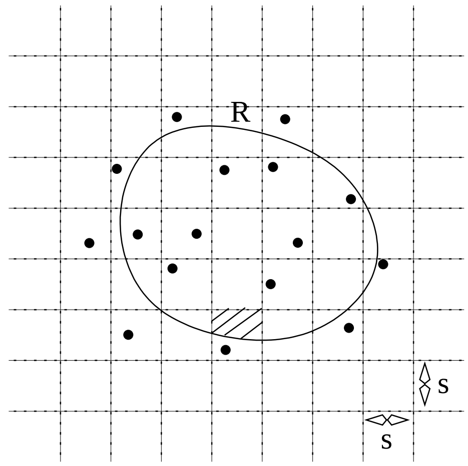


Figure 4-6: An arbitrary region in the GPL domain, with one of its cells shaded.

### 4.4.1 Setting up the Domain

In much of the literature on random Euclidean graphs, the distribution of particles is assumed to be Poisson [28, 25, 49, 50], but here we use a different approach. The principal motivation being that a Poisson distribution necessarily has a fixed variance, equal to its mean. Since we want to show that these two properties of the neighbourhood sizes have different effects, we have to use a distribution that would allow the variation of one parameter while keeping the other constant.

We use a binomial distribution with a user-supplied parameter $p$ for placing the particles in the domain. The user supplies the desired expected number of particles, the neihgbourhood radius, as well as the dimensions of the rectangle in which the particles are contained. The rectangle is then divided into small squares in such a way that each square contains at most one particle. A particle occupies a square as a Bernoulli trial with parameter $p$. Given that a square contains a particle, its coordinates are chosen from a uniform distribution inside the square.

If the desired expected number of particles is $m$, then the total number of squares, $N$, is given by $m/p$. Since $N$ must be an integer, we choose it to be the ceiling of $m/p$. The overall process is binomial, so the variance of the total number of particles is $Np(1-p)$ (or $m(1-p)$ expressed in terms of user-supplied parameters). The advantage of using this binomial process is that we can easily vary the nature of our distributions by adjusting the value of $p$. For example, if we want a Poisson distribution, then we set $p$ to a very small value (say 0.01), if we want a relatively even distribution (i.e. low variance) then we set $p$ to a value close to 1.

### 4.4.2 Characterizing the Domain

The fact that the distribution is binomial allows us to describe global parameters such as the distribution mean and variance. However, we are actually interested in the local properties of the distribution, namely the expected neighbourhood size and its variance. It turns out that we can make useful statements about these local parameters.

Figure 4-6 shows a section of the domain with squares of side $s$ highlighted, and a few particles already placed. Suppose we are given a region, $R$, in the domain with a fixed location, shape and size. Let $n_R$ denote the number of particles in $R$, and let $A(R)$ denote its area. We have the following theorem regarding the expected value of $n_R$:

**Theorem 4.3** *The expected number of particles in a given closed region $R$ is $pA(R)/s^2$.*

**Proof:** We refer to any square or portion of a square in $R$ as a cell. For any cell, $c$ in $R$, let $x_c$ be the number of particles in $c$ (i.e. either 0 or 1). Since the Bernoulli trials performed in the setup of the domain are independent of the uniform coordinate selection, the probability mass function for $x_c$ is given by:

$$\begin{aligned} \Pr[x_c = 0] &= 1 - \tfrac{pA(c)}{s^2} \\ \Pr[x_c = 1] &= \tfrac{pA(c)}{s^2} \end{aligned}$$

Now we can define $n_R$ as

$$n_R = \sum_{c \in R} x_c \tag{4.3}$$

Since $n_R$ is a linear function of the $x_c$'s,

$$\begin{aligned} E[n_R] &= E\left[\sum_{c \in R} x_c\right] \\ &= \sum_{c \in R} E[x_c] \\ &= \sum_{c \in R} \frac{pA(c)}{s^2} \\ &= \frac{pA(R)}{s^2} \end{aligned}$$

■

We can also say something about the variance of the number of particles in the region $R$.

**Theorem 4.4** *The variance of $n_R$ is given by*

$$\frac{pA(R)}{s^2} - \frac{p^2}{s^4} \sum_{c \in R} A(c)^2$$

**Proof:** Since the $x_c$'s are independent, the variance of $n_R$ is the sum of the variances of the $x_c$'s.

$$\begin{aligned} V[n_R] &= V[\sum_{c \in R} x_c] \\ &= \sum_{c \in R} V[x_c] \\ &= \sum_{c \in R} \frac{pA(c)}{s^2}(1 - \frac{pA(c)}{s^2}) \end{aligned}$$

$$= \frac{pA(R)}{s^2} - \frac{p^2}{s^4}\sum_{c\in R} A(c)^2$$

■

The second term of the final equation varies with the geometry of the region R, so it is not possible to reduce its expression further in the completely general case. This point is actually quite important because of the following observation. The expected value is independent of the geometry of the region $R$. So if we want to characterise the effects of irregularity of the domain on the behaviour of our growing points, we must have some other parameters which we use to describe the distribution. Variance can be one of them (though not necessarily the only other one) because its value depends upon the geometry.

Although we are hard-pressed to reduce the variance to more meaningful terms, we can obtain a lower bound. We do this by finding a useful upper bound on the second term in the final equation.

$$\frac{1}{s^4}\sum_{c\in R} A(c)^2 = \sum_{c\in R}\left(\frac{A(c)}{s^2}\right)^2$$

Since

$$\forall c \in R : 0 < \frac{A(c)}{s^2} \leq 1,$$

$$\forall c \in R : \left(\frac{A(c)}{s^2}\right)^2 \leq \frac{A(c)}{s^2}$$

Therefore,

$$\begin{aligned}\frac{1}{s^4}\sum_{c\in R} A(c)^2 &\leq \sum_{c\in R}\frac{A(c)}{s^2}\\ &= \frac{A(R)}{s^2}\end{aligned}$$

Hence,

$$\frac{pA(R)}{s^2}(1-p) \leq V[n_R] < \frac{pA(R)}{s^2} \tag{4.4}$$

Note that we could obtain a tighter upper bound. Observe that the set of cells internal to the region $R$ each contribute a value of 1 to the second term of equation 4.4. Therefore,

$$\frac{pA(R)}{s^2}(1-p) \leq V[n_R] < \frac{pA(R)}{s^2}(1-p) + P(R)$$

where $P(R)$ denotes the number of cells bounded by the perimeter of $R$. Later, we discuss best and worst case conditions on the variance of the neighbourhood size, for which inequalities 4.4 are good enough.

At this point we "take a step back" to recapitulate the expressions derived up to this point in more accessible terms. To get an intuitive grasp for the meanings of these expressions, let us try to express $s$ in terms of the user-supplied parameters. Since $N$ is the number of squares, then each square has an area of $hw/N$, where $h$ and $w$ are the user-supplied height and width of the domain rectangle. So we can write $s^2 = hw/N$. Since $N = \lceil m/p \rceil$, it is only approximately correct to say $s^2 = phw/m$, but then we would see that

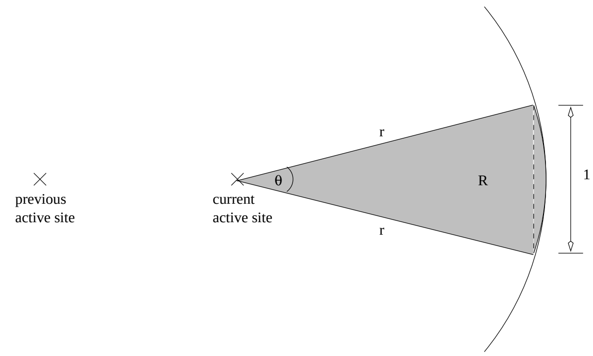


Figure 4-7: The region of tolerance for representing the direction established by the locations of the previous and current active sites of a growing point.

$p/s^2$ is approximately $m/hw$ which is familiar to us as the average density of particles in the domain. Viewed in this way, we see that the expressions for the mean and variance are, in fact, intuitively plausible. However with the exact formulations of both the mean and variance, we do not have to rely on our intuition for answering questions such as: "What parameters should we use to ensure that a ray growing point is likely to propagate on each step in a nearly straight line?"

### 4.4.3 Domain Requirements for Good Propagation

In order to propagate a growing point in the correct direction, a particle must have at least one neighbour lying in that direction. By defining a tolerance region within a neighbourhood for representing a direction, we can apply Theorem 4.3 and Theorem 4.4 to obtain requirements on the domain parameters so that with high probability no such region will be empty.

Figure 4-7 shows the region, $R$, of tolerance for representing a particular direction within a particle's neighbourhood. We normalise distances so that the diameter of a particle is 1. In the figure, $r$, represents the radius of the neighbourhood region (also known as the step-size), and we have chosen to make the tolerance be one particle diameter at the circumference of the neighbourhood region, centred on the ray towards the represented direction. The region here is a sector of the neighbourhood with angle $\theta = 2\sin^{-1} 1/(2r)$, but we could have restricted the region further to contain only the points near the circumference of the neighbourhood since that is sometimes more appropriate.

The area of $R$ is $\frac{1}{2}r^2\theta = r^2\sin^{-1}(\frac{1}{2r})$ so if $n_R$ represents the number of neighbours in $R$, then

$$E[n_R] = \frac{p}{s^2}r^2\sin^{-1}(1/2r)$$

$$\frac{p}{s^2}(1-p)r^2\sin^{-1}(1/2r) \leq V[n_R] < \frac{p}{s^2}r^2\sin^{-1}(1/2r)$$

by Theorem 4.3 and Theorem 4.4.

Now we want to obtain the appropriate conditions that make it very unlikely that from any particle there is a direction that is not represented within that particle's neighbourhood. In other words, we want conditions that will make the chances of $R$ being empty very small.

Assuming the domain contains numerous particles, and therefore there are numerous wedges $R$, we can invoke the central limit theorem to conclude that the distribution of $n_R$ is approximately normal with mean $E[n_R]$ and variance $V[n_R]$. To be reasonably sure that there are no regions like $R$ that are empty, we want to make sure that 1 is at least two standard deviations from $E[n_R]$. From the Inequalities 4.4, we see that $V[n_R]$ and $E[n_R]$ are related, namely

$$(1-p)E[n_R] \leq V[n_R] < E[n_R]. \tag{4.5}$$

This allows us to write down a constraint on them that reduces to constraints on the original user-supplied parameters.

Let $\mu = E[n_R]$ and $\sigma = \sqrt{V[n_R]}$ be actual instances of the mean and standard deviation of $n_R$ that satisfy the constraint:

$$\mu - 2\sigma \geq 1$$

We know from Inequalities 4.5 that

$$(1-p)\mu \leq \sigma^2 < \mu$$

Rewriting the constraint, we get $(\mu-1)^2 \geq 4\sigma^2$. In the worst case, the variance is as large as possible. Substituting $\mu$ for $\sigma^2$, we obtain

$$(\mu-1)^2 \geq 4\mu$$

and solving for $\mu$, we see that $\mu \geq 3 + 2\sqrt{2}$. In the best case, the constraint becomes

$$(\mu-1)^2 \geq 4(1-p)\mu.$$

Again, solving for $\mu$, we obtain $\mu \geq 3 - 2p + 2\sqrt{(2-p)(1-p)}$.

Remembering that $\mu = pA(R)/s^2 = pr^2 \sin^{-1}(1/2r)$ for the sector, we see that if we are trying to choose $r$ when given all the other parameters, then we need $r$ to satisfy:

$$r^2 \sin^{-1}\frac{1}{2r} = \frac{s^2}{p}(3 - 2p + 2\sqrt{(2-p)(1-p)})$$

In more familiar (to the user), but approximate terms,

$$r^2 \sin^{-1}\frac{1}{2r} = \frac{hw}{m}(3 - 2p + 2\sqrt{(2-p)(1-p)})$$

We now interpret these results. If we want to be fairly confident (95%) that no direction sectors will be empty if $p$ is very low (i.e. the distribution looks like a Poisson distribution) then the expected number of neighbours within each sector should be at least 5.83 ($3 + 2\sqrt{2}$). On the other hand, when the value of $p$ is relatively high, then we can reduce that lower bound to a value very close to 1 (the value of $\mu$ approaches 1 as $p$ approaches 1). Remembering that a loose way of thinking of $\mu$ is as the product of the average density and the area of the sector, then we have a method for determining a "good" value for the communications radius, when given the values of $p$ and the average density. We stress here, that obtaining such a "good" value of the communication radius (i.e. step-size) does not guarantee that our growing points will propagate correctly. Our analysis yields necessary conditions in the general case (i.e. when we do not know what kind of program we are going to execute, but we want it to have a good chance of succeeding anyway), not sufficient ones.

Characterizing sufficient conditions appears to be a much harder problem.

### 4.4.4 Errors in Secretions

Up until now, we have considered how the geometry of the GPL domain can directly affect the behaviour of a growing point. We now investigate how the geometry of the domain indirectly affects growing point trajectories, namely by disrupting secretion patterns. We have already seen from Chapter 3 that our secretion patterns are monotonic decreasing in the shortest-path length from the source. It turns out that our secretion algorithm also tends to yield pheromone values that are inversely correlated with the Euclidean distance between the particle and the pheromone source. If we have secretions that are highly correlated with the Euclidean distances, then our geometric intuition about the outcomes of our programs become more accurate. In addition, striving to make pheromone levels sensitive to the Euclidean metric helps to refine those pheromone values within a single neighbourhood (thereby providing more choices for propagation). For the rest of this section, we shall attempt to characterise the properties of the domain that disrupt the correlation between the pheromone levels and their Euclidean distances.

#### The Secretion Algorithm

The pheromone levels in the secretion algorithm used for these experiments are computed as a function of the shortest-path-lengths of the particles. Specifically, a particle assigns itself the sum of all the values broadcast by its neighbours. The value broadcast by a particle is an exponential function of the path-length from the source. At the end of the secretion process, each particle at (shortest-path) distance $d$ from the source, has a pheromone value, $u(d)$, given by

$$u(d) = (N_{d-1}\alpha^{d+1} + N_d\alpha^d + N_{d+1}\alpha^{d+1})v_0 \tag{4.6}$$

where $v_0$ is the value broadcast by the source, $\alpha$ is a prespecified constant smaller than 1, and $N_k$ represents the number of neighbours at distance $k$ from the source.

The secretions were modeled under perfect conditions (i.e. all shortest paths and neighbour counts were computed directly from the domain, and not simulated). We did this because we wanted to focus on the effect that irregularities in the GPL domain had on the correlation of pheromone values with Euclidean distances. Furthermore, both the shortest path and neighbour counts can be computed with reasonable reliability using the secretion method outlined in Chapter 3. Therefore, although our results are obtained under controlled conditions, they could plausibly be observed under normal (amorphous) operating conditions.

Equation 4.6 shows that the pheromone levels can get quite large, especially if the pheromone is secreted for a large extent. So for the purposes of detection, we use the logarithm of the value computed from equation 4.6. In actual GPL evaluation, the reduced values allow pheromones to be compared in the same way (by differences) as other secretion implementations so that the implementations of tropism expressions do not have to be modified. We also used the logarithmically reduced values for visualizing the pheromone levels over the entire GPL domain.

Figure 4-8 shows the distribution of a pheromone after a secretion from a particle near the center for an extent of 5 step-sizes. Observe that although we use shortest-path-lengths as our metric, the shape of the secretion is very nearly circular! It is not obvious why this should be the case, and indeed, the shape of the same secretion on a regular square grid is

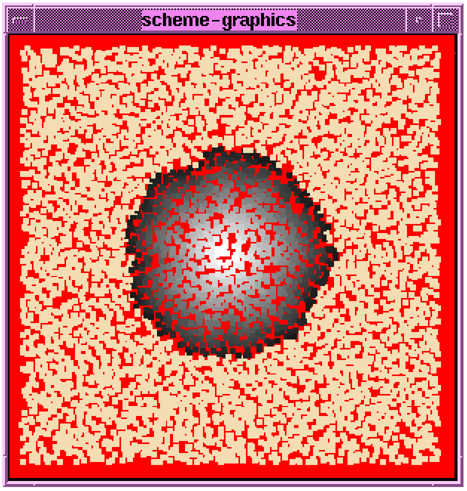


Figure 4-8: Secretion of extent 10. Darker greys indicate lower pheromone values

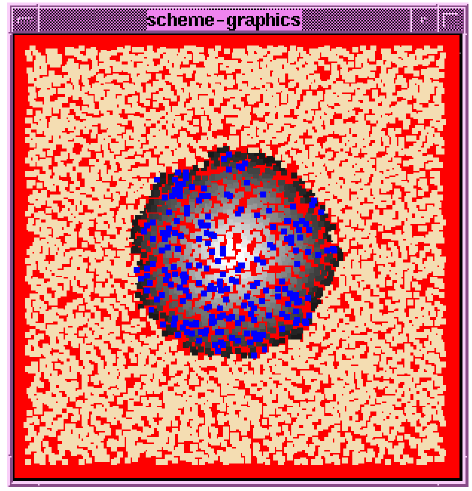


Figure 4-9: Same secretion of extent 10 but with errors highlighted in blue (dark grey if no colour).

certainly not circular. Vahidi and Wierman [49, 50] showed that for a Poisson distribution of points, the shape of all the reachable points of a random walk is approximately circular. The difference between their model and the distribution of points considered here is that in their model, each step of the walk was along an edge of the Voronoi diagram for the points. In the case of secretions, each step may be the equivalent of several edges in the Voronoi diagram.

In light of the apparent circularity of the secretion shown in Figure 4-8 and of the analysis presented by Vahidi and Wierman, we investigated the correlation between the pheromone levels and the Euclidean distance from the source. Note that a circular pattern of pheromone levels implies a correlation with the Euclidean distance from the source, but the converse is not true. For example, on a regular square grid, the secretions we produce are error-free in the sense that the pheromone levels are perfectly (negatively) correlated with the Euclidean distance. That is, all pairs of neighbouring particles have pheromone levels that correctly imply their Euclidean distances from the source. Yet, on such a regular lattice, the shape of the secretion is not circular. Figure 4-10 shows a scatter plot of the pheromone levels against the Euclidean distance from the source. From Figure 4-10 it seems that there is a strong linear (negative) correlation between the pheromone values and their distances from the source. We computed the linear correlation coefficient to be -0.944, so in fact the scatter plot is not misleading.

Figure 4-9 shows the same secretion as shown in Figure 4-8, but with the errors highlighted. There were 256 errors out of the 974 particles that detected the pheromone. That means that a strong correlation does not say much about the frequency of the errors. Since growing points work by comparing neighbouring values of pheromones, these errors of mismatched pairs of particles as described above, really do matter. These observations led us to investigate the conditions that affect the frequency of errors in secretions. We wanted to know what properties of the domain would affect the error frequency as well as the correlation between pheromone levels and Euclidean distances from the secretion source.

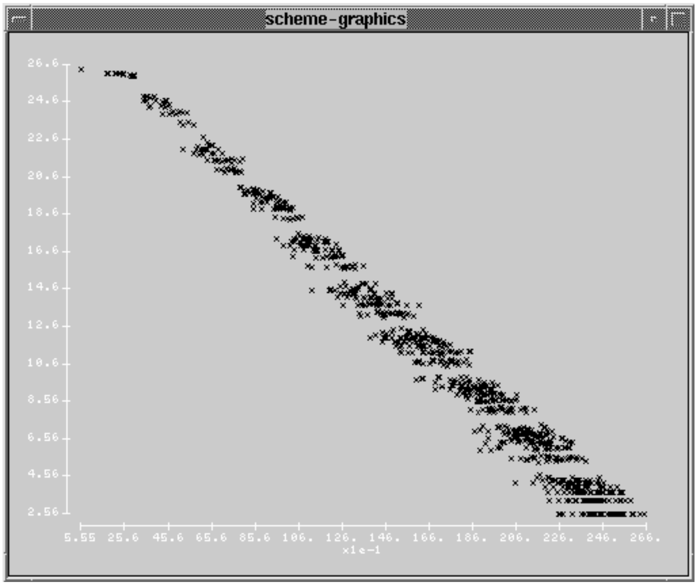


Figure 4-10: Scatter plot of secretion levels vs. Euclidean distance from the source

**The hypothesis**

We postulated that the frequency of secretion errors is dependent upon either the variance of the number of neighbours or the variance of the Euclidean distances between neighbours. We define an error to have occurred if two neighbouring particles have different phermone levels and the larger value belongs to the particle that is farther from the source. The variance of the number of neighbours was computed by tallying the neighbours of every particle in the domain. We computed the variance of the Euclidean distances by including the distance between the locations of every neighbouring pair of particles.

**Rationale:** Errors in pheromone levels arise because of shortest-paths whose lengths do not correlate with the Euclidean distance between their end points. Figure 4-11 shows an example of such a shortest path. We arrived at our postulate by observing that shortest-path lengths become bad indicators of the Euclidean distance between the end points when the path is not very constrained. In other words, if there is much room to relocate each particle along a path without changing the shortest path between the end points, then that path is likely to give rise to a secretion error. Based on this, we hypothesized that a large variance in the nearest neighbour distances would weaken the geometric constraints on the shortest paths. We guessed that the variance of the number of neighbours a particle has would also be high as a consequence, so we extended the hypothesis to include that property of the distribution as well.

**The Experiment**

- Setup multiple distributions as follows: With expected numbers of particles ranging from 1000 to 8000, vary step-size to maintain constant expected neighbourhood size across all distributions. For each distribution size, use multiple values of $p$, ranging

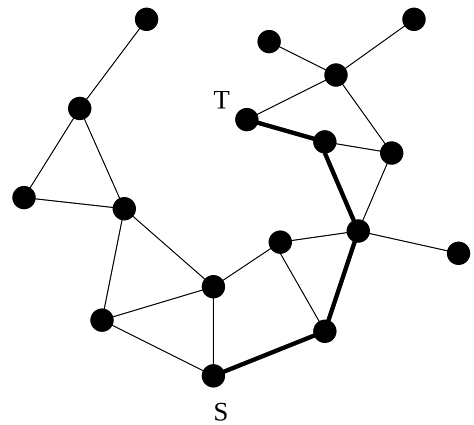


Figure 4-11: A shortest path that is likely to lead to errors in pheromone levels. All edges of the communication graph have been drawn, the bold lines represent the shortest path from the source $S$ to a node $T$.

from 0.1 to 0.9 in 0.1 increments. For each parameter $p$, generate $200(1-p)$ different distributions with the same parameters.

- For each distribution:

  1. Produce 20 secretions of extent 5 (step-size units), from random particles selected uniformly from an inner rectangle bordered by a frame of width 5.
  2. For each secretion, record the number of instances when there is a pair of particles whose pheromone values are incorrectly correlated with their distances from the source.
  3. Also record: the actual domain size, the average neighbourhood size, the variance of the neighbourhood size, the average neighbour separation, and the variance of the neighbour separation.
  4. Compute the average number of correlation errors.

**The Results**

We summarize most of our results in Table 4.2 and Table 4.3. Each table contains data from three parameter settings. Specifically, they each contain a summary of the measurements taken for each value of $p$ for distributions with expected sizes of 1000, 5000 and 8000 particles. The distributions of the other expected sizes produced data with almost identical qualities as those represented here in the tables.

The table contains numbers that are statistical summaries of the measurements taken for all the distributions for each value of $p$. The average neighbourhood size was determined for each distribution during the experiment, and then those means were themselves averaged to yield the numbers appearing in the column labeled "avg. nbd size." Similarly for each distribution, we measured the variance of the number of neighbours for each particle in the domain. Each value reported in the column labeled "var. nbd size" is the mean of all the variances that were calculated for each distribution with the indicated parameters

| p | #particles | | avg. nbd size | | var. nbd size | | #errors | |
|---|---|---|---|---|---|---|---|---|
| | mean | sd | mean | sd | mean | sd | mean | sd |
| .1 | 1024. | 32.23 | 13.27 | .4565 | 14.8 | 1.903 | 67.87 | 8.879 |
| .2 | 1006. | 27.19 | 13.1 | .3993 | 13.53 | 1.678 | 64.66 | 7.973 |
| .3 | 1009. | 27.52 | 13.05 | .3964 | 12.47 | 1.362 | 63.41 | 7.018 |
| .4 | 1037. | 21.08 | 12.91 | .2944 | 11.27 | 1.25 | 60.91 | 7.357 |
| .5 | 1015. | 23.27 | 12.86 | .3177 | 10.22 | .9729 | 59.39 | 6.197 |
| .6 | 1006. | 20.1 | 12.7 | .2631 | 9.104 | .6923 | 55.7 | 5.066 |
| .7 | 1011. | 17.5 | 12.67 | .2336 | 8.105 | .5494 | 54.81 | 3.837 |
| .8 | 1035. | 11.91 | 12.52 | .1424 | 7.033 | .443 | 52.19 | 4.362 |
| .9 | 1040. | 11.28 | 12.45 | .1489 | 6.139 | .3288 | 49.8 | 3.885 |
| .1 | 5019. | 76.07 | 13.69 | .2157 | 13.95 | .8772 | 67.08 | 5.174 |
| .2 | 5057. | 63.54 | 13.59 | .1892 | 12.64 | .7331 | 65.55 | 4.183 |
| .3 | 5076. | 62.39 | 13.5 | .175 | 11.29 | .6033 | 62.43 | 3.57 |
| .4 | 5024. | 58.66 | 13.38 | .172 | 10.01 | .4424 | 60.6 | 3.609 |
| .5 | 5103. | 50.22 | 13.29 | .1376 | 8.948 | .4448 | 58.58 | 3.295 |
| .6 | 5078. | 52.29 | 13.18 | .1378 | 7.804 | .333 | 56.16 | 3.54 |
| .7 | 5048. | 37.24 | 13.06 | .09884 | 6.67 | .2629 | 53.93 | 2.479 |
| .8 | 5119. | 27.7 | 12.98 | .09244 | 5.589 | .1941 | 51.53 | 2.953 |
| .9 | 5060. | 13.85 | 12.87 | .04654 | 4.61 | .1185 | 49.37 | 2.053 |
| .1 | 8008. | 89.19 | 13.75 | .1635 | 13.66 | .5887 | 67.37 | 4.347 |
| .2 | 8079. | 74.54 | 13.65 | .1344 | 12.32 | .5208 | 65.09 | 4.054 |
| .3 | 8064. | 71.61 | 13.55 | .1208 | 11.08 | .4709 | 62.83 | 3.791 |
| .4 | 8070. | 66.48 | 13.46 | .1146 | 9.803 | .3806 | 60.27 | 3.27 |
| .5 | 8078. | 58.44 | 13.38 | .09979 | 8.667 | .2625 | 58.85 | 3.3 |
| .6 | 8062. | 60.66 | 13.23 | .1041 | 7.477 | .2738 | 56.55 | 2.821 |
| .7 | 8018. | 52.11 | 13.16 | .09332 | 6.391 | .1754 | 53.94 | 2.635 |
| .8 | 8159. | 36.71 | 13.05 | .06889 | 5.322 | .1767 | 51.77 | 2.569 |
| .9 | 8121. | 29.06 | 12.96 | .04216 | 4.344 | .09204 | 48.8 | 1.29 |

Table 4.2: Selected data showing trends in the neighbourhood sizes and the number of errors in the pheromone levels.

| p | #particles | | avg. nbd sep. | | var. nbd sep. | | #errors | |
|---|---|---|---|---|---|---|---|---|
| | mean | sd | mean | sd | mean | sd | mean | sd |
| .1 | 1024. | 32.23 | 4.47 | .01743 | 2.43 | .03287 | 67.87 | 8.879 |
| .2 | 1006. | 27.19 | 4.496 | .01708 | 2.349 | .02705 | 64.66 | 7.973 |
| .3 | 1009. | 27.52 | 4.522 | .01409 | 2.284 | .02258 | 63.41 | 7.018 |
| .4 | 1037. | 21.08 | 4.545 | .01264 | 2.23 | .02419 | 60.91 | 7.357 |
| .5 | 1015. | 23.27 | 4.566 | .01111 | 2.181 | .02269 | 59.39 | 6.197 |
| .6 | 1006. | 20.1 | 4.589 | .01101 | 2.138 | .02367 | 55.7 | 5.066 |
| .7 | 1011. | 17.5 | 4.611 | .01231 | 2.099 | .02038 | 54.81 | 3.837 |
| .8 | 1035. | 11.91 | 4.628 | .008717 | 2.064 | .01888 | 52.19 | 4.362 |
| .9 | 1040. | 11.28 | 4.646 | .009245 | 2.027 | .02169 | 49.8 | 3.885 |
| .1 | 5019. | 76.07 | 2.006 | .003665 | .483 | .002769 | 67.08 | 5.174 |
| .2 | 5057. | 63.54 | 2.018 | .003157 | .4681 | .002603 | 65.55 | 4.183 |
| .3 | 5076. | 62.39 | 2.029 | .00282 | .4554 | .002253 | 62.43 | 3.57 |
| .4 | 5024. | 58.66 | 2.039 | .002721 | .4446 | .002098 | 60.6 | 3.609 |
| .5 | 5103. | 50.22 | 2.049 | .0023 | .4348 | .001606 | 58.58 | 3.295 |
| .6 | 5078. | 52.29 | 2.058 | .002128 | .4258 | .001446 | 56.16 | 3.54 |
| .7 | 5048. | 37.24 | 2.067 | .002152 | .4182 | .001465 | 53.93 | 2.479 |
| .8 | 5119. | 27.7 | 2.076 | .002328 | .4106 | .001806 | 51.53 | 2.953 |
| .9 | 5060. | 13.85 | 2.083 | .001967 | .4051 | .001793 | 49.37 | 2.053 |
| .1 | 8008. | 89.19 | 1.587 | .002102 | .3018 | .001414 | 67.37 | 4.347 |
| .2 | 8079. | 74.54 | 1.597 | .002159 | .2924 | .001272 | 65.09 | 4.054 |
| .3 | 8064. | 71.61 | 1.605 | .001861 | .2844 | .001168 | 62.83 | 3.791 |
| .4 | 8070. | 66.48 | 1.613 | .001764 | .2774 | .0009679 | 60.27 | 3.27 |
| .5 | 8078. | 58.44 | 1.621 | .001325 | .2714 | .001035 | 58.85 | 3.3 |
| .6 | 8062. | 60.66 | 1.628 | .001417 | .2659 | .0009342 | 56.55 | 2.821 |
| .7 | 8018. | 52.11 | 1.635 | .001147 | .2612 | .000772 | 53.94 | 2.635 |
| .8 | 8159. | 36.71 | 1.642 | .001579 | .2567 | .0009116 | 51.77 | 2.569 |
| .9 | 8121. | 29.06 | 1.649 | .001175 | .2532 | .001053 | 48.8 | 1.29 |

Table 4.3: Selected data showing trends in secretion errors and in the Euclidean distances between neighbours.

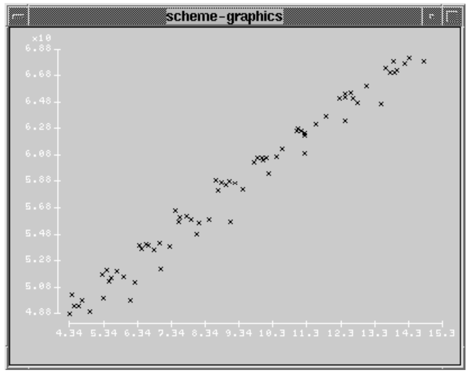


Figure 4-12: Scatter plot showing variation of secretion errors with average variance of the neighbourhood size.

| | #errors | nbd mean | nbd var | sep mean | sep var |
|---|---|---|---|---|---|
| nbd mean | .833 | – | .752 | -.559 | – |
| nbd var | .988 | | – | – | .221 |
| sep mean | -.0104 | | | – | .982 |
| sep var | .0927 | | | | – |

Table 4.4: Correlations between selected measurements.

during the experiment. The number of errors, as defined in the hypothesis, were counted for each secretion. In other words, for each value of $p$, there was one distribution. For each distribution, 20 secretion error tallies were taken whose mean was then computed and reported.

From the tables it is clear that within each expected size of the distribution, there is a trend where the number of errors decreases with the value of $p$. It appears, as well, that all the other parameters (except the average total number of particles) vary gradually with the value of $p$. However when we look for trends like these across the different sized distributions, most of them disappear. In Table 4.2 we see that the number of secretion errors varies with the variance in the neighbourhood size even across different sized distributions. Figure 4-12 shows on a scatter plot just how well these two measurements vary together. In fact, the correlation coefficient of the variance of the neighbourhood size and the average frequency of secretion errors is 0.988.

In Table 4.3, we again see the same trends of values within a given expected size of the distribution. But in this case there is not a strong correlation between the variance of separation between neighbours and the error frequency. In fact, the correlation coefficient of the variance of the Euclidean distances within a neighbourhood and the average frequency of secretion errors is only 0.0927.

Table 4.4 lists the correlation coefficients for different pairs of data that we thought could have been related to each other. It shows that there is also a reasonably high correlation between the average neighbourhood size and the number of errors. We suspect, though, that is an artifact of the dependence of the variance of the neighbourhood size on its mean, rather than any intrinsic connection.

Not being satisfied with the outcome of the correlation of the separation variance with the error frequency, we took a closer look at Table 4.3. We observed that, in fact, the values of the separation variance were significantly different from one domain size to another. Realizing that the step-size was also being varied to maintain a (nearly) constant average neighbourhood size, we decided to normalize the separation variance by the step-size. Actually, to make the units compatible, we took the square root of the variance and then divided by the step-size. We then recomputed the correlation coefficient, and found it to be 0.990.

**Discussion and Conclusions**

The data convincingly shows that the neighbourhood variance is indeed a very likely factor in the frequency of errors in a secretion. The *normalised* variance of the neighbour separation has a similarly strong correlation, but the straightforward variance does not. Although it seems plausible that these two measurements of variance are related, it is not immediately obvious how. The variance of the neighbourhood separation is calculated from the physical distance between a particle and each of its neighbours. On the other hand the variance of the neighbourhood size is computed strictly from the number of neighbours each particle has, and is blind to their distances from it.

### 4.4.5 Summary

Even if we do not fully understand the reasons, we are now fairly certain that the variance of the neighbourhood size of a distribution is extremely important for evaluating the effect of its geometry on our programs. We have seen this analytically, from the discussion on how the ability to propagate a growing point is affected, and now we have seen it empirically from the data produced for this experiment. We also saw that the mean neighbourhood size is oblivious to the geometry of the distribution, but the variance is not. Therefore it is completely plausible that our investigations into the effects of irregular geometry on our programs should identify the variance as an important measurement.

At this point we still do not have a very clear understanding of how a distribution ought to be characterised in order to yield the most information about its effect on our programs. We have identified the variance as an important measurement, but it is only one, and we do not know how many other properties are important. Indeed, we do not even know whether the variance is the most important property to consider when analysing the effects of irregular geometries. More experiments and analysis will be required to get a fuller understanding of the effects of irregularities in geometry on GPL programs, and Amorphous Computing in general.

## 4.5 Bibliographic Notes

The particles of an amorphous computer, and therefore a GPL domain can be modeled as a geometric graph (or more specifically a Euclidean graph). In a geometric graph the given vertices are assigned coordinates in $\mathbf{R}^2$ and straight lines are used to connect the end points of the given edges. See [39, 27, 51] for definitions and some results on geometric graphs. Results on geometric graphs that bound the number of edges in a graph with few of them crossing [44, 40, 30] indicate that if we can probably draw such graphs with little additional computational penalty.

Another model for an amorphous computer is as a hypergraph. A hypergraph is simply a generalization of the usual definition for a graph, but the edges, now called hyperedges, are sets of many vertices instead of only two. Since the definition is so general, we expect that many results on hypergraphs will be applicable to amorphous computing, although we had no opportunity to use any of the results on hypergraphs that we found [29].

In writing the definitions for *draw* and *resembles* we relied heavily on work from topology, graph theory and computational geometry. The text by White [53] provided many of the preliminary definitions that helped to shape the definition for resemble. The concepts of induced subgraphs and connected components are well established and can be found in almost any introductory text on graph theory, see [15] for example. From computational geometry, the concept of alpha-shapes [37] related to the concept of *draw* as defined. Alpha shapes are based on power diagrams [5] which are a generalization of Voronoi diagrams.

# Chapter 5

# Explorations in the Expressiveness of GPL

In this chapter we present a potpourri of programming techniques for GPL and outline practical approaches for describing some interesting patterns.

## 5.1 Generating Line Segments, Rays and Arcs

Line segments represent one of the simplest forms that GPL can generate, yet they are often at the heart of the construction of any pattern. The principle is quite simple, we assume we are given the two end points for the line segment, and we invoke two different growing points at those end points. At one end, a pheromone is secreted far enough to be detected at the other end. That other end initiates a growing point that has positive orthotropism to the pheromone emitted from the first end.

```
(define-gp (AB-segment)
  (material A-material)
  (size 0)
  (tropism (ortho+ B-pheromone))
  (actions
   (when ((sensing? B-material)
          (terminate))
         (default
           (propagate)))))
```

Line segments need not be completely drawn. For some patterns it may be desirable to make them grow towards some pheromone source, but terminate early under some appropriate conditions. That was the case with the line segments used in the previous chapter in the generation of planar graphs. In this case, we want to keep the definition simple, so we simply terminate when the growing point has reached the pheromone source (as expressed by the `(sensing? B-material)` clause).

```
(define-gp (B-point)
  (material B-material)
  (size 0)
  (actions
   (secrete+ radius B-pheromone)
   (terminate)))
```

Even though we can generate the pheromone secretion in the `with-initial-locs` statement, we still need a growing point to be invoked at the target end of the line, because we need to mark that end with the material that terminates `AB-segment`. As seen above, the necessary properties of that growing point definition are quite few. Most of the body of the definition is taken up by overhead syntax.

Finally we come to the point of invocation for the line segment. The `with-initial-locs` statement defines the number of types of initial conditions.

```
(with-initial-locs
 (a b)
 (at a (start-gp AB-segment))
 (at b (start-gp B-point)))
```

In this case, there are only two types, namely the two end points of the line segment. When this program is invoked, we may give it multiple points for each type. Each of those points in the domain will be initialised with the instructions described in the body of the `with-initial-locs` statement. For example, if we invoked the GPL program with the lists of points: $[(1,0),(0,1)]$ and $[(3,3)]$, then both points in the first list would be considered as type `a` and the point from the second list would be considered type `b`. That would lead to the growing point `A-to-B-segment` being invoked at both the points $(1,0)$ and $(0,1)$ and the growing point `B-point` being invoked at $(3,3)$. Assuming the value of `radius` multiplied by the step-size was longer than the distance between the points in the first list and the second point, then the resulting pattern would be two line segments.

### 5.1.1 Generating Rays: Modeling Inertia

The second method of generating a line is as a ray. Namely, if we are given a point and a direction, then we have a representation of a semi-infinite line. The direction can be modeled in GPL by two locations.

The idea behind rays is a very powerful one, and can serve as a GPL model of inertia. Given two nearby points, if we superpose two identical pheromone secretions, one centred at each of these points, we see that within the step-size of each point, the locally minimum values of the pheromone are located away from the other source. See the discussion on rays in Chapter 4 § 4.1.5 for the reasons for this.

So if we implement a growing point that secretes a short-range pheromone, for which it has a negative orthotropism, then it will be repelled by its own secretions and propel itself along a path that moves the active site away from the initial location. The actual implementation follows.

```
(define-growing-point (ray life)
  (material ray-mat)
  (tropism (ortho- self-pheromone))
  (size 0)
  (actions
   (secrete+ 3 self-pheromone)
   (when ((< life 1)
          (terminate))
         (default
           (propagate (- life 1))))))
```

To add a little more control to the direction initially taken, we can add a second initial location from which we secrete `self-pheromone` for exactly the same extent as the growing point does (three in this case).

Rays are a very useful tool for building more complex patterns. They will reoccur several times in the rest of the discussion presented in this chapter. However, there is a drawback to the dependence on rays as line generators: they are highly susceptible to the underlying Geometry of the domain. So on a regular grid, a ray's trajectory is usually a remarkably straight path, however it tends to meander wildly on a random distribution. Various measures can be taken to correct for some of that meandering, and it will depend on the circumstances for which corrective measure is appropriate.

### An Alternate Implementation

At first glance, it appears that rays cannot be implemented without the use of superposing pheromones. However, it turns out that it is possible to implement rays using only the `secrete` command. This fact is important since the implementation of `secrete+` requires more resources than that of `secrete`. In particular, the computation of the maximum of a sequence of values is independent of the multiplicity of each value, but not so for the sum. So considerable care has to be taken in the implementation of `secrete+` to distinguish between redundant messages for one secretion process and messages for different secretions. But for `secrete`, redundant messages do not affect the final computed value.

The idea of secreting a self-repelling pheromone is still the essence of the implementation, but in this case, there are two such pheromones. Two growing points are defined in such a way that each one secretes only one of the pheromones, but is repelled by both pheromones.

```
(define-gp (ray-A life)
  (material ray-mat)
  (tropism (and (ortho- self-pheromone-B) (ortho- self-pheromone-A)))
  (size 0)
  (actions
   (when ((< life 1)
          (terminate))
         (default
           (secrete 3 self-pheromone-A)
           (connect
            (propagate 0)
            (start-gp ray-B (- life 1)))))))

(define-gp (ray-B life)
  (material ray-mat)
  (tropism (and (ortho- self-pheromone-A) (ortho- self-pheromone-B)))
  (size 0)
  (actions
   (when ((< life 1)
          (terminate))
         (default
           (secrete 3 self-pheromone-B)
           (connect
            (propagate 0)
            (start-gp ray-A (- life 1)))))))
```

Growing point `ray-A` secretes `pheromone-A` and propagates for one step (the effect of `(propagate 0)`). After completing that single step, `ray-B` is invoked at that location so that `pheromone-B` can be secreted at the location. The result is an alternating pattern of secretions of `pheromone-A` and `pheromone-B` that cause the location that is simultaneously furthest from the previous two locations to be selected as the successive location in the trajectory of the ray.

The initial conditions simply setup the alternating pattern of pheromones by secreting one of the pheromones at the first location, and invoking the other growing point at the second location.

```
(with-initial-locs
 (base-pt source)
 (at base-pt
     (secrete 3 self-pheromone-A))
 (at source
     (start-gp ray-B 20)))
```

Observe that each growing point gives preference to the pheromone secreted by its counterpart. This encourages successor locations to be far from the active site and likely to be colinear with the active site and its previous location. In that respect, this alternative version of rays is less dependent upon the shape of pheromone distribution than the first version is. However, in practice, the rays produced by either method on irregular particle distributions tend not to be straight lines. That is because there are no feedback mechanisms to monitor and correct the overall deviation from a straight-line path. (It is not clear how such a correction mechanism would be implemented in GPL).

### 5.1.2 Arcs

Generating arcs is very similar to generating line segments. We need at least two points: one for the centre and the other for the start of the arc. The length of the arc may be specified as a parameter passed to the growing point responsible for drawing the arc, or it may be specified by supplying a third point to mark the end point of the arc. In the latter case, it may not be possible to control which of the two possible arcs gets generated unless yet another point is added, or some other extra measure is taken to differentiate between the two.

Like the line segment, there are two growing points involved. They also play similar roles: one generates a guiding pheromone which constrains the motion of the other one. The key difference between the growing points for the arc and those for the line segment lies in the tropism of the mobile growing point. The mobile growing point for the arc has diatropism to the guiding pheromone instead of positive orthotropism as was the case for the line segment.

```
(define-growing-point (centre)
  (material C-material)
  (size 0)
  (for-each-step
   (secrete+ radius C-pheromone)
   (terminate)))
```

```
(define-growing-point (arc@C length)
  (material arc-material)
  (size 0)
  (tropism (and (ortho- arc-pheromone) (dia C-pheromone)))
  (for-each-step
   (secrete+ 3 arc-pheromone)
   (when ((< length 1) (terminate))
         (default (propagate (- length 1))))))
```

The version of `arc@C` shown above has an additional tropism specified to encourage growth in a single direction. Since at any point on the circumference of a circle, all neighbouring points on the circumference will have the same pheromone level, then they will all qualify for diatropic propagation. Including a self-repelling pheromone encourages the motion to continue in the direction taken at the first step.

## 5.2 Approximating Euclidean Constructions

Euclidean Constructions are perhaps better known as "compass and straight edge" constructions. Although they are often taught at the high school level, one must understand Finite Field Theory[1] before one can fully appreciate the extent of possibilities afforded by Euclidean Constructions.

In this section we show that the operations involved in any Euclidean Construction can be expressed in GPL. Owing to the finiteness of any realizable GPL domain, we never actually obtain the perfect results of a construction, we must instead settle for an approximation.

The method for expressing Euclidean Constructions in GPL is inspired by the Finite Field theoretic view of them. Euclidean Constructions can be viewed as operations on the plane, $\mathbf{R}^2$; their results are all the points that arise from intersecting lines or arcs. As initial conditions, the points $(0, 0)$ and $(1, 0)$ are considered constructible points. A number is constructible if it is the distance between two constructible points.

The primitive operations of Euclidean constructions are:

1. the ability to draw a line through two constructible points.
2. the ability to draw an arc of constructible radius about a constructible point.
3. the ability to draw an arc of crudely specified radius (say larger than some constructible number) about a constructible point.

Finite Field Theory contributes some remarkable facts about Euclidean Constructions by manipulating the set of constructible numbers that can possibly arise from these primitive operations (a full explanation can be found in many advanced Algebra texts, see Artin's [4] for example). At this point however, we deviate from the Finite Field Theory view of Euclidean Constructions, and focus on translating these primitive operations into GPL, while allowing them to be composed in the manner of a typical construction.

[1] Finite Field Theory is sometimes referred to as 'Galois Field Theory' since a finite field is called a Galois field. We insist upon the term 'Finite Field Theory' partly because 'Galois Field Theory' is typically associated with the (arguably) more important issue of the solvability of polynomials with rational coefficients in terms of surds of those coefficients. The other reason is in recognition of the fact that Gauss published his results on Euclidean constructible polygons (which were subsequently incorporated into Field Theory) at about the time Galois was born. See Boyer and Merzbach's book [9] for more details.

### 5.2.1 Translating the primitives

We have already shown in the previous two sections that line segments, rays and arcs can be expressed in GPL. Those subroutines form the basis of our argument that Euclidean Constructions are GPL expressible. We first observe that the concept of "intersection" is GPL expressible: simply associate some material with the growing point representing the object to be intersected, then test for its presence with the growing point for the intersecting object using the `sensing?` predicate. This establishes the plausibility of the whole idea.

Expressing the operation of drawing a line through two constructed points is accomplished by composing a slightly modified version of `AB-segment` and `ray`. The `AB-segment` growing point will only draw a line segment between the two specified end points. Since Euclidean Constructions sometimes call for lines to be *produced* (i.e. extended beyond their end points), we also need the `ray` growing point. We have to modify the growing point definition for `AB-segment` to secrete the pheromone to which `ray` is negatively orthotropic.

```
(define-growing-point (AB-segment)
  (material A-material)
  (size 0)
  (tropism (ortho+ B-pheromone))
  (for-each-step
   (when ((sensing? B-material)
          (terminate))
         (default
           (secrete+ 3 ray-pheromone)
           (propagate)))))
```

In this way, invoking `ray` at the termination point of `segment` expresses a ray that points in the direction parallel to the line and away from the termination point of `AB-segment`. The definitions of `B-point` remains unchanged. Expressing rays and arcs is done similarly as described in the previous section. In addition to counting steps for our termination condition, we may sometimes use the `sensing?` predicate to test for the intersection of arcs and lines. The following is an example growing point definition that expresses an arc about some source secreting `B-pheromone`.

```
(define-growing-point (arc)
  (material arc-material)
  (size 0)
  (tropism (and (dia B-pheromone) (ortho- arc-pheromone)))
  (for-each-step
   (secrete+ 3 arc-pheromone)
   (when ((sensing? some-material) (terminate))
         (default (propagate)))))
```

The growing point terminates when it encounters a growing point that deposits *some-material.* This behaviour is analogous to finding the point of intersection of the arc that `arc` represents and the curve that the material *some-material* indicates. Whether the point of termination is used subsequently is dependent on the particular construction, but by terminating there, we make it available (through either the `connect` form or a network form) to an expression that invokes `arc`.

Strictly speaking, Euclidean constructions do not permit specifications of distances that have not been constructed. However we retain the length-terminated growing points as a convenience to facilitate specifying bounded length arcs and line segments when we can be certain that they have been extended far enough to intersect as intended. (We typically do this when manually performing a construction. To bisect a line for example, we often draw only the portion of the arc that extends above the point we estimate to be the centre of the given segment.)

### 5.2.2 Combining the Primitives: An Example

The code fragments for generating arcs, line-segments and rays may be combined with the aid of the `connect` form. Consider, for example, the construction of a sixty degree angle. Like any other Euclidean construction, we start with two initial conditions. In this case, we choose to make them represent a line segment of a length that we shall specify as a constant, called `radius`, in the program. (We can think of that constant as being the length of our given straight edge, for example.) One of the initial locations, specifically the second, will simply invoke the stationary growing point of the pair for generating a line segment. Here is the entry point for our construction:

```
(with-locations
 (a b)
 (at a
     (connect (start-gp base-line)
              (start-gp dest)
              (start-gp dest-arc))
     (start-gp src)
     (start-gp src-arc))
 (at b (start-gp base-line-stop)))
```

From the first point, we draw our base line with length `radius` and establish at the end of it the point from which we will swing an arc, called `dest-arc`, about our initial start point `a`. Simultaneous with drawing our base line, we establish at `a` the centre about which `dest-arc` is swung, and invoke an arc called `src-arc` to be draw about the end point of the base line. The growing point for `src-arc` does not actually begin propagating until the line segment has terminated and the `dest` growing point has secreted the necessary pheromone for `src-arc` to follow.

Figure 5-1 demonstrates the result of this GPL program. Notice that the arcs had to be given non-zero sizes so that they would be guaranteed to intersect. This approximation is a direct consequence of the size of step-size, the fundamental distance in the GPL domain. The smaller it is, then the smaller the region for intersection, so the more localised is the point of intersection. On the other hand, in order to represent multiple directions for rays, we must preserve a large number of particles within the neighbourhood of the invoking particle. Smaller fundamental distances and large neighbourhoods together imply higher densities. Hence, in order to obtain reasonable approximations to Euclidean constructions, the domain must be dense. As we have seen from the previous sections, we also want the densities to have low variance so that growing points like `AB-segment` and `ray` approximate straight lines better. The complete GPL code for the construction presented is given in Appendix A.

We have described how the primitive operations for Euclidean constructions may be expressed in GPL, and shown an example of how they may be combined. In general, to initiate the appropriate growing points at the appropriate locations, we make use of the `connect` form and possibly any networks we may define. Since the overall construction pro-

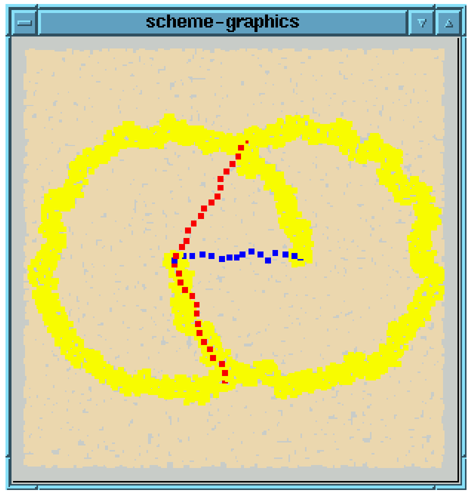


Figure 5-1: A GPL construction of 60°.

cess is deterministic, we know ahead of time which constructed points will be responsible for what actions. That knowledge allows us to encode the appropriate sequence of invocations of growing points by ensuring that growing points terminate (or invoke another growing point) at a point of intersection. At times, we may have to choose one from two or more newly constructed points to initiate a growing point. At those times, we resort to symmetry breaking, by either specifying extra initial conditions or by arranging for previously constructed objects to secrete appropriate pheromones to indicate their relative positions to the vying locations. An example of the former type of symmetry breaking is discussed in Section5.7, where an extra initial condition is used to distinguish one side of a line from the other.

## 5.3 Drawing Circuit Layouts

We have already presented by way of example, how CMOS circuit layouts can be expressed in GPL. So we will limit our discussion here to the general techniques that can be applied to the specific tasks required by circuits. We begin the discussion by describing a library of circuit parts. During the discussion to follow, we will present a few relevant excerpts from it. The complete code listing of this library is given in Appendix A.

### 5.3.1 A Library of Building Blocks

The network abstraction and the connect form allow us to build up complicated patterns from simpler ones. Consequently, we can build a library of routines for drawing common patterns found in circuits, and thereby simplify the description of complicated circuits. We will now give a few examples of the modularity afforded by the library, and present some of the abstractions that we considered useful.

### Establishing the Reference Points

Since line segments produce more reliable patterns than rays do, we always want to minimize the use of rays to produce straight line approximations. Therefore, in general, for any pattern in which the geometry of the lines is important, we are usually interested in finding the minimum number of lines that must be expressed as rays (i.e. whose end points are not already established as either termination points of other growing points or initial locations). For the case of the CMOS circuit layouts we have considered, we chose to implement the rails with rays and specify everything else in terms of the reference pheromones generated by them[2].

As described in Chapter 2, rails were implemented as a pair of alternating rays and line-segments. Each rail would grow for a short distance and generate a long range pheromone to constrain the other rail which would be growing in its line segment phase. Then they would exchange roles, and the former rail would grow as a line segment constrained by the other. The important feature that we wanted to produce was a pair of parallel lines. We could, in principle, choose any implementation of parallel lines, and simply substitute their definition into the library. Indeed, we could even name the implementations differently and provide them all, allowing the programmer to choose the most desirable implementation as appropriate to the intended domain.

### Capturing Common (Programming) Patterns

In our circuit library implementation, everything else that is not a rail is specified in terms of the pheromones that the rails are expected to secrete, namely `vdd-long` and `vss-long`. Using those pheromones, we specify horizontal and vertical lines of the various types of materials relevant to CMOS circuits: metal, diffusion and polysilicon. One common pattern is a horizontal line of a specified length. We use diatropism to one or both of the rail pheromones to implement that, but since there are usually two options when propagating, we usually need to modify the implementation a little to create a line that proceeds in only one direction.

That modification is essentially to combine the idea underlying how rays are generated with the diatropism we originally wanted. To get the first step in the correct direction, we also need the help of another pheromone that marks where the rails started, so that left-to-right is defined as the direction of growth of the rails. To be concrete, here is the complete definition of the growing points used to produce a horizontal strip of polysilicon between the rails.

[2]That choice arose from an early implementation of the inverter circuit, and remained so throughout the rest of the circuit library development. With the benefit of hindsight, since rails are usually long, and need to be straight, that was probably not the best choice. Several alternatives exist, but not many have been explored, for improving the robustness of the implemented circuits to the GPL domain Geometry.

```
(define-gp (horiz-poly length)
  (material poly)
  (size N-WIDTH)
  (avoids poly-id-B)
  (actions
   (secrete 2 poly-id-A)
   (connect (start-gp poly-left->right 2)
            (start-gp poly-with-inertia (- length 2)))))

(define-gp (poly-left->right length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- dir-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (actions
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))

(define-gp (poly-with-inertia length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- poly-pheromone)
                (and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (actions
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

Although the implementation is relatively simple, it takes three growing points to accomplish. By providing definitions like these in the circuits library, the programmer may invoke `horiz-poly` without having to implement all the supporting code himself, so long that he ensures that `dir-pheromone` is maintained as the circuit progresses between the rails.

### Extending the Library

Of course, we also provide transistor abstractions. In our library, we have included code to draw both types of transistors vertically.

```
(define-network (vert-p-fet (gate) (src drain))
  (at gate
      (start-gp horiz-poly 2)
      (connect (start-gp up-p-diff FET-HEIGHT)
               (->output src))
      (connect (start-gp down-p-diff FET-HEIGHT)
               (->output drain))))
```

Many CMOS layouts often call for transistors to be horizontal instead. So one useful set of additions to our library would be to include the appropriate combinations of growing points to draw a transistor horizontally. This could, in principle, be easily accomplished by using the techniques for drawing polysilicon horizontally to draw diffusion horizontally (in both directions) and combining them with a vertical segment of polysilicon in a similar manner to `vert-p-fet`

### 5.3.2 Preventing Unwanted Proximity

As mentioned in Chapter 2 it is sometimes necessary to not only discourage, but to prevent at all costs one growing point from getting too close to another. In such circumstances, we use the avoids clause to indicate which pheromones will poison the growing point.

The avoids clause adds a measure of robustness to our programs by guaranteeing that certain growing points maintain a minimum distance between themselves. However, there is a cost associated with that added robustness: increased code size. Although, the problem is readily fixed, under the current implementation, all pheromone names are specified as symbolic constants. Therefore, in a general appication like the circuits library, we will need to define many copies of a single type of circuit element. For example, to incorporate the robustness afforded by the avoids clause in the implementation of `poly-horiz` given above, we must define alternative versions of both `poly-with-inertia` and `poly-left->right` that differ from the given versions only in that they avoid the pheromone `poly-id-A` and secrete `poly-id-B`. As a result, a complete library would contain large numbers of growing point definitions to represent all the necessary flavours of circuit parts between the rails. Of course, in a given application, a clever compiler could simply ignore the growing points from an included library that were not used, thereby reducing the actual code size loaded into a processing particle.

The more general solution is to allow variables to be bound to pheromone names. The most significant impact of such a change would be that the syntax of GPL would have to be modified slightly to allow a quoting mechanism to distinguish between a pheromone being directly named and one being denoted by a variable. The implementation issues are, in fact, surprisingly trivial. They were not incorporated into the language prototype described in this document because the drawback of using the avoids clause was not realized until very late.

### 5.3.3 Alternative Implementations

With the benefit of hindsight, we realised that a drawback to defining all the circuit parts relative to the rails was that the rails sometimes need to be placed very far apart which require secretions over very long distances. Those secretions take a long time to settle, and a long time to simulate. Furthermore, values at the tail regions of a secretion are all very low and therefore provide a smaller range for resolving directions among a particle's neighbours. There are a number of alternatives that could have been used, and perhaps ought to be tried in future. We present a few ideas here.

One alternative to generating rails is to have a ray that secretes a long range pheromone of extent half the distance between the rails. The rails would be generated by points (perhaps established in the initial conditions) that grew diatropically to that pheromone. Since the secretion distribution is well-behaved under conditions of low variance in density, the two rails are likely to remain apart at their initial distance. The issue with this alternative is to ensure that the growing points grew in the same direction as the central guiding ray did.

Another alternative is to define some of the circuit components in terms of rays and the rails in terms of line segments. In this implementation, rails would have to be generated on demand, and circuit elements would have to rely on shorter range pheromones to proceed in the appropriate directions. For example, in the case of the inverter, we could generate the polysilicon and diffusion pieces with rays, and then when we grew a transistor, we would invoke a line segment to connect the previous rail terminal with the appropriate recently established transistor terminal. This implementation would have the advantage that the circuit elements would depend on shorter range pheromones, but the disadvantages would be that long stretches of polysilicon, metal or diffusion would vary more than the current implementation. Another drawback would be that the pheromone extent of rail growing point attractors might still have to be long if the estimated distance between consecutive rail-contacting circuit elements was large.

The alternate strategies for generating circuits presented here are just a few ideas and are probably worth implementing independently to measure their relative successes to the current strategy. Even better would be if those alternatives were offered along with the current implementation to provide a GPL programmer with plenty flexibility in implementing a robust circuit.

## 5.4 Looping Constructs

Mobile growing points are, in some sense, inherently recursive: invocation at a particular location can lead to an identical evaluation of expressions, albeit at a different location. So far, we have seen this property in all incarnations of the `ray` and `AB-segment` growing points, as well as in other growing points. However, we have not yet seen how GPL might be used to express the concept of repeating a computation at a single site for a number of times not determined until run-time. How to accomplish looping at a site in GPL is the focus of this section. Although not explicitly presented as a language construct, such a facility is available in GPL, and we shall now demonstrate how it can be exploited.

### 5.4.1 Repeating execution at a single location

First we demonstrate the basic looping construct. In the example below, the growing point `loop` takes two arguments, `count` and `length`. The `length` parameter is used only by the particular action that this loop happens to repeat, namely the invocation of the `ray` growing point.

```
(define-gp (loop count length)
  (actions
   (when ((= count 0) (terminate))
         (default
           (start-gp ray length)  ;; arbitrary actions go here
           (start-gp loop (- count 1) length)))))
```

The `count` parameter is the index for the loop, and in this case, the loop invokes the `ray` growing point as many times as the value of `count` for the initial call to `loop`. Notice that `loop` itself is not mobile, but that does not constrain the actions that it may loop over.

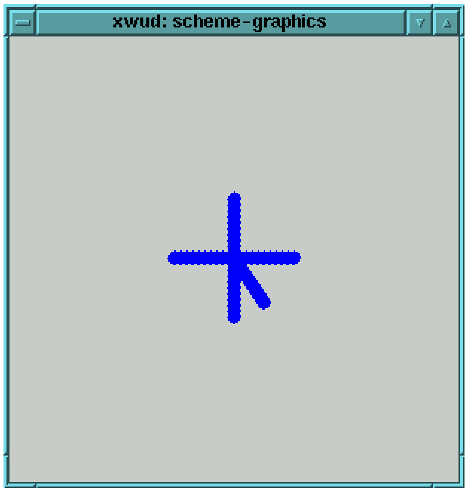


Figure 5-2: Invoking `ray` 5 times at the same point.

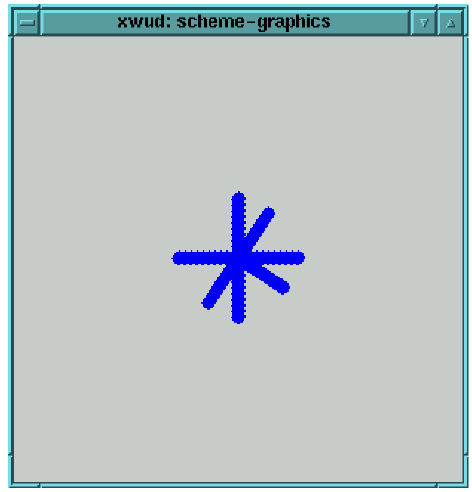


Figure 5-3: Invoking `ray` 7 times at the same point.

In this example, invoking `(start-gp loop` $n$ `10)` at a point, causes `(start-gp ray 10)` to be invoked at the same location $n$ times. Since rays repel each other (and themselves), the resulting pattern is a 1-level n-way tree, rooted at the initial point of invocation. Figures 5-2 and 5-3 illustrate the result of invoking `(start-gp loop 5 10)` and `(start-gp loop 7 10)`, respectively on a square grid. (The sizes of the growing points were set to 1 to make the patterns more prominent.) See the complete code listings in Appendix A for any further details of how these patterns were obtained.

If we wished to loop until a particular condition was met (akin to a *while* loop in C or Pascal, for example) we would simply replace the termination condition with the appropriate predicate.

One limitation to the kinds of looping constructs we may create is that growing points are not first class objects since they cannot be passed as arguments to other growing points. (That was intentionally implemented so that bandwidth limitations would not be accidentally violated by passing huge growing point representations across particles representing the path of a growing point.) Therefore, we cannot express a generalised looping construct that accepts a sequence of actions as an argument and then loops over the evaluation of that sequence.

## 5.5 Biological Inspirations

A number of Biological forms can be represented (at least topologically) in GPL. That is perhaps not a surprising fact, since on a gross scale, Biological forms tend to have rather simple topologies.

### 5.5.1 Starfish

We start simply by describing the topology of a starfish. The remarkable property of the form of a starfish (from our viewpoint anyway) is not that it has five arms, but that they are arranged symmetrically around it. From our viewpoint, arranging a topology of five branches meeting in a single point is as simple as invoking a variant of the `ray` growing

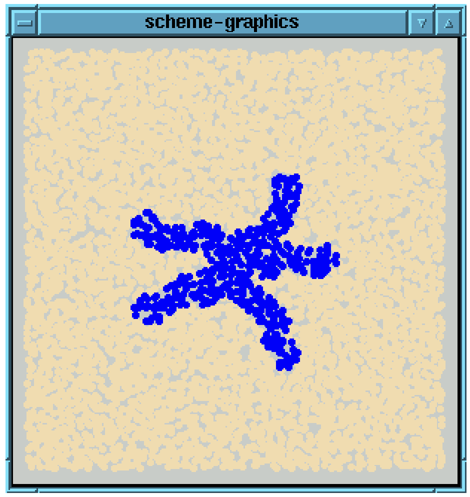


Figure 5-4: A starfish, as interpreted by GPL.

point five times at some given point. The GPL implementation follows, and Figure 5-4 shows the result of its evaluation on a random distribution of 5000 particles.

```
(define-gp (loop count length)
  (actions
   (when ((= count 0) (terminate))
         (default
           (start-gp arm length)
           (start-gp loop (- count 1) length)))))

(define-gp (arm length)
  (material arm-material)
  (size 1)
  (tropism (and (ortho- arm-p) (ortho- source-ph)))
  (actions
   (secrete+ (+ (- ARM-LEN length) 2) arm-p)
   (when ((= length 0) (terminate))
         (default (propagate (- length 1))))))

(with-initial-locs
 (root)
 (at root
     (secrete ARM-LEN source-ph)
     (start-gp loop 5 (- ARM-LEN 1))))
```

Since rays secrete self-repelling pheromones, they tend to fill the space available when many of them are invoked from a common point. That property is the basic principle behind the code above. We vary the extent of the secretions as the arms grow so that they continue to repel each other. Since we want the arms to always make progress away from the initial location, we add the `(ortho- arm-p)` clause to the tropism of the arms, and secrete `arm-p` at the initial location (as indicated in the `with-initial-locs` clause).

### 5.5.2 Trees

Generating trees is also based on the repulsion between identically defined rays. We again vary the vanilla ray by manipulating the extent of the secretions at appropriate points in the growth, so that the bifurcating branches are repelled from the root more than from each other. That added tropism causes the branches to continue to grow "upwards" (as defined by the two initial conditions), rather than perpendicular to the main branch. To illustrate the difference, we show in Figure 5-5 the result of a very simple tree implementation whose code is presented below. We use a width of zero for the trees illustrated here so that the smallest branches are not obscured.

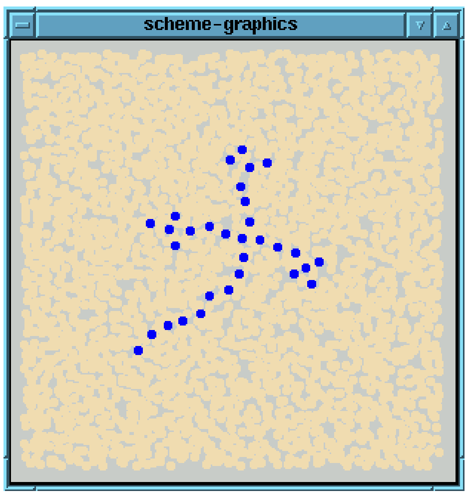


Figure 5-5: A tree with branch factor 3. Notice how the locations around each branch point are relatively evenly spaced, because each side branch repels the other as much as the trunk repels each of them.

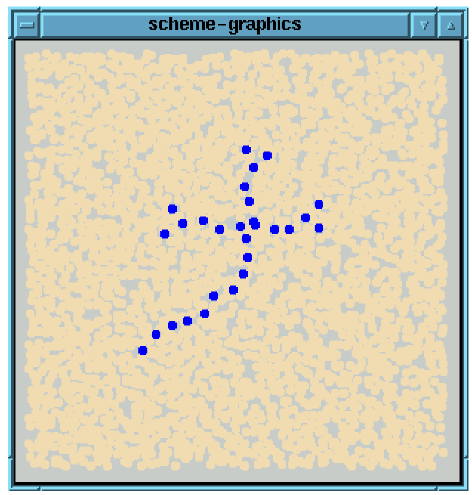


Figure 5-6: A tree with branch factor 3. Each new side branch is pushed away from the trunk more than from each other, as indicated by the bunching up of locations in the trajectory near the point of bifurcation.

```
(define-gp (tree init-life life)
  (material plant)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< init-life 3)
          (terminate))
         ((< life 1)
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3)))
         (default
           (propagate init-life (- life 1))))))

(color
 ((plant) "blue"))    ;; colour the tree blue
```

```
(with-locations
 (base-pt source)
 (at base-pt
     (secrete+ 3 self-pheromone))
 (at source
     (start-gp tree 9 9)))
```

In this case, we did not bother to encourage the branches to grow away from where the roots would be (the way a real tree might), and we can see in Figure 5-5 that the branches tend to form right angles with the "trunk" (i.e. they would be right angles if the tree were drawn on a regular grid). When we arrange for a little extra self-pheromone to be secreted just before the branch point, then we obtain branches that are closer to each other and further from the trunk. The modification to achieve this is simply an extra condition in the `when` clause of the definition of `tree`.

```
(define-growing-point (tree init-life life)
  (material plant)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< init-life 3)
          (terminate))
         ((= life 1)
          (propagate init-life 0)
          (secrete+ 4 self-pheromone))
         ((< life 1)
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3))
          (propagate (/ init-life 3) (/ init-life 3)))
         (default
           (propagate init-life (- life 1))))))
```

When `life` has a value of 1, then the branch (or trunk) is about to terminate and branch into three pieces. At this point we secrete some extra `self-pheromone` (extent 4) so that the last two locations selected in the branching process will be repelled more by the current trunk than by the first branch location. We still want the first branch (which acts like the extension of the trunk) to be in approximately the same place as it was before, so we secrete only after it has been established. In this way, the extra secretion affects only the two side branches, and not the centre one.

Figure 5-6 shows the result of the modified code evaluated with the same initial coniditions and domain as the original version. Notice that the locations starting branches off the main trunk are very bunched together. The branches do not grow as closely as that though, because they still repel each other. The extent to which the branches are pushed away from the trunk is controlled by extent of the secretion at the location before the trunk branches.

### 5.5.3 An Arm

For our final Biologically inspired form, we will show how the size attribute can be used to obtain limited geometric characteristics, by producing a cartoon hand. Figures 5-7, 5-8 show the pattern generated on both a regular grid (using the Euclidean metric) and an

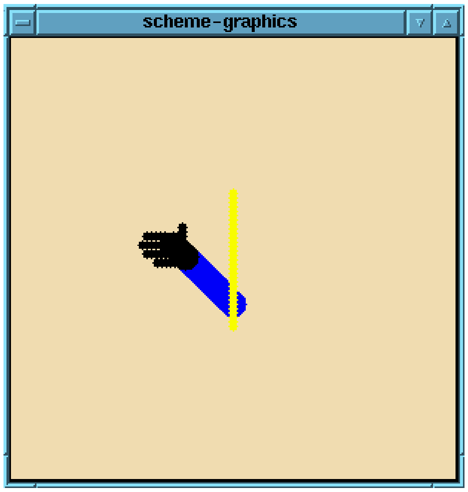


Figure 5-7: A hand, rendered by GPL on a regular grid with a Euclidean metric.

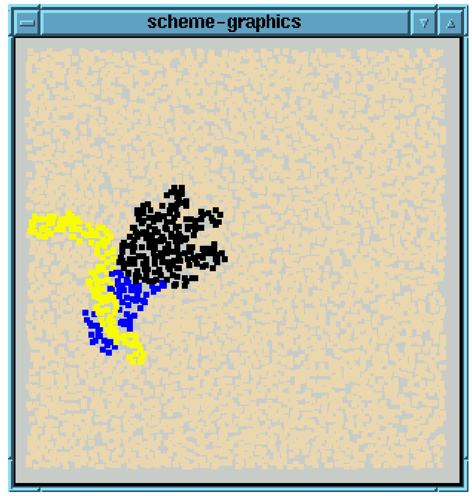


Figure 5-8: The same hand rendered on an irregular particle distribution with a shortest-path metric

irregular particle distribution. Below we present the more interesting fragments of the code responsible for the pattern.

The approach is to establish a reference line that secretes a long-range pheromone, called `mirror-p`. That long-range pheromone acts as a direction indicator for the arm that supports the hand, as well as to differentiate the growth of the fingers from the thumb.

```
(define-gp (arm length)
  (material arm-m)
  (tropism (and (ortho- dir-p)
                (ortho- arm-self-p)
                (plagio- mirror-p)))
  (size 3)
  (actions
   (secrete+ 4 arm-self-p)
   (when ((> length 0) (propagate (- length 1)))
         (default (terminate)))))
```

The arm is defined like a ray, but embellished with an negative plagiotropic growth from the reference pheromone so that it grows away from the reference line. The pheromone `dir-p` is secreted at the base of the reference line, so that the inclusion of the `ortho- dir-p` clause, eliminates one of the two possible options for the plagiotropic directions with respect to `mirror-p`. The arm is given size 3 so that it will be thicker than the fingers (which will be given size 1).

```
(define-gp (hand finger-length)
  (actions
   (secrete+ 4 finger-p)
   (connect (start-gp palm-starter 1)
            (start-gp palm PALM-SIZE finger-length)
            (start-gp fingers 3 finger-length)
            (start-gp thumb finger-length)
            (start-gp finger-rest finger-length))))
```

The hand is composed of the palm, the thumb and four fingers—three of which are produced in an identical manner. The `palm-starter` growing point is nothing more than a zero-width version of the arm. It simply extrapolates one point beyond the termination of the arm (the point of invocation of the `hand` growing point. That point is then used as the centre of the palm, and the source of all the digits.

```
(define-gp (palm sz finger-length)
  (material hand-m)
  (size sz)
  (actions
   (secrete (+ sz finger-length) hand-p)
   (terminate)))
```

The palm is nothing more than a very fat point, its size determined by `PALM-SIZE`, a globally declared constant. The palm also secretes a pheromone `hand-p` which is used by the digits to encourage their progress away from the centre of the palm.

The fingers are produced by defining code to produce a single finger and then taking advantage of the looping construct described earlier to reproduce it three times. The fourth finger is produced slightly differently so that it comes out shorter than the other three. It is also started last, so that the thumb and other fingers encourage it to occupy the space between the three fingers and the arm. Here are the definitions of `fingers` (a looping construct) and `finger`, the growing point that it loops over.

```
(define-gp (fingers count length)
  (actions
   (when ((= count 0) (terminate))
         (default
           (start-gp finger length)
           (start-gp fingers (- count 1) length)))))

(define-gp (finger length)
  (actions
   (connect (start-gp finger-start)
            (start-gp finger-rest length))))
```

Each finger is produced in two stages. The first stage, `finger-start`, tries to establish the proper direction of the finger and the second stage emphasizes the extension of the finger.

```
(define-gp (finger-start)
  (material hand-m)
  (size PALM-SIZE)
  (tropism (and  (ortho- arm-self-p)
                 (plagio- mirror-p) (ortho- finger-p)))
  (actions
   (secrete+ 2 finger-p)
   (when ((sensing? palm-m)
          (propagate))
         (default
           (terminate)))))
```

The first stage of finger production, `finger-start`, produces growth away from the base of the hand, and away from the reference line. It also tries to separate the fingers by secreting `finger-p` to which the fingers grow negatively orthotropically. The material deposited by `finger-start` is actually part of the palm, so its size is set to the PALM-SIZE constant. The point at which the first stage of finger growth transitions to the second is determined by where the palm material `palm-m` ends. The second stage of finger growth is represented by the `finger-rest` growing point.

```
(define-gp (finger-rest length)
  (material finger-m)
  (size 1)
  (tropism (and (ortho- finger-p) (ortho- hand-p)
                (or (ortho- mirror-p) (plagio- mirror-p))))
  (actions
   (secrete+ 2 finger-p)
   (when ((> length 0) (propagate (- length 1)))
         (default (terminate)))))
```

This stage of finger growth is the part responsible for producing the digits that we recognize in the pattern as fingers. Its primary tropism is to grow away from `finger-p` which it secretes as it progresses. So fundamentally, it behaves like a ray. The other tropisms are there to encourage the growth to occur along the general directions already established by the arm and the hand.

Finally, we make a growing point that combines the arm and the hand, called `arm+hand`.

```
(define-gp (arm+hand)
  (actions
   (connect (start-gp arm ARM-LEN)
            (start-gp hand FINGER-LEN))))
```

The `with-initial-locs` expression sets up a reference line, and invokes `arm+hand` once to grow the entire arm. We defer the presentation of the `with-initial-locs` expression until the next section, where we show what happens if we invoke `arm+hand` twice instead of once.

## 5.6 Expressing Mirror Symmetric Forms

Generating mirror symmetric forms is actually simpler than it may seem at a first glance. Recall that in a plane, given a line and a point on that line, there are two possible locations in the plane that are a fixed distance from that point. Those two points are mirror images of each other after reflection of the plane through the given line.

The approach we take is to first establish a mirror line (say by using `AB-segment`) which secretes a long range pheromone all along its trajectory. Observe that the long range pheromone is of lower concentration on either side of the line, so that if we specify that a growing point grows away from this line, both the left and right sides are acceptable destinations for propagation. In fact, we have no control over whether it takes the left or right path. The idea behind producing a form that is symmetric is to twice invoke a ray-like growing point (i.e. secretes a self-repelling pheromone) that is sensitive to the mirror-line's long-range pheromone as well. When the first ray chooses one side of the mirror line, it secretes its self-repelling pheromone which causes the second invocation to choose the other side of the mirror line.

Figures 5-9 and 5-10 illustrate the principle applied to the definition of the hand given in

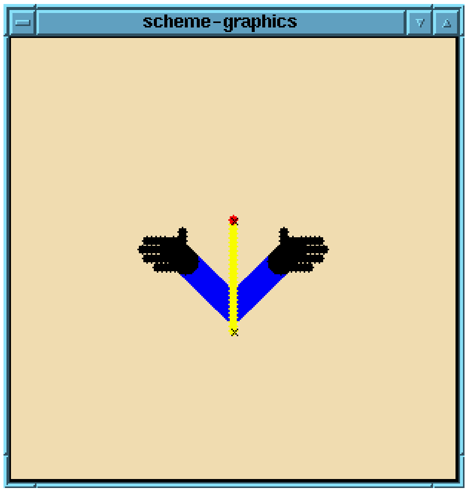


Figure 5-9: A pair of hands generated by mirroring the code for a single hand.

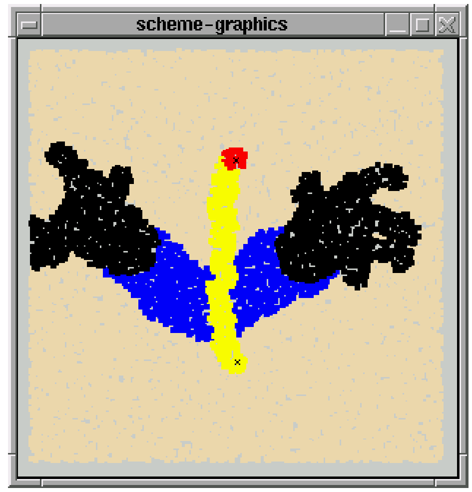


Figure 5-10: The same program rendered on an irregular particle distribution with the shortest-path metric

the preceding section. The only modification made to the code for the hand was to invoke the `arm+hand` growing point twice instead of once in the `with-initial-locs` clause.

```
(with-initial-locs
 (mirror-start mirror-end)
 (at mirror-start
     (secrete (+ MIRROR-LEN-1 MIRROR-LEN-2) dir-p)
     (connect (start-gp mirror-line-start MIRROR-LEN-1)
              (start-gp arm-starter-attractor (+ ARM-LAG-LEN 1))
              (connect (start-gp mirror-line-start ARM-LAG-LEN)
                       (connect (start-gp arm-starter ARM-LAG-LEN)
                                (start-gp arm+hand ARM-LEN FINGER-LEN)
                                (start-gp arm+hand ARM-LEN FINGER-LEN)
                               )
                       (start-gp mirror-line))))
 (at mirror-end
     (start-gp mirror-stop)))
```

The two commands responsible for producing the two arms are highlighted in boldface. All the other commands simply setup the mirror line, and make sure that the arm is not invoked until the long range pheromone secreted by the mirror has had a chance to spread. All the upper case variable names denote constants. The constant `ARM-LAG-LEN` denotes the length of the mirror that is drawn after the starting point of the arm has been established. The arm is not invoked until that segment of the mirror has completed. For a complete listing of this program, see Appendix A.

## 5.7 Simple Text

Figure 5-11 shows that simple text is also expressible in GPL. The yellow curve in the picture represents the line that supports the text. It secretes a long range pheromone which is the reference to which each letter is defined. Just as when a child learns to write, she

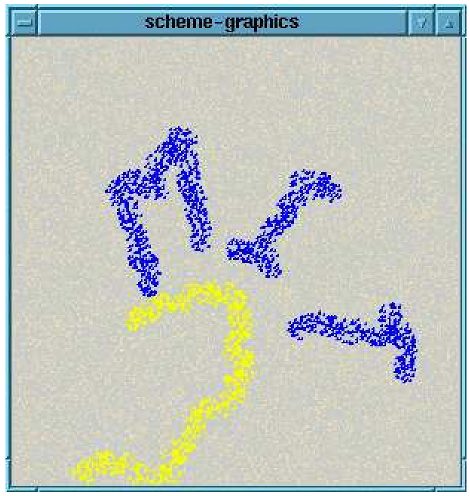


Figure 5-11: Text expressed in GPL.

is taught how each letter of the alphabet must be "drawn" relative to the line, we have to encode each letter similarly. We do that as a series of strokes that are either towards or away from the line, and either directly (perpendicularly) or obliquely. Letters with curves would be expressed as arcs, similar to the way they were described in § 5.1.2.

Since the letters are all built out of the strokes described, we define those strokes as primitive growing points that have tropisms with respect to `base-line-long`, the pheromone secreted by the base-line, and `dir-pheromone`, a pheromone that determines the left-to-right direction along the line. We then implement the letters themselves as networks, which can then be strung together with the cascade form. As each letter is drawn, the line is extended beneath it. So each network representing a letter has two inputs and two outputs. The inputs indicate the start location of the letter to be drawn, and the point from which the line should be extended. The two outputs represent the point where the letter terminates, and the tip of the extended line. For example, the letters "MIT" were generated by the following fragment of code, taken from the `with-initial-locs` clause of the program.

```
...
(==> (txt-start line-start)
     M spc I spc T spc
     (txt-ignore line-ignore))))
```

Here, the expressions `M`, `I`, `T` and `spc` represent networks that were defined to draw the corresponding letters and a blank.

### 5.7.1 The initial conditions

Since under normal circumstances, either side of the line is equally acceptable for the "up" direction, we supply an initial location to indicate where we want the "up" side of the line to be. In total, there are three initial locations, two for the base line and the third to indicate the up-direction. The third location is used as the starting point for drawing the text, so it is usually given close to the line, but far enough that the pheromone it secretes will have clearly differentiated values on either side of the line. Here is the first part of the `with-initial-locs` clause that produces the important setup pheromones.

```
(with-initial-locs
 (base-1 base-2 left)
 (at left (secrete LINE-HEIGHT up-pheromone))
 (at base-1
     (secrete+ 3 base-line-short)
     (secrete+ LINE-HEIGHT dir-pheromone))
```

The location `left` is the location that determines the side of the line that will be "up." The pheromone `up-pheromone` will be sought out eventually by a growing point that will be responsible for triggering the text to be generated. In order to establish the left-to-right orientation to be in parallel with the direction of propagation of the line, we cause `dir-pheromone` to be secreted from the first point on the base-line. Each letter also secretes enough `dir-pheromone` at key points to affect the following letter. In that way, the text continues to propagate in a consistent direction, long after it has left the realm of influence of the original `dir-pheromone` secretion from `base-1`.

The second part of the `with-initial-locs` statement generates two points as inputs to a network: the start point for text and the start point for the line.

```
(let-locs
 ((txt-start line-start)
  (at base-2
      (--> (start-gp base-line 5)
           (start-gp text-starter-attractor 5)
           (--> (start-gp base-line 5)
                (--> (start-gp text-starter)
                     (--> (start-gp decide-up 4)
                          (->output txt-start)))
                (->output line-start)))))
 (==> (txt-start line-start)
      M spc I spc T spc
      (txt-ignore line-ignore))))
```

The expression evaluated at `base-2`, `(start-gp base-line 5)`, simply extends the base-line by five units. At the end of that we invoke the `decide-up` growing point, but only after ensuring that we have extended the base-line another five units. The combination of `text-starter` and `text-starter-attracter` act as a serializer. The reason we want the base line to get a head start on the text generation is that the base-line secretes the pheromones upon which the letters depend. So we must ensure that we have the pheromones in place before the text is generated otherwise the letters may track misleading pheromone levels.

### 5.7.2 Defining the strokes

The strokes are defined using a pen metaphor. The pen may be lifted and moved, or it may be down and moved (to leave a mark). For each direction that we might need to move the pen, we must define a growing point to implement the motion. For example, below we show how the growing point that moves the pen to the right is implemented.

```
(define-gp (right-pen length)
  (actions
   (--> (start-gp orient-pen 2)
        (start-gp pen-with-inertia:horiz (- length 2)))))
```

To move the pen right, we first orient the pen to start moving in the correct direction, and then we use our model of inertia (i.e. the invaluable ray) to keep the pen moving in the same direction for the remainder of the required distance. In this way, the pheromone responsible for orienting the pen (`dir-p` in this case) need not extend as far as the line we are trying to draw, only as far as its beginning.

```
(define-gp (orient-pen length)
  (material ink)
  (size TEXT-THICKNESS)
  (tropism (and (dia base-line-long) (ortho- dir-pheromone)))
  (actions
   (secrete+ 3 pen-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))

(define-gp (pen-with-inertia:horiz length)
  (material ink)
  (size TEXT-THICKNESS)
  (tropism (and (ortho- pen-pheromone) (dia base-line-long)))
  (actions
   (secrete+ 3 text-pheromone)
   (when ((< length 1)
          (terminate))
         (default
           (propagate (- length 1))))))
```

Observe that `orient-pen` is simply a growing point that would normally move horizontally away from the source of `dir-pheromone`. Since the base-line grows in a direction away from that pheromone, then we associate that direction with the right. The growing point `pen-with-inertia:horiz` is just an embellished ray that tries to move parallel to the base-line.

The other strokes are defined in a similar manner. Notice that we needed the orienting growing point only because there are two possible options for diatropic growth (i.e. parallel to the base-line) at each step of propagation. That is also true for plagiotropic growth (i.e. the oblique strokes). For the strokes that are orthotropic to the `base-line-long` pheromone (i.e. up or down), there is no need for orienting the growing point, so those are simpler to implement.

### 5.7.3 Defining the Letters

Once we have defined all the necessary strokes and lifted-pen growing points, it is a relatively simple matter to compose them into forms that we would recognize as letters. For example, here is the implementation of the letter "M":

```
(define-network (M (txt-in line-in) (txt-out line-out))
  (at txt-in
      (--> (start-gp up-pen CHAR-HEIGHT)
           (--> (start-gp oblique-down-pen CHAR-HEIGHT)
                (secrete+ LINE-HEIGHT dir-pheromone)
                (--> (start-gp oblique-up-pen CHAR-HEIGHT)
                     (--> (start-gp right-pen 2)
                          (--> (start-gp down-pen CHAR-HEIGHT)
                               (->output txt-out)))))))
  (at line-in
      (--> (start-gp base-line CHAR-WIDTH)
           (->output line-out))))
```

The global constants CHAR-HEIGHT and CHAR-WIDTH are self-explanatory: they are simply constants that determine the size of the characters. The names of the growing points that produce the strokes are also self-explanatory. The direction left-to-right is implicit in the `oblique-down-pen` and `oblique-up-pen` names, so the resulting form produces one vertical stroke upwards (`up-pen`), connected to an oblique stroke downwards (`oblique-down-pen`), then an oblique stroke upwards (`oblique-up-pen`), then to a small horizontal stroke, which connects to a vertical stroke downward. Notice that the final point of the down stroke is supposed to be at the same height above the base-line as the input location `txt-in` was. That property must be true for each letter to allow them to be composed to form arbitrary words.

As mentioned earlier, each letter produces not only its text, but it also extends the base-line a little, by the distane `CHAR-WIDTH` in this case. From the initial conditions, the base-line was a little ahead of the text generated. So as each letter gets generated, there is always more line ahead of the letter, thereby preparing the appropriate levels of `base-line-long` pheromone for future letters.

## 5.8 Limitations of Network Abstractions

We have demonstrated a wide range of expressible forms. Most of them have been composed from `ray` and `AB-segment` and growing points like them. The more involved examples used the network abstraction to manage multiple growing points and locations at the same time. It is clear from those examples that the network abstraction is a powerful programming tool. However, it is not without its pitfalls, and we shall now show one way in which the abstraction can lull us into complacency so that we are surprised by the resulting pattern of the program.

Consider the problem of drawing a sector of a circle. Given that we already know how to draw line segments and arcs, drawing a sector should simply be a matter of composing those growing points appropriately. We use the already familiar definitions of `AB-segment` and `arc@C` as primitively defined growing points. We then define network abstractions for line segments and arcs. The `AB-segment` network takes two input locations, the end points of the segment, and produces the termination point of `AB-segment` as the output location. This means that if `AB-segment` terminates normally, the output location is the same as the second input location. Similarly, the `arc` network takes two input locations and produces one output. In the case of `arc`, the first input location identifies the starting location of the arc, and the second input identifies the centre of the arc. The output location is the termination location of `arc@C`. Here are the actual network definitions for the foregoing descriptions.

```
(define-network (segment (a b) (c))
  (at a (connect (start-gp AB-segment)
                 (->output c)))
  (at b (start-gp B-point)))

(define-network (arc (a c) (d))
  (at a (connect (start-gp arc@C length)
                 (->output d)))
  (at c (start-gp centre)))
```

In the definition of arc, length is a globally declared constant. At this point, we are comfortable thinking about the results of arc and AB-segment as an arbitrarily locatable arc and segment respectively. So if we are not careful, we might be tempted to define the sector as a straightforward combination of these network definitions, as follows:

```
(define-network (sector (c a) (d))
  (let-locs ((b) (==> (c a) segment (b)))
    (let-locs ((e) (==> (b c) arc (e)))
      (==> (e c) segment (d)))))
```

The sector network also takes two inputs and produces one output. The first input location indicates the centre of the sector and the second indicates another vertex of the sector. The output location is the output of the second invocation of AB-segment. If all invocations terminate normally, that output location should be the centre of the sector.

For completeness, we present the entry point for the program as well. The entry point takes two inputs, which are passed on to sector, so they have the same interpretations as the inputs for sector. The output is ignored, we explicitly call halt at that location, but that was not necessary since that is the default behaviour in any case.

```
(with-locations (c a)
  (let-locs ((ignored)
             (==> (c a) sector (ignored)))
    (at ignored (halt))))
```

Now when we evaluate our program, we obtain the pattern in Figure 5-12. We get a closed bounded region like we hoped for, but it is not quite right. Why is our second segment not a straight line? We are offered a hint, if we increase the value of length so that the angle subtending the arc becomes obtuse. Figure 5-13 shows the resulting pattern after the increase in length. Now we obtain exactly what we had hoped for.

The reason for this behaviour is that both the centre of the sector and the distinguished second vertex given as an initial condition, invoke the B-point growing point. Therefore the region between those two points contains the superposition of B-pheromone and so we obtain a curve that first heads towards the midpoint of the two sources of B-pheromone and then at some critical point veers towards one of those sources. Which of the sources is chosen is highly dependent on the value of arc-len since that determines where the values of B-pheromone are examined. When there are equal contributions to the level of B-pheromone from both sources, the result is actually a random phenomenon and depends on which neighbour of the point of criticality was chosen first. (So different evaluations yield different patterns.)

In the case of the obtuse angle, there is only one "closest" source, and so we obtain the sector we hoped for. To fix this problem in general, we need a different segment definition

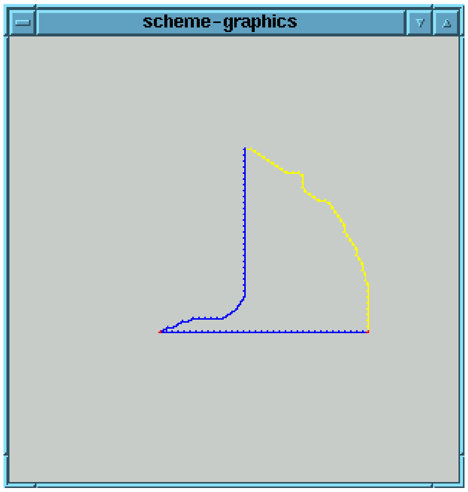


Figure 5-12: Sector looks bad for short arc lengths because of superposition of pheromones.

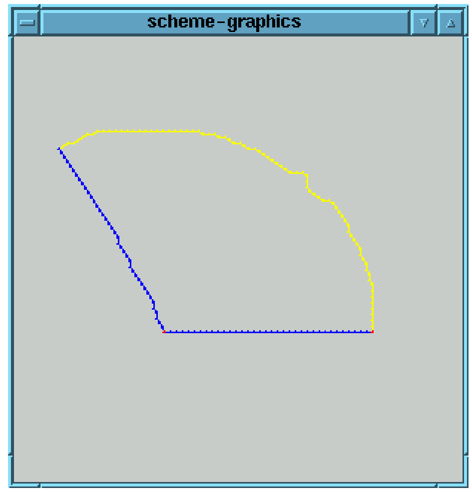


Figure 5-13: Sector looks better for the same initial conditions, but a different arc length.

that uses a different pheromone from `B-pheromone`. Observe that modifying the `secrete+` command to `secrete` in the definition of `B-point` would not solve the problem. When the arc end point was too close to its start point, the second line segment produced by `segment` would target the start point of the arc instead of the centre. That would result in the segment (of a circle), not at all the sector we were hoping for.

Abstracting with networks in GPL is somewhat like abstracting in a high level programming language with procedures that reference globally accessible free variables: those free variables represent a means of communication between the procedures. But reasoning about the behaviours of those procedures require care that false assumptions are not made about the values of those free variables.

# Chapter 6

# Conclusions and Future Work

In closing, we first present a discussion of the results and then discuss possible extensions to GPL. We also describe how ECOLI might be extended to handle mobile particles and actuation abilities, and the how those extensions could be exploited by higher level languages like GPL. We then speculate on how GPL might possibly be physically implemented and what issues would arise on each of those implementations. Finally we summarise the contributions of this work and speculate on its implications for the control of large computational systems.

## 6.1 Discussion of Results

From the results of Chapter 4, we see that in order for our GPL programs to have a high chance of success, there are a few necessary conditions that the GPL domain should meet. First the average value of the neighbourhood size should be relatively large so that the domain will be connected, and so that each particle will have more directions as options for propagation. There is a penalty though, for high neighbourhood sizes because the secretion process is retarded proportionately. Each particle waits a longer time per operation because it has more neighbours to hear from to complete a transaction.

The second necessary property of the distribution is that the variance of the number of neighbours should be low. Low variance in neighbourhood sizes means that the particles are more evenly distributed in space. So if the variance is low, a particle that has many neighbours is just as likely to have a neighbour in one direction as it is to have one in the opposite direction. We also obtained empirical evidence to show that high variance of the neighbourhood sizes cause higher frequencies of miscorrelated secretion distributions. These miscorrelations adversely affect the geometric deductions that the particles make from the pheromone values arriving at their locations.

The third lesson that we have learned is that there are tradeoffs involved with the types of initial conditions given and the success of the growing points that use them. For example, in the implementation of rays versus that of line segments, line segments are invariably well-behaved. Rays, on the other hand, are highly susceptible to the underlying geometry of the domain. The tradeoff is that in order to specify the line segment, we need some non-local information given in the initial conditions. Namely, we need that two distant particles to coordinate propagation and secretion so that they can construct a line segment between them. On the other hand, the ray is capable of growing from highly localised information, and eventually conveying its active site to distant locations.

## 6.2 Related Topics

Computational systems, comprised of numerous elements whose states evolve simultaneously and interdependently often exhibit what is called *emergent behaviour* [23, 18]. The term is used to describe "discovered" global properties that were not anticipated from the inputs to the system. In this dissertation, the type of emergent behaviour we have discussed is patterns of interconnect topology on the Amorphous Computing particles. However, we have generated these patterns by designing the local processes that gave rise to them. The specification of the local processes were obtained *directly* from the topological specification. So our emergent phenomena are not surprises. Consequently, we have refrained from using the term "emergent behaviour", to avoid the connotation of mystery in our outcome.

The morphology of an embryo is a good example of a naturally occurring type of global behaviour that interests us. Specifically of interest are the processes that govern the specialisation and self-organisation of cells in the early stages of embryonic development. Biologists have been able to discover for a few organisms, the genetic role in some of the embryo's developmental functions. For example, there is now strong evidence that the Homeotic (Hox) genes are responsible for the differentiation of body segments in embryonic development (see [19] for a full discussion). This type of knowledge can be used to produce rather impressive mutants, but it has not yet given us enough insight in the mechanisms at work to be able to design with them. In this example, the details of the embryonic development processes are overwhelming and so preclude our co-opting them to suit our needs. However, the example raises an interesting question in the context of this dissertation: if we fully understood the chemical consequences of every genetic mutation, would we be able to "program" an organism's morphology using the techniques developed in the implementation of GPL? Recent work by Knight and Sussman [48] and Weiss [52] indicate that we may be able to implement logic circuits in bacteria, using proteins as signals. Perhaps with time, we will understand how to co-opt a cell's natural processes to emulate a programmable element, much like our Amorphous Computing particle. Then we might be able to bring to bear the lessons learnt from the implementation of GPL.

Pattern formation in chemical Reaction-Diffusion models is an example of topologically constrained global behaviour. Reaction-Diffusion models describe the interaction of two or more chemical species with non-linear reaction kinetic equations. These non-linearities give rise to interesting phase space properties such as limit cycles and chaotic trajectories. Under the appropriate initial conditions, these phase space properties manifest as patterns in the concentrations of the chemical species involved (see [6] for further explanation and some examples). Reaction-Diffusion models have been used to "explain" a wide variety of naturally occuring patterns, from snail shells to leopard spots and zebra stripes. Although these systems have been studied extensively, results are usually either classifications of parameter spaces according to the types of patterns formed, or analyses of the properties of the patterns that emerge from the simulation of the models (see [42] and [32] for examples). However, it appears that it is still unknown how to systematically obtain novel patterns with these systems.

At present the bulk of our knowledge of complex systems, including the synthesized ones like cellular automata or neural nets, is in the form of classifications. That is to say, we can make general statements about what types of behaviours are achievable, and (sometimes) what types are not. The demonstration of the limitations of perceptrons by Minsky and Papert [36] is a good example of that type of knowledge. Wolfram [54] in his characterisation of the complexity of various classes of cellular automata has also contributed to this type of knowledge. While this type of classification helps us avoid attempting the impossible, it usually does very little to facilitate our achieving a specific goal that is possible.

## 6.3 Extensions to GPL

As demonstrated in the chapter 5, GPL is capable of expressing a wide range of patterns that would have been difficult to obtain using only *ad hoc* methods. However, there is still much room for improvement. We discuss here several modifications that could (and probably should) be made to GPL.

### 6.3.1 Other Tropisms

The manner in which tropisms have been implemented are, in fact, only one of several possibilities. There are many alternatives and extensions to the current implementation that would enhance the power and usability of GPL. We now briefly outline some of modifications, and discuss the advantages they would present.

#### Pheromone precedence determined by concentration

In compound tropism expressions that involve many different pheromones, the order of precedence for consideration is determined by the order of appearance of the pheromones in the expression. Therefore, that precedence is statically determined and fixed during a growing point computation.

It might be helpful to permit the precedence of the pheromones to be determined by the strength with which a pheromone is sensed. For example, if a growing point is supposed to move away from each of two pheromone sources with equal weighting, this type of tropism would be more appropriate than the current implementation.

#### Multipropagate

Growing points are able to produce multiple successors to a locus, but only by using multiple calls to `propagate`. An interesting modification to the language would be to add a primitive (or make a modification to the tt propagate form) that would allow more than one neighbour to be selected in a single query round.

Currently, after each `propagate` expression is evaluated, enough time is given for the successor particle to begin its own execution. The difference in behaviour between the current implementation and the proposed modification would be determined by the effect that first successor had on the local pheromone levels. Since we are using pheromone levels to detect directions, we would expect that simultaneous selection of successors would always yield particles that were near each other. If the growing point is self-repelling then serializing the choice of the successors would generally yield particles that were far apart.

#### User-specifiable tie-breaking mechanism

The GPL programmer is unable to specify how the choice of one among many potential successors ought to be made. We assume in the current implementation that the particle that best satisfies the tropism (with the implied precedences) should be the successor. In the case when there are many such particles, we choose one at random.

If the programmer could specify, for example an additional pheromone to use in tie-breaking, then he could potentially prevent bad decisions being made by the growing point. This mechanism is similar to that implemented by the ∼and and ∼or forms, except that a growing point will wait until all the pheromones specified are available before moving. With the mechanism described, it would do so only for the subset specified as it is now, and it would depend on the additional pheromones only in the case of a tie.

### Tropisms that did not stall on missing pheromones

Related to the previous modification, is the option of not forcing growing points to stall if all their pheromones are not present. The syntax of tropisms could be extended to allow some tropisms to assume some default value (presumably zero) for any pheromones in its expression that were absent at the time of propagation. This modification is really a generalization of the previous suggestion, and offers the same benefit.

However it does offer more power, particularly in the case of growing points that are trying to grow away from some pheromone source. For example, we may have a growing point that should avoid, if possible, a certain region marked by a pheromone. Specifying that in the non-strict manner described here, would permit that the region did not have to extend to the location of the growing point before it got started. On the other hand, it would not force the growing point to become stuck if the only options for its successors lay in the undesirable region. Those two extremes are the options available in the current implementation, and the middle ground that this non-strict mechanism would provide is also different from that provided by the previous suggested modification.

### Programmable use of the predecessor's concentrations

In order to reduce the drift in error when simulating diatropic motion, a growing point maintains the pheromone levels of its predecessor. It attempts to find successors that have pheromone levels similar to its own, but also close to its predecessor's level. In the case of diatropism, when propagating to a successor, the predecessor's value is passed on to the successor instead of the current value. That way, each particle in the trajectory attempts to maintain a pheromone level close to that of the original particle's.

There are other uses of the predecessor's information than the simple one described for implementing diatropic growth. A language modification that allowed the programmer to specify how a particle's predecessor's value was used and propagated could be a very powerful tool. A particularly interesting example application would be to simulate the effects of an initial velocity on a growing point. For example, consider a growing point moving diatropically to a pheromone source. Adding a constant to the predecessor's value at each step could simulate an initial velocity towards the source of the pheromone, since a particle's successors's values would tend toward the now higher predecessor value.

In general, by adjusting the predecessor's value for a pheromone in a controlled way (say, adding a number proportional to the number of steps taken), we would be modifying the effective pheromone potential that a particle observed, analogous to the way kinetic energy interacts with potential energy in a closed system.

### Variable tropisms

Yet another variation on tropisms is the ability to have a growing point with a variable tropism. That is, the tropism would vary as the growing point propagated in the domain.

The variations in the tropism could be determined by pheromone levels, detected materials, or even explicitly by a new language statement, for example.

In the current implementation, there are no GPL expressions for examining the pheromone levels at a site. All pheromone responses are encoded as tropism expressions. This property is entirely due to a design decision and could easily be modified to accommodate the feature described in the previous paragraph. The cost of such a modification would be that growing points would have to pass along extra information during propagation that described the current tropism. The advantage of this feature is that programmers would still be restricted from explicitly accessing the pheromone concentrations, but at the same time, they would be able to implement much more flexible growing points.

#### Customizable tropisms

A variation on the previously suggested feature is to have user-specifiable tropisms. The language could be extended to allow tropism expressions to be defined, and subsequently used in growing points in the same way the the current five primitive tropisms are used now. This feature is basically a generalization of all the previously suggested features. As a consequence, it would require the most effort to do correctly.

Each tropism expression is implemented as a combination of a filter and a sorter. A collection of these filters and sorters may be combined by cascading them, or by choosing the output of one of them, or some combination of those two combining mechanisms. So the infrastructure for handling user-defined tropisms is already present. The missing pieces are language primitives for conveniently handling sets of pheromone concentrations, and the appropriate control mechanisms to describe filters and sorters and their compositions.

### 6.3.2 Time sensitivity

At the ECOLI level, we have to explicitly program timeout states and incorporate blocking mechanisms that require the notion of the passage of time to implement the `propagate` and `secrete` primitives of GPL. Therefore those mechanisms are available as programming devices, however they are not exposed to the programmer by GPL. It is plausible to believe that there is merit in providing controlled access to these mechanisms to the programmer, despite the increased complexity in the semantics of GPL. Here we discuss a few scenarios in which such a mechanism would be useful.

#### Time varying secretions

Currently, secretions are all static. That means that the pheromone levels from a secretion must be remembered for the rest of the execution of the GPL program. A more realistic model is to have the pheromone levels disappear by default after some time. It could be a gradual disappearance, like evaporation, or it could be an artificial discrete removal, modeling an extreme case of some pheromone-consuming mechanism.

It is likely that this change in the behaviour of pheromones would not greatly upset the formation of a large set of GPL-constructed patterns, since most of the time, new growing points depend on recently secreted pheromones to direct their growth. In the cases where a pheromone needs to be maintained, we would employ some kind of timed triggering mechanism to replenish the secreted pheromone. Here is where access to the timing mechanisms provided by the ECOLI implementation would be useful.

One version of time-varying secretions that would probably minimally impact the GPL programs already written, is a model where pheromones had three stages of their lifetime. The first stage would be their most volatile: values at sites would rapidly change as the pheromone values were being secreted to their appropriate range. The second stage would be when all the values had settled, and the `secrete` command would be considered completed. These two stages are, in fact, how the current static secretions are done. The third stage would be a pheromone depletion stage, which could be gradual or drastic depending on which version made our programs simpler to analyse.

### Specifying a Timeout Event

The ability to specify a timeout event would be useful in certain circumstances even in the currently static pheromone implementation. For example, when a pheromone is needed for growth but is not present at a site, the growing point there blocks indefinitely until the pheromone presents itself. Sometimes we may want the absence of the pheromone to be treated as a zero value, and for growth to continue (for example, if we suspect that the secreting growing point terminated early or inappropriately). In these instances it would be convenient to be able to declare a timeout event for the `propagate` command to deal appropriately with the absence of the pheromone.

Timeout events could also be used as simple delaying mechanisms. Such a feature would be useful when, for example, we would like to make one growing point follow a pheromone secreted by another. Consider as a specific example, the problem of drawing two parallel lines by starting one ray and a following line segment (with diatropic growth to the guiding pheromone). If the line-segment proceeds too fast for the ray, it may start to circle the ray's active site. A programmable delay would be useful in deliberately lagging the growth of the line-segment.

### Time varying tropisms

A third example of possible time-sensitivity in GPL is in time-varying tropisms. Although such an extension would significantly complicate the analysis of a growing point trajectory, we would then be able to produce oscillatory growing point trajectories, for example, by varying from negative to positive orthotropic growth over time. No doubt much more complicated trajectories would be specifiable, and perhaps there is a class of such trajectories that would be worth the increased semantic complexity. Although, at this time, we do not perceive of any such class, we shall outline how one might implement time varying tropisms if it were desired.

First, to clarify a technical point, we are assuming that by time varying tropisms, we mean that the tropism expression has a time dependent meaning. We do not mean that the actions attribute is allowed to specify a variation on the tropism based on a timed event — that would be a combination of the preceding time-sensitivity example and one of the variable tropism models outlined in the previous section. So the required extensions to the implementation would be that the growing point would have to maintain time across its trajectory, and the tropism filters and sorters would have to take a time parameter that would influence their behaviour in some specified way, appropriate to the specific tropism being represented.

Notice that this proposed extension would be using the timing mechanisms in a different way from the preceding two extensions. In those extensions, time was being maintained on a single particle. In this extension, the time on one particle would have to be combined somehow with the number of steps taken from the original invoking particle. Therefore, the meaning of the time value here is very different from that of the preceding two proposals, something that would have to be borne in mind if some combination of these proposals were being implemented.

### 6.3.3 Denotable Growing Point Paths

Perhaps the most important contribution of GPL was to provide, through the growing point metaphor, a means of describing trajectories in a coordinate-free manner. Now that we can express the connectivity of our patterns, it seems the next logical step is to try to manipulate the patterns we produce as if they were themselves individual entities.

#### Potential Applications

The ability to denote entire trajectories (instead of just the active site) would offer us a considerable amount of expressive power. For example, with a denotable growing point path, we could potentially:

- express non-local constraints
- allow its motion
- allow its growth

**Trajectories that enforce constraints** A growing point always labels a connected component of the domain graph, therefore a denotable trajectory could be used as a conduit for sharing information between two distant sites in the domain. That shared information could be used in various ways including maintaining some constraint between the two sites.

The previous example could be generalised from a constraint on the end-points of a trajectory to a constraint along its entire length. For example, it may be desirable to maintain a specific level of some pheromone at the neighbouring sites all along the trajectory. If pheromone levels are time-varying, then the locations along the trajectory could secrete enough pheromone to maintain the desired concentration level.

**Mobile trajectories:** Mobile trajectories could be used to enforce constraints on the growing point path and its immediate neighbourhood in situations where the environment could not be changed easily. For example, suppose that a pattern of a circuit layout contained two paths of polysilicon that were too close to each other. If GPL had mobile trajectories, then the polysilicon pieces could adjust their positions to increase the space between them, without affecting the rest of the circuit.

**Growing trajectories:** Trajectories that grow are probably the key to obtaining fine Geometric control. Given a particular Geometric pattern, we could first construct a topologically correct miniature version of it, and slowly expand it to the desired proportions. The irregularities in the underlying Geometry would be compensated for by constraining the growth to maintain the topology of the trajectory and to be sensitive to the space surrounding it. In that way, static trajectories could act as barriers, discouraging growth in certain areas (by their presence) while encouraging it in others by their absence.

**Growing, mobile trajectories that constrain:** Perhaps the most benefit from denotable growing point trajectories, would come not from any one of the above mentioned applications, but from a combination of all three! Mobile growing point trajectories that had the ability to grow could now express a wide variety of constraints: both static and dynamic for itself and its immediate environment, and even global constraints for the entire domain by sampling it in various places as it moved.

#### Implementation Issues

The first modification necessary would be that labels and pheromones no longer be monotonically established. The growing point trajectory exists only in the interpretations of the labeling of the domain. Therefore, trajectories would probably be very limited as entities if labelings were not allowed to be undone. Since pheromones are instrumental in the formation of the trajectories, it seems reasonable to expect that they would play an important role in their maintenance as well. Therefore it seems that if we allowed labels to change, we would also want to be able to reflect the change in labels by a corresponding change in pheromone levels.

The representation of connectivity would probably also have to be changed. Currently, each locus keeps track of its predecessor and all of its successors for each growing point. Perhaps that information will need to be augmented with more information about the neighbours, for example invariant properties that should be maintained across trajectory connections. Currently the tropism is an example of such a property, but it is not a persistent constraint. It may be that in order to control an entire trajectory, the programmer would need to specify additional tropism-like constraints, or perhaps the tropism expressions could be reinterpreted as persistent constraints that must be obeyed for longer than simply the moment of establishing a connection.

It seems almost certain that in order to allow the programmer to express non-trivial properties of growing point trajectories, that there will be some sort of communication mechanism required for propagating information only along the connections in the trajectory. We already see some elements of this in the implementation of the `let-locs` and `serialize` forms, where the connectivity is used to implement a backtracking mechanism. This modification is perhaps the simplest to accomplish, since it is probably readily supportable on the communications infrastructure already assumed to be in place on the domain.

Once we have a communication mechanism in place for endpoints of a trajectory, we would then be able to use the information in a growing point becoming stuck in a non-local manner. Coupling that information with the non-monotonic assignment of labels could lead to growing points repeatedly attempting to produce a pattern. So denotable growing points could also improve the reliability with which our current programs achieve their patterns.

## 6.4 Reliability Issues

The Growing Point Language is implemented in an extremely friendly environment. The particles are well-behaved (i.e. they compute according to their specfications), never die, and communicate without errors. The Amorphous Computing model, though, assumes that none of those properties can be taken for granted. We now discuss how the implementation of GPL could be adjusted to accommodate some more of the Amorphous Computing assumptions.

### 6.4.1 Communication Errors

There are only two aspects of the implementation of GPL that are dependent on communication: secretions and propagation. In a noisy environment where messages can be lost or corrupted, we probably would first implement a layer of communication that attempted to present an abstraction of error-free communication. However, we really need robust communication only when we are propagating a growing point. In the case of secretion, we would probably want to leverage the broadcast mechanism that is assumed in the model, and so it would be difficult to confirm that every broadcast reaches every neighbour. The noisiness of the messages being received is probably not really an issue since we can probably choose an appropriate encoding that permits recovery from most errors in transmission. So we shall consider the effect of messages that either arrive at their destination correctly or fail to arrive altogether.

In the case of secretions, lost messages are probably not a big problem. A particle that loses a message indicating how to update its pheromone value is still likely to hear from a large fraction of its neighbours, each contributing a little towards its pheromone value. Therefore, secretions will be distorted due to lost messages, but probably not in a critical way.

In the case of propagation of a growing point, lost messages can have a more serious effect since the growing point is propagated to only one location. This problem is already partly addressed in the implementation. One consequence of a lost message in the process of polling for pheromone values is that the deciding particle is unable to find any "good" directions to choose based on its responses. In such a situation, it would become stuck. The implementation tries to delay assuming that stuck state by repeating the polling process a fixed number of times (three in the actual implementation). The other consequence of a lost message is that the selected particle does not receive its notification of succession. This problem would be solved by implementing a handshaking mechanism so that both the deciding particle and the succeeding particle would be aware of their mutual state changes. So all in all, the problem of lost messages is unlikely to completely sabotage the growing point paradigm.

### 6.4.2 Increased Survivability of Growing Points

Growing points are currently implemented so that a single particle is responsible for propagation. In an environment where the particles are fragile (a default assumption of Amorphous Computing) that represents a real problem for the successful growth of a growing point. However, it seems that not much is required to improve the implementation.

When a particle decides to propagate the growing point property, it polls all its neighbours for the appropriate pheromone values and then makes a decision, selecting one of those neighbours as its successor. The successor is not the only particle to hear the message of succession though, and therein lies a solution. When the succeeding particle acknowledges the propagation, there will typically be a set of neighbours who hear both the succession message as well as the acknowledgement. Those particles could then act as potential stand-ins for the selected particle, in case it should die before being able to complete its propagation task. This extension would require a few more messages to be sent per propagation transaction, but that number seems to be small. The new successor particle would need to periodically notify its backup particles that it was still alive until it had dispatched its responsibilities. It would also need to indicate when it had done so, since a neighbouring particle would not be able to distinguish between a succeeding particle deciding to terminate, and one that failed to propagate when it should have.

### 6.4.3 Incorrect computations

One problem that seems a little harder to solve is to make growing points resilient to malicious particles that interfere with computations by producing incorrect answers. This failure mode is known as Byzantine failure, and it is much harder to detect and to compensate for than the problem of dying particles or lossy communication. One way to get around this problem is to perform redundant computations for each step of the process on many neighbouring particles. The particles must agreement before the process can proceed any further. In Distributed Computing, there are algorithms to obtain agreement in the presence of Byzantine failures [34]. Some of these algorithms are probably directly portable to the Amorphous Computing environment, but there will be complications. It is known that the number of redundant processors needs to be at least three times the number of failures. Since the neighbourhood sizes are fixed, then the number of failures tolerable will necessarily be limited. Perhaps it is possible to incorporate more particles than a single neighbourhood size to solving this problem, but it is bound to be very complicated, and likely to be very costly in terms of the number of messages that need to be sent.

## 6.5 Extensions to ECOLI

There are many potentially useful extensions to ECOLI , depending on the specific details of its intended application. Here we shall restrict our discussion to the extensions of ECOLI that would impact our implementation of GPL.

### 6.5.1 The consequences of mobile particles

The connection matrix would become time varying, likely introducing significant complications in our program analysis. The time to compute neighbourhood sensitive results would now be constrained. Here are some of the issues that would be involved with dealing with a mobile-particle medium.

- The method for growing point propagation might need enhancement because there may not be enough time to get pheromone values, apply the tropism, and then notify the successor before the neighbourhood changed. The modification could involve constantly monitoring the neighbourhood while the internal computations of the tropism were occurring (probably bad). Perhaps better would be to choose the first from the resulting list of neighbour-pheromone associations that is still a neighbour.

- Our models for improving robustness may need modification, because the methods we have employed to improve robustness usually involve taking more time to perform redundant computation. e.g. waiting for a particle's pheromones when it is not ready.

- If we need the status of the neighbourhood, we will have to recompute it every time it is needed. (This does not affect our implementation of GPL).

- The assumptions made inside the dictionary for a process may not be valid, thereby complicating the way the handlers are written. For example, we would have to make sure that a message normally received in the middle of some protocol did not have disastrous effects if it were received (as a result of the time-varying neighbourhood) in a different program context.

### 6.5.2 Sensing and Actuation

There is no reason why GPL secretions have to be implemented as strictly computational processes. If a particle has the ability to sense its environment as well as affect it, then those two abilities may be sufficient to *physically* implement secretions (or perhaps a useful subset of the specified behaviour).

For example, imagine an amorphous computer whose particles are located on a flexible membrane, fixed taught at its boundary. Suppose that each particle is equipped with an actuator that, by contraction, can stretch the membrane radially towards its location. Suppose also that the particles have a sensor that can detect local changes in the length of the membrane, so that within a few communication hops of another particle applying its actuator, the stretching effect is detectable. Then we may regard the distribution of sensor measurements in the locale of a contraction as the "pheromone" levels of a secretion. (This holds under the assumption that the magnitude of the strain on the rubber sheet, measured locally by each particle, is subject to Laplace's equation on constant valued boundary conditions.)

In this setting, we would be able to evaluate GPL programs that did not require secretions with extents beyond the area of effect of the particle's actuators. Secretions would be "computed" potentially faster (at the speed of sound instead of the speed of the communications protocol). They would probably also be superior approximations to radially symmetric functions than the straightforward "single-assignment" computational simulations presented in Chapters 3 and 4.

## 6.6 Concluding Remarks

By presenting a general method for interpreting GPL programs locally, we have shown how particles that are programmed uniformly can coherently differentiate their behaviours to obtain a global objective. As it turns out, it is the very act of interpretation of GPL code that gives rise to the differentiation of behaviour. Therefore, control over the differentiation phenomenon across the entire collection of particles is placed squarely in the programmer's hands.

The sophistication of our control represents a significant advance in our understanding of the connections between a complex system's global behaviour and its individuals' actions. It is therefore likely to improve our understanding of natural systems. For example, the cells in an embryo constitute a complex system, and the morphology of the organism is an example of an interesting global behaviour. The methods we have presented allow us to engineer caricatures of that behaviour and of the processes that underlie it. These cartoon caricatures can then help us to formulate testable, new hypotheses about the actual developmental process of morphology in organisms.

Another example of an interesting global behaviour in a natural system is the three dimensional structure of a protein. There, the system's elements are the constituent amino acids of the protein (and perhaps auxiliary catalytic proteins called chaperones), and their states of interest are their spatial configurations (see [12, 41] for a discussion of protein folding). Is it possible that there is a growing point formulation for the trajectory of each amino acid residue so that the final state of our computation corresponds to the native conformation of the protein? We do not yet understand enough about protein folding to answer that question. But thinking about how to answer it can lead us to exploring novel approaches, for example, we might search for better ways of approximating the interactions between residues of a protein than are currently known. In that way, our engineering techniques can produce new research directions for solving important scientific problems.

Since our methods require us to fully understand the processes of the system's elements we must, for the moment, restrict our applications to synthetic systems. However, these two examples hint at the possibilities that become accessible to us when we discover how to manipulate entities like cells and molecules in the same way we manipulate computer programs. Therefore, the techniques we have presented not only inspire new research directions in Science, but they also promise tremendous advances in Engineering techniques.

# Appendix A

# GPL code listings

## A.1 Lines and Arcs

**segment-w1.gpl**

```
;;; -*- Scheme -*-

(define-growing-point (A-to-B-segment)
  (material A-material)
  (size 1)
  (tropism (ortho+ B-pheromone))
  (for-each-step
   (when ((sensing? B-material)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (B-point)
  (material B-material)
  (size 1)
  (for-each-step
   (secrete+ 15 B-pheromone)))

(color
 ((B-material) "red")
 ((A-material) "red"))

(with-locations
 (a b)
 (at a (start-growing-point A-to-B-segment))
 (at b (start-growing-point B-point)))
```

**ray.gpl**

```
;;; -*- Scheme -*-

(define-growing-point (ray life)
  (material ray-mat)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< life 1)
  (terminate))
 (default
   (propagate (- life 1))))))

(color
 ((ray-mat) "red"))

(with-locations
 (base-pt source)
 (at base-pt
     (secrete+ 3 self-pheromone))
 (at source
     (start-growing-point ray 20)))
```

**ray4.gpl**

```
;;; -*- Scheme -*-

(define-gp (ray-A life)
  (material ray-mat)
  (tropism (and (ortho- self-pheromone-B) (ortho- self-pheromone-A)))
  (size 0)
  (actions
   (when ((< life 1)
  (terminate))
 (default
   (secrete 3 self-pheromone-A)
   (connect
    (propagate 0)
    (start-gp ray-B (- life 1)))))))

(define-gp (ray-B life)
  (material ray-mat)
  (tropism (and (ortho- self-pheromone-A) (ortho- self-pheromone-B)))
  (size 0)
  (actions
   (when ((< life 1)
  (terminate))
 (default
   (secrete 3 self-pheromone-B)
   (connect
    (propagate 0)
    (start-gp ray-A (- life 1)))))))

(color
 ((ray-mat) "red"))

(with-locations
 (base-pt source)
 (at base-pt
     (secrete 3 self-pheromone-A))
 (at source
     (start-gp ray-B 20)))
```

**arc.gpl**

```
;;; -*- Scheme -*-

(constant radius 10)

(define-growing-point (centre)
  (material C-material)
  (size 0)
  (for-each-step
   (secrete+ radius C-pheromone)
   (terminate)))

(define-growing-point (arc@C length)
  (material arc-material)
  (size 0)
  (tropism (and (ortho- arc-pheromone) (dia C-pheromone)))
  (for-each-step
   (secrete+ 3 arc-pheromone)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(color
 ((C-material) "red")
 ((arc-material) "yellow"))

(with-locations (c a)
  (at c (start-growing-point centre))
  (at a (start-growing-point arc@C radius)))
```

### A.1.1 Euclidean Construction

**sixty-degrees2.gpl**

```
;;; -*- Scheme -*-

(Constant radius 20)

(define-growing-point (base-line)
  (material base-line-material)
  (size 0)
  (tropism (ortho+ base-line-ph))
  (for-each-step
   (secrete+ 3 line-pheromone)
   (when ((sensing? base-line-stop-m) (terminate))
 (default (propagate)))))

(define-gp (base-line-stop)
  (material base-line-stop-m)
  (size 0)
  (actions
   (secrete+ RADIUS base-line-ph)
   (terminate)))

(define-growing-point (segment-to-src)
  (material segment-material)
  (size 0)
  (tropism (ortho+ src-pheromone))
  (for-each-step
   (when ((sensing? base-line-material)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (src)
  (for-each-step
   (secrete+ (+ radius 1) src-pheromone)))

(define-growing-point (dest)
  (for-each-step
   (secrete+ (+ radius 1) dest-pheromone)))

(define-growing-point (dest-arc)
  (material dest-arc-material)
  (size 1)
  (tropism (and (ortho- dest-arc-p) (dia src-pheromone)))
  (for-each-step
   (secrete+ 3 dest-arc-p)
```

```
   (when ((sensing? src-arc-material)
  (start-growing-point segment-to-src))
 (default
   (propagate)))))

(define-growing-point (src-arc)
  (material src-arc-material)
  (size 1)
  (tropism (and (ortho- src-arc-p) (dia dest-pheromone)))
  (for-each-step
   (secrete+ 3 src-arc-p)
   (when ((sensing? dest-arc-material)
  (start-growing-point segment-to-src))
 (default
   (propagate)))))

(color
 ((base-line-material) "blue")
 ((segment-material) "red")
 ((src-arc-material) "yellow")
 ((dest-arc-material) "yellow")
 ((ray-material) "cyan"))

(with-locations
 (a b)
 (at a
     (connect (start-growing-point base-line)
      (start-growing-point dest)
      (start-growing-point dest-arc))
     (start-growing-point src)
     (start-growing-point src-arc))
 (at b (start-gp base-line-stop)))
```

## A.2 CMOS Circuit Layouts

**inverter.gpl**

```
;;; -*- Scheme -*-

(define-growing-point (vdd-rail-1)
  (material metal)
  (size 1)
  (for-each-step
   (secrete+ 3 metal-pheromone)
   (secrete 17 vdd-pheromone)))

(define-growing-point (vdd-rail-2 vss-length length)
  (material metal)
  (size 1)
  (tropism (ortho- metal-pheromone))
  (for-each-step
   (when ((< length 1)
  (start-growing-point vss-starter vss-length))
 (default
   (secrete+ 3 metal-pheromone)
   (secrete 17 vdd-pheromone)
   (propagate vss-length (- length 1))))))

(define-growing-point (vss-starter length)
  (tropism (ortho+ vss-beacon-pheromone))
  (for-each-step
   (when ((sensing? vss-marker)
  ;;(start-growing-point vss-rail-2 length)
  (start-growing-point vss-rail-2 (* 2 length))
  (secrete 17 vss-pheromone))
 (default
   (propagate length)))))

(define-growing-point (vss-rail-1 range)
  (material metal vss-marker)
  (size 1)
  (tropism (dia vdd-pheromone))
  (for-each-step
   (secrete range vss-beacon-pheromone)))

(define-growing-point (vss-rail-2 length)
  (material metal)
  (size 1)
  (tropism (and (dia vdd-pheromone) (ortho- metal-pheromone)))
  (for-each-step
```

```
   (when ((< length 1)
  (terminate))
 (default
   (secrete+ 3 metal-pheromone)
   (secrete 17 vss-pheromone)
   (propagate (- length 1))))))

(define-growing-point (poly-input-contact life)
  (material poly contact)
  (size 1)
  (for-each-step
   (secrete (* 2 life) input-pheromone)
   (start-growing-point drop-contact)
   (start-growing-point init-poly life)))

(define-growing-point (init-poly life)
  (material poly)
  (size 1)
  (tropism (and (ortho- fork-pheromone)
(and (dia vdd-pheromone) (dia vss-pheromone))))
  (for-each-step
   (when ((< life 1)
  (secrete 10 fork-pheromone)
  (start-growing-point up-poly 5)
  (start-growing-point down-poly 5))
 (default
   (propagate (- life 1))))))

(define-growing-point (up-poly life)
  (material poly)
  (size 1)
  (tropism (~and (ortho+ vdd-pheromone) (ortho- input-pheromone)))
  (for-each-step
   (when ((< life 1)
  (start-growing-point poly-p-diffusion 5))
 (default
   (propagate (- life 1))))))

(define-growing-point (down-poly life)
  (material poly)
  (size 1)
  (tropism (~and (ortho+ vss-pheromone) (ortho- input-pheromone)))
  (for-each-step
   (when ((< life 1)
  (start-growing-point poly-n-diffusion 5))
 (default
   (propagate (- life 1))))))
```

```
(define-growing-point (poly-p-diffusion life)
  (material poly)
  (size 1)
  (tropism (and (dia vdd-pheromone) (ortho- fork-pheromone)))
  (for-each-step
   (when ((< life 1)
  (terminate))
 ((= life 2)
  (start-growing-point p-diff-up)
  (start-growing-point p-diff-down)
  (propagate (- life 1)))
 (default
   (propagate (- life 1))))))

(define-growing-point (poly-n-diffusion life)
  (material poly)
  (size 1)
  (tropism (and (dia vss-pheromone) (ortho- fork-pheromone)))
  (for-each-step
   (when ((< life 1)
  (terminate))
 ((= life 2)
  (start-growing-point n-diff-down)
  (start-growing-point n-diff-up)
  (propagate (- life 1)))
 (default
   (propagate (- life 1))))))

(define-growing-point (p-diff-up)
  (material p-diffusion)
  (size 1)
  (tropism (ortho+ vdd-pheromone))
  (for-each-step
   (when ((sensing? metal)
  (start-growing-point drop-contact))
 (default
   (propagate)))))

(define-growing-point (n-diff-down)
  (material n-diffusion)
  (size 1)
  (tropism (ortho+ vss-pheromone))
  (for-each-step
   (when ((sensing? metal)
  (start-growing-point drop-contact))
 (default
   (propagate)))))
```

```
(define-growing-point (p-diff-down)
  (material p-diffusion)
  (size 1)
  (tropism (or (ortho+ n-diff-pheromone) (ortho- vdd-pheromone)))
  (for-each-step
   (secrete 17 p-diff-pheromone)
   (when ((sensing? n-diffusion)
  (start-growing-point drop-contact)
  ;;(start-growing-point poly-input-contact 5)
  ;; start next circuit element here
  (terminate))
 (default
   (propagate)))))

(define-growing-point (n-diff-up)
  (material n-diffusion)
  (size 1)
  (tropism (or (ortho+ p-diff-pheromone) (ortho- vss-pheromone)))
  (for-each-step
   (secrete 17 n-diff-pheromone)
   (when ((sensing? p-diffusion)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (drop-contact)
  (material contact)
  (size 1)
  (for-each-step
   (terminate)))

(color
 ((contact) "black")
 ((poly p-diffusion) "orange")
 ((poly n-diffusion) "SeaGreen3")
 ((poly) "red")
 ((p-diffusion) "yellow")
 ((n-diffusion) "green")
 ((metal) "blue"))

(with-locations
 (vdd-1 vdd-2 vss poly-start)
 (at vdd-1 (start-growing-point vdd-rail-1))
 (at vdd-2 (start-growing-point vdd-rail-2 8 15))
 (at vss (start-growing-point vss-rail-1 25))
 (at poly-start
     (secrete 5 fork-pheromone)
     (start-growing-point poly-input-contact 5)))
```

**circuits-lib.gpl**

```
;;; -*- Scheme -*-

(constant RAIL-WIDTH 1)
(constant RAIL-SEP 40)
(constant UNIT-LEN 5)

(constant FET-HEIGHT 4)

;;;; Colour Scheme

(color
 ((contact) "black")
 ((poly p-diffusion) "orange")
 ((poly n-diffusion) "SeaGreen3")
 ((poly metal) "purple")
 ((poly) "red")
 ((p-diffusion) "yellow")
 ((n-diffusion) "green")
 ((metal) "blue"))

;;;; Rails

(define-growing-point (vdd-rail-1 length)
  (material metal)
  (size RAIL-WIDTH)
  (tropism (and (ortho- rail-support-p) (ortho- vdd-short)))
  (for-each-step
   (secrete+ 3 vdd-short)
   (secrete RAIL-SEP vdd-long)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(define-growing-point (vdd-rail-2 length)
  (material metal)
  (size RAIL-WIDTH)
  (tropism (and (ortho- vdd-short) (dia vss-long)))
  (for-each-step
   (secrete+ 3 vdd-short)
   (secrete RAIL-SEP vdd-long)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(define-growing-point (vss-rail-1 length)
  (material metal)
  (size RAIL-WIDTH)
```

```
  (tropism (and (ortho- vss-short) (dia vdd-long)))
  (for-each-step
   (secrete+ 3 vss-short)
   (secrete RAIL-SEP vss-long)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(define-growing-point (vss-rail-2 length)
  (material metal)
  (size RAIL-WIDTH)
  (tropism (and (ortho- rail-support-p) (ortho- vss-short)))
  (for-each-step
   (secrete+ 3 vss-short)
   (secrete RAIL-SEP vss-long)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(define-growing-point (vdd-starter-beacon)
  (material serializer-material)
  (for-each-step
   (secrete (+ RAIL-SEP UNIT-LEN) vdd-starter-attractor)))

(define-growing-point (vdd-starter)
  (tropism (ortho+ vdd-starter-attractor))
  (for-each-step
   (when ((sensing? serializer-material)
  (terminate))
 (default (propagate)))))

(define-growing-point (vss-starter-beacon)
  (material serializer-material)
  (for-each-step
   (secrete (+ RAIL-SEP UNIT-LEN) vss-starter-attractor)))

(define-growing-point (vss-starter)
  (tropism (ortho+ vss-starter-attractor))
  (for-each-step
   (when ((sensing? serializer-material)
  (terminate))
 (default (propagate)))))

(define-network (rails-1 (vdd-in vss-in) (vdd-out vss-out))
  (at vdd-in
      (secrete UNIT-LEN rail-support-p)
      (connect (start-growing-point vdd-rail-1 UNIT-LEN)
       (connect (start-growing-point vss-starter)
(connect (start-growing-point vss-rail-1 UNIT-LEN)
 (->output vss-out))
```

```
       (->output vdd-out)))
  (at vss-in
      (start-growing-point vss-starter-beacon)))

(define-network (rails-2 (vdd-in vss-in) (vdd-out vss-out))
  (at vdd-in
      (start-growing-point vdd-starter-beacon))

  (at vss-in
      (secrete UNIT-LEN rail-support-p)
      (connect (start-growing-point vss-rail-2 UNIT-LEN)
       (connect (start-growing-point vdd-starter)
(connect (start-growing-point vdd-rail-2 UNIT-LEN)
 (->output vdd-out)))
       (->output vss-out))))

(define-network (rails (vdd-in vss-in) (vdd-out vss-out))
  (==> (vdd-in vss-in) rails-1 rails-2 (vdd-out vss-out)))

(define-network (init-rails (vdd-1 vss-1) ())
  (at vdd-1 (secrete+ 3 vdd-short))
  (at vss-1 (secrete+ 3 vss-short)))

;;;; Circuit Elements

;;; Polysilicon

(define-growing-point (poly-left->right length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- dir-pheromone)
(and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (poly-with-inertia length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- poly-pheromone)
(and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-B)
  (for-each-step
```

```
   (secrete 2 poly-id-A)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (horiz-poly length)
  (material poly)
  (size N-WIDTH)
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (connect (start-growing-point poly-left->right 2)
    (start-growing-point poly-with-inertia (- length 2)))))

(define-growing-point (up-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (or (ortho+ vdd-long) (ortho- vss-long)))
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (down-poly length)
  (material poly)
  (size N-WIDTH)
  (tropism (ortho+ vss-long))
  (avoids poly-id-B)
  (for-each-step
   (secrete 2 poly-id-A)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (poly-left->right-B length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- dir-pheromone)
(and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-A)
  (for-each-step
   (secrete 2 poly-id-B)
```

```
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (poly-with-inertia-B length)
  (material poly)
  (size N-WIDTH)
  (tropism (and (ortho- poly-pheromone)
(and (dia vdd-long) (dia vss-long))))
  (avoids poly-id-A)
  (for-each-step
   (secrete 2 poly-id-B)
   (secrete+ 3 poly-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (horiz-poly-B length)
  (material poly)
  (size N-WIDTH)
  (avoids poly-id-A)
  (for-each-step
   (secrete 2 poly-id-B)
   (connect (start-growing-point poly-left->right-B 2)
    (start-growing-point poly-with-inertia-B (- length 2)))))

(define-growing-point (up-poly-B length)
  (material poly)
  (size N-WIDTH)
  (tropism (or (ortho+ vdd-long) (ortho- vss-long)))
  (avoids poly-id-A)
  (for-each-step
   (secrete 2 poly-id-B)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (down-poly-B length)
  (material poly)
  (size N-WIDTH)
  (tropism (ortho+ vss-long))
  (avoids poly-id-A)
  (for-each-step
   (secrete 2 poly-id-B)
```

```
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

;;; Metals

(define-growing-point (metal-left->right length)
  (material metal)
  (size N-WIDTH)
  (tropism (and (and (dia vdd-long) (dia vss-long))
(ortho- dir-pheromone)))
  (for-each-step
   (secrete+ 3 metal-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (metal-with-inertia length)
  (material metal)
  (size N-WIDTH)
  (tropism (and (ortho- metal-pheromone)
(and (dia vdd-long) (dia vss-long))))
  (for-each-step
   (secrete+ 3 metal-pheromone)
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (horiz-metal length)
  (material metal)
  (size N-WIDTH)
  (for-each-step
   (connect (start-growing-point metal-left->right 1)
    (start-growing-point metal-with-inertia (- length 1)))))

(define-growing-point (up-metal length)
  (material metal)
  (size N-WIDTH)
  (tropism (ortho+ vdd-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))
```

```
(define-growing-point (down-metal length)
  (material metal)
  (size N-WIDTH)
  (tropism (ortho+ vss-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (metal->vss)
  (material metal)
  (size N-WIDTH)
  (tropism (~and (ortho+ vss-long) (ortho- dir-pheromone)))
  (for-each-step
   (when ((sensing? metal)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (metal->vdd)
  (material metal)
  (size N-WIDTH)
  (tropism (~and (ortho+ vdd-long) (ortho- dir-pheromone)))
  (for-each-step
   (when ((sensing? metal)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (drop-contact radius)
  (material contact)
  (size radius))

(define-network (via (input) (output))
  (at input
      (start-growing-point drop-contact N-WIDTH)
      (->output output)))

;;; P-diffusion

(define-growing-point (up-p-diff length)
  (material p-diffusion)
  (size P-WIDTH)
  (tropism (ortho+ vdd-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
```

```
 (default
   (propagate (- length 1))))))

(define-growing-point (down-p-diff length)
  (material p-diffusion)
  (size P-WIDTH)
  (tropism (ortho+ vss-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (p-diff->vdd)
  (material p-diffusion)
  (size P-WIDTH)
  (tropism (~and (ortho+ vdd-long) (ortho- dir-pheromone)))
  (for-each-step
   (when ((sensing? metal)
  (terminate))
 (default
   (propagate)))))

;;; N-diffusion

(define-growing-point (up-n-diff length)
  (material n-diffusion)
  (size N-WIDTH)
  (tropism (ortho+ vdd-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (down-n-diff length)
  (material n-diffusion)
  (size N-WIDTH)
  (tropism (ortho+ vss-long))
  (for-each-step
   (when ((< length 1)
  (terminate))
 (default
   (propagate (- length 1))))))

(define-growing-point (n-diff->vss)
  (material n-diffusion)
  (size N-WIDTH)
```

```
  (tropism (~and (ortho+ vss-long) (ortho- dir-pheromone)))
  (for-each-step
   (when ((sensing? metal)
  (terminate))
 (default
   (propagate)))))

;;; FETs

(define-network (vert-p-fet (gate) (src drain))
  (at gate
      (start-growing-point horiz-poly 2)
      (connect (start-growing-point up-p-diff FET-HEIGHT)
       (->output src))
      (connect (start-growing-point down-p-diff FET-HEIGHT)
       (->output drain))))

(define-network (vert-n-fet (gate) (src drain))
  (at gate
      (start-growing-point horiz-poly 2)
      (connect (start-growing-point up-n-diff FET-HEIGHT)
       (->output src))
      (connect (start-growing-point down-n-diff FET-HEIGHT)
       (->output drain))))

(define-network (vert-p-fet-B (gate) (src drain))
  (at gate
      (start-growing-point horiz-poly-B 2)
      (connect (start-growing-point up-p-diff FET-HEIGHT)
       (->output src))
      (connect (start-growing-point down-p-diff FET-HEIGHT)
       (->output drain))))

(define-network (vert-n-fet-B (gate) (src drain))
  (at gate
      (start-growing-point horiz-poly-B 2)
      (connect (start-growing-point up-n-diff FET-HEIGHT)
       (->output src))
      (connect (start-growing-point down-n-diff FET-HEIGHT)
       (->output drain))))

;;; Serializers and segments to connect FETs

(define-growing-point (short-range-beacon range)
  (material beacon-material)
  (for-each-step
   (secrete range beacon-pheromone)))
```

```
(define-growing-point (n-diff-seg width)
  (material n-diffusion)
  (size width)
  (tropism (ortho+ beacon-pheromone))
  (for-each-step
   (when ((sensing? beacon-material)
  (terminate))
 (default (propagate width)))))

(define-growing-point (p-diff-seg width)
  (material p-diffusion)
  (size width)
  (tropism (ortho+ beacon-pheromone))
  (for-each-step
   (when ((sensing? beacon-material)
  (terminate))
 (default (propagate width)))))

(define-growing-point (invisible-seg)
  (size 0)
  (tropism (ortho+ beacon-pheromone))
  (for-each-step
   (when ((sensing? beacon-material) (terminate))
 (default (propagate)))))

(define-growing-point (input-starter-beacon dist)
  (material serializer-material)
  (for-each-step
   (secrete dist input-starter-attractor)))

(define-growing-point (input-starter)
  (tropism (ortho+ input-starter-attractor))
  (for-each-step
   (when ((sensing? serializer-material)
  (terminate))
 (default (propagate)))))

(define-network (input-serializer (first second-in) (second-out))
  (at first
      (--> (start-growing-point input-starter)
   (->output second-out)))
  (at second-in
      (start-growing-point input-starter-beacon (+ (* 4 UNIT-LEN) RAIL-SEP))))
```

**net-inverter2.gpl**

```
;;; -*- Scheme -*-

(include "circuits-lib")

(constant N-WIDTH 1)
(constant P-WIDTH 1)
(constant POLY-LEN 5)
(constant INV-HEIGHT (+ FET-HEIGHT 1))

(define-growing-point (down-join-metal)
  (material metal)
  (size N-WIDTH)
  (tropism (~and (ortho+ up-join-pheromone) (ortho- vdd-long)))
  (for-each-step
   (when ((sensing? join-marker)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (up-join-metal dist length)
  (material metal join-marker)
  (size N-WIDTH)
  (for-each-step
   (secrete+ dist up-join-pheromone)
   (connect (start-growing-point up-metal 2)
    (connect (start-growing-point horiz-metal (- length 2))
     (secrete+ (* 3 POLY-LEN) dir-pheromone)
     (start-growing-point metal-with-inertia 2)))))

(define-network (inverter-fets (p-fet n-fet) (common-out))
  (let-locs ((p-src p-drain)
     (==> (p-fet) vert-p-fet (p-src p-drain)))
    (at p-src
(start-growing-point drop-contact N-WIDTH)
(start-growing-point metal->vdd))
    (at p-drain
(start-growing-point drop-contact N-width)
(start-growing-point down-join-metal)))

  (let-locs ((n-src n-drain)
     (==> (n-fet) vert-n-fet (n-src n-drain)))
    (at n-drain
(start-growing-point drop-contact N-WIDTH)
(start-growing-point metal->vss))
    (at n-src
```

```
(start-growing-point drop-contact N-WIDTH)
(connect (start-growing-point up-join-metal
      (+ (* 2 INV-HEIGHT) POLY-LEN)
      2)
 (->output common-out)))))

(define-network (inverter (poly-in) (poly-out))
  (let-locs
   ((p-fet n-fet)
    (at poly-in
(--> (start-growing-point horiz-poly POLY-LEN)
     (secrete+ (* 2 POLY-LEN) dir-pheromone)
     (--> (start-growing-point up-poly
       (+ INV-HEIGHT N-WIDTH))
  (--> (start-growing-point horiz-poly
    (+ POLY-LEN (* 2 N-WIDTH)))
       (->output p-fet)))
     (--> (start-growing-point down-poly
       (+ INV-HEIGHT N-WIDTH))
  (--> (start-growing-point horiz-poly
    (+ POLY-LEN (* 2 N-WIDTH)))
       (->output n-fet))))))
   (==> (p-fet n-fet) inverter-fets (poly-out))))

(define-network (inverter+rails (vdd-in vss-in poly-in)
(vdd-out vss-out poly-out))
  (let-locs ((vdd-tmp vss-tmp)
     (==> (vdd-in vss-in) rails (vdd-tmp vss-tmp)))
    (let-locs ((ser-poly-in)
       (==> (vdd-tmp poly-in) input-serializer (ser-poly-in)))
      (at vdd-tmp (->output vdd-out))
      (at vss-tmp (->output vss-out))
      (==> (ser-poly-in) inverter (poly-out)))))

(with-initial-locs (vdd-1 vdd-2 vss-1 vss-2 input)
  (==> (vdd-1 vss-1) init-rails ())
  (at input (secrete+ (* 3 POLY-LEN) dir-pheromone))
  (==> (vdd-2 vss-2 input) inverter+rails (vdd-ignore vss-ignore out-ignore)))
```

**net-nand.gpl**

```
;;; -*- Scheme -*-

(include "net-inverter2")

(constant METAL-LEN (* 2 POLY-LEN))
(constant INPUT-SEP (+ POLY-LEN (* 2 N-WIDTH) 1))
(constant POLY-HEIGHT-A (* 2 (+ FET-HEIGHT N-WIDTH)))
(constant POLY-HEIGHT-B (+ POLY-HEIGHT-A
   (+ FET-HEIGHT (* 2 N-WIDTH))
   INPUT-SEP))

(define-network (input-b-serializer (src dest-in) (dest-out))
  (at dest-in (start-growing-point short-range-beacon INPUT-SEP))
  (at src (--> (start-growing-point invisible-seg)
       (->output dest-out))))

(define-network (n-diffusion-segment (source dest-in) (dest-out))
  (at dest-in (start-growing-point short-range-beacon POLY-LEN))
  (at source (--> (start-growing-point n-diff-seg N-WIDTH)
  (->output dest-out))))

(define-network (nand-p-fet (p-fet) ())
  (let-locs ((src drain) (==> (p-fet) vert-p-fet (src drain)))
    (at src
(start-growing-point drop-contact N-WIDTH)
(start-growing-point metal->vdd))
    (at drain
(start-growing-point drop-contact N-WIDTH)
(start-growing-point down-join-metal))))

(define-network (nand-p-fet-B (p-fet) ())
  (let-locs ((src drain) (==> (p-fet) vert-p-fet-B (src drain)))
    (at src
(start-growing-point drop-contact N-WIDTH)
(start-growing-point metal->vdd))
    (at drain
(start-growing-point drop-contact N-WIDTH)
(start-growing-point down-join-metal))))

(define-network (nand-n-fets (n-fet-a n-fet-b) (output))
  (let-locs ((src-a drain-a)
     (==> (n-fet-a) vert-n-fet (src-a drain-a))
     (src-b drain-b)
     (==> (n-fet-b) vert-n-fet-B (src-b drain-b)))
    (at drain-b
```

```
(start-growing-point drop-contact N-WIDTH)
(start-growing-point metal->vss))
    (==> (drain-a src-b) n-diffusion-segment (ignore))
    (at src-a
(start-growing-point drop-contact N-WIDTH)
(--> (start-growing-point up-join-metal POLY-HEIGHT-B (* 2 POLY-LEN))
     (->output output)))))

(define-network (nand (a b) (result))
  (let-locs
   ((n-fet-a p-fet-a)
    (at a (--> (start-growing-point horiz-metal METAL-LEN)
       (start-growing-point drop-contact N-WIDTH)
       (--> (start-growing-point up-poly POLY-HEIGHT-A)
    (--> (start-growing-point horiz-poly (+ METAL-LEN N-WIDTH))
 (->output p-fet-a)))
       (--> (start-growing-point horiz-poly POLY-LEN)
    (->output n-fet-a))))
    (n-fet-b p-fet-b)
    (at b (--> (start-growing-point horiz-poly-B (- METAL-LEN (* 4 N-WIDTH)))
       (--> (start-growing-point horiz-poly-B
 (+ POLY-LEN (- (* 4 N-WIDTH) 1)))
    (->output n-fet-b))
       (--> (start-growing-point up-poly-B POLY-HEIGHT-B)
    (--> (start-growing-point horiz-poly-B METAL-LEN)
 (->output p-fet-b))))))
   (==> (p-fet-a) nand-p-fet ())
   (==> (p-fet-b) nand-p-fet-B ())
   (==> (n-fet-a n-fet-b) nand-n-fets (result))))

(define-network (nand+rails (vdd-in vss-in a b) (vdd-out vss-out result))
  (let-locs ((vdd-tmp vss-tmp)
     (==> (vdd-in vss-in) rails (vdd-tmp vss-tmp)))
    (let-locs ((ser-a)
       (==> (vss-tmp a) input-serializer (ser-a)))
      (let-locs ((ser-b)
 (==> (ser-a b) input-b-serializer (ser-b)))
(==> (vdd-tmp vss-tmp) rails (vdd-out vss-out))
(==> (ser-a ser-b) nand (result))))))

(with-locations (vdd-1 vdd-2 vss-1 vss-2 a b)
  (==> (vdd-1 vss-1) init-rails ())
  (at a (secrete+ (+ METAL-LEN POLY-HEIGHT-B) dir-pheromone))
  (==> (vdd-2 vss-2 a b) nand+rails (vdd-ignore vss-ignore out-ignore)))
```

**net-and2.gpl**

```
;;; -*- Scheme -*-

(include "net-nand")

(define-network (via+rails (vdd-in vss-in input) (vdd-out vss-out output))
  (==> (input) via (output))
  (at vdd-in (->output vdd-out))
  (at vss-in (->output vss-out)))

(define-network (and+rails (vdd-in vss-in a b) (vdd-out vss-out output))
  (==> (vdd-in vss-in a b) nand+rails via+rails inverter+rails
       (vdd-out vss-out output)))

(with-locations (vdd-1 vdd-2 vss-1 vss-2 a b)
  (==> (vdd-1 vss-1) init-rails ())
  (at a (secrete+ (+ POLY-LEN POLY-HEIGHT-B) dir-pheromone))
  (==> (vdd-2 vss-2 a b) and+rails (vdd-ignored vss-ignored ignored)))
```

## A.3 Looping Constructs

**loop-ray.gpl**

```
;;; -*- Scheme -*-

(constant RAY-LEN 10)

(define-gp (loop count length)
  (actions
   (when ((= count 0) (terminate))
 (default
   (start-gp ray length)
   (start-gp loop (- count 1) length)))))

(define-gp (ray length)
  (material ray-material)
  (size 1)
  (tropism (ortho- ray-p))
  (actions
   (secrete+ (/ length 2) ray-p)
   (when ((= length 0) (terminate))
 (default (propagate (- length 1))))))

(color
 ((ray-material) "blue"))

(with-initial-locs
 (root)
 (at root
     (start-gp loop 5 (- RAY-LEN 1))))
```

## A.4 Biological Inspirations

**starfish.gpl**

```
;;; -*- Scheme -*-

(constant ARM-LEN 10)

(define-gp (loop count length)
  (actions
   (when ((= count 0) (terminate))
 (default
   (start-gp arm length)
   (start-gp loop (- count 1) length)))))

(define-gp (arm length)
  (material arm-material)
  (size 1)
  (tropism (and (ortho- arm-p) (ortho- source-ph)))
  (actions
   (secrete+ (+ (- ARM-LEN length) 2) arm-p)
   (when ((= length 0) (terminate))
 (default (propagate (- length 1))))))

(color
 ((arm-material) "blue"))

(with-initial-locs
 (root)
 (at root
     (secrete ARM-LEN source-ph)
     (start-gp loop 5 (- ARM-LEN 1))))
```

**dtree2.gpl**

```
;;; -*- Scheme -*-

(define-growing-point (tree init-life life)
  (material plant)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< init-life 3)
  (terminate))
 ((< life 1)
  (propagate (/ init-life 3) (/ init-life 3))
  (propagate (/ init-life 3) (/ init-life 3))
  (propagate (/ init-life 3) (/ init-life 3)))
 (default
   (propagate init-life (- life 1))))))

(color
 ((plant) "blue"))

(with-locations
 (base-pt source)
 (at base-pt
     (secrete+ 3 self-pheromone))
 (at source
     (start-growing-point tree 9 9)))
```

**dtree-up.gpl**

```
;;; -*- Scheme -*-

(define-growing-point (tree init-life life)
  (material plant)
  (tropism (ortho- self-pheromone))
  (size 0)
  (for-each-step
   (secrete+ 3 self-pheromone)
   (when ((< init-life 3)
  (terminate))
 ((= life 1)
  (propagate init-life 0)
  (secrete+ 4 self-pheromone))
 ((< life 1)
  (propagate (/ init-life 3) (/ init-life 3))
  (propagate (/ init-life 3) (/ init-life 3))
  (propagate (/ init-life 3) (/ init-life 3)))
 (default
   (propagate init-life (- life 1))))))

(color
 ((plant) "blue"))

(with-locations
 (base-pt source)
 (at base-pt
     (secrete+ 3 self-pheromone))
 (at source
     (start-growing-point tree 9 9)))
```

**hands2.gpl**

```
;;; -*- Scheme -*-

;;;; This program is intended to demonstrate how mirror symmetry can be
;;;; obtained.  A hand was chosen as the pattern to be reflected.

;;; The constants used (all capitalised) are declared at the end of the file,
;;; just before the with-locations clause.

;;; This is the mirror line.
;;; This version uses a line segment instead of a ray for the mirror

(define-gp (mirror-line-start length)
  (material mirror-m)
  (size 1)
  (tropism (and (ortho+ mirror-self-p) (ortho- dir-p)))
  (actions
   (secrete WIDTH mirror-p)
   (when ((sensing? mirror-stop-m) (terminate))
 ((< length 1) (terminate))
 (default  (propagate (- length 1))))))

(define-gp (mirror-line)
  (material mirror-m)
  (size 1)
  (tropism (and (ortho+ mirror-self-p) (ortho- dir-p)))
  (actions
   (secrete WIDTH mirror-p)
   (when ((sensing? mirror-stop-m) (terminate))
 (default  (propagate)))))

(define-gp (mirror-stop)
  (material mirror-stop-m)
  (size 1)
  (actions
   (secrete (+ MIRROR-LEN-1 ARM-LAG-LEN MIRROR-LEN-2) mirror-self-p)
   (terminate)))

;;;  The arm connects to the hand

(define-gp (arm+hand)
  (actions
   (connect (start-gp arm ARM-LEN)
    (start-gp hand FINGER-LEN))))

(define-gp (arm length)
```

```
  (material arm-m)
  (tropism (and (ortho- arm-self-p)
(ortho- dir-p)
(plagio- mirror-p)))
  (size 3)
  (actions
   (secrete+ 4 arm-self-p)
   (when ((> length 0) (propagate (- length 1)))
 (default (terminate)))))

;;; the hand is comprised of the palm, the thumb and 4 fingers

(define-gp (hand finger-length)
  (actions
   (secrete+ 4 finger-p)
   (connect (start-gp palm-starter 1)
    (start-gp palm PALM-SIZE finger-length)
    (start-gp fingers 3 finger-length)
    (start-gp thumb finger-length)
    (start-gp finger-rest finger-length))))

;;; the palm-starter establishes the middle of the palm

(define-gp (palm-starter length)
  (tropism (and (ortho- dir-p)
(ortho- arm-self-p)
(plagio- mirror-p)))
  (actions
   (secrete+ 4 arm-self-p)
   (when ((> length 0) (propagate (- length 1)))
 (default (terminate)))))

;;; the palm's extent defines where fingers stop spreading apart

(define-gp (palm sz finger-length)
  (material palm-m)
  (size sz)
  (actions
   (secrete (+ sz finger-length) hand-p)
   (terminate)))

;;; three fingers are generated in the same way from the same location

(define-gp (fingers count length)
  (actions
   (when ((= count 0) (terminate))
 (default
   (start-gp finger length)
```

```
   (start-gp fingers (- count 1) length)))))

;;; a finger is generated in two stages

(define-gp (finger length)
  (actions
   (connect (start-gp finger-start)
    (start-gp finger-rest length))))

;;; the first stage starts the finger in the correct direction and
;;; encourages it to separate from other fingers in the area.

(define-gp (finger-start)
  (material hand-m)
  (size PALM-SIZE)
  (tropism (and  (ortho- arm-self-p)
 (plagio- mirror-p) (ortho- finger-p)))
  (actions
   (secrete+ 2 finger-p)
   ;;(secrete+ 2 arm-self-p)
   (when ((sensing? palm-m)
  (propagate))
 (default
   (terminate)))))

;;; the second stage extends the finger beyond the palm, maintaining the
;;; direction established by the first stage.

(define-gp (finger-rest length)
  (material finger-m)
  (size 1)
  (tropism (and (ortho- finger-p) (ortho- hand-p)
(or (ortho- mirror-p) (plagio- mirror-p))))
  (actions
   (secrete+ 2 finger-p)
   (when ((> length 0) (propagate (- length 1)))
 (default (terminate)))))

;;; The thumb grows differently wrt the mirror from the other fingers.

(define-gp (thumb length)
  (material thumb-m)
  (size 1)
  (tropism (and (ortho- arm-self-p)
(dia mirror-p)
(ortho- hand-p)
(ortho- finger-p)
))
```

```
  (actions
   (secrete+ 2 arm-self-p)
   (secrete+ 2 finger-p)
   (when ((> length 0) (propagate (- length 1)))
 (default (terminate)))))

;;; these are serializers to make sure the arm does not start before the
;;; mirror pheromones are in place.

(define-gp (arm-starter-attractor range)
  (material starter-m)
  (actions
   (secrete range arm-starter-p)
   (terminate)))

(define-gp (arm-starter)
  (tropism (ortho+ arm-starter-p))
  (actions
   (when ((sensing? starter-m) (terminate))
 (default (propagate)))))

;;; changing these colours will emphasize how the different regions
;;; were generated by different growing points.

(color
 ((mirror-stop-m) "red")
 ((mirror-m) "yellow")
 ((thumb-m) "black")
 ((finger-m) "black")
 ((hand-m) "black")
 ((palm-m) "black")
 ((arm-m) "blue")
 )

(constant WIDTH 20)
(constant MIRROR-LEN-1 5)
(constant MIRROR-LEN-2 20)
(constant ARM-LAG-LEN 5)
(constant ARM-LEN 10)
(constant FINGER-LEN 7)
(constant PALM-SIZE 3)

(with-initial-locs
 (mirror-start mirror-end)
 (at mirror-start
     (secrete (+ MIRROR-LEN-1 MIRROR-LEN-2) dir-p)
     (connect (start-gp mirror-line-start MIRROR-LEN-1)
      (start-gp arm-starter-attractor (+ ARM-LAG-LEN 1))
```

```
      (connect (start-gp mirror-line-start ARM-LAG-LEN)
       (connect (start-gp arm-starter ARM-LAG-LEN)
(start-gp arm+hand ARM-LEN FINGER-LEN)
(start-gp arm+hand ARM-LEN FINGER-LEN)
)
       (start-gp mirror-line))))

 (at mirror-end
     (start-gp mirror-stop)))

#|
To run, use draw-hands2 below.  Make sure to change d to be dependent on
MIRROR-LEN-1 + MIRROR-LEN-2

(define (draw-hands2 x y)
  (let ((d (* 25 ds))) ; 25 = total mirror length
    (illustrate "hands2"
(list (make-pt x y))
(list (make-pt x (+ y d))))))

|#
```

## A.5 Cautions

**sector.gpl**

```
;;; -*- Scheme -*-

(constant radius 40)
(constant arc-len 80) ; good case
;(constant arc-len 45) ; bad case

(define-growing-point (B-point)
  (material B-material)
  (size 0)
  (actions
   (secrete+ radius B-pheromone)
   (terminate)))

(define-growing-point (AB-segment)
  (material A-material)
  (size 0)
  (tropism (ortho+ B-pheromone))
  (actions
   (when ((sensing? B-material)
  (terminate))
 (default
   (propagate)))))

(define-growing-point (centre)
  (material C-material)
  (size 0)
  (actions
   (secrete+ radius C-pheromone)
   (terminate)))

(define-growing-point (arc@C length)
  (material arc-material)
  (size 0)
  (tropism (and (ortho- arc-pheromone) (dia C-pheromone)))
  (actions
   (secrete+ 3 arc-pheromone)
   (when ((< length 1) (terminate))
 (default (propagate (- length 1))))))

(define-network (segment (a b) (c))
  (at a (connect (start-growing-point AB-segment)
 (->output c)))
  (at b (start-growing-point B-point)))
```

```
(define-network (arc (a c) (d))
  (at a (connect (start-growing-point arc@C arc-len)
 (->output d)))
  (at c (start-growing-point centre)))

(define-network (sector (c a) (d))
  (let-locs ((b) (==> (c a) segment (b)))
    (let-locs ((e) (==> (b c) arc (e)))
      (==> (e c) segment (d)))))

(color
 ((B-material) "red")
 ((A-material) "blue")
 ((arc-material) "yellow"))

(with-locations (c a)
  (let-locs ((ignored)
     (==> (c a) sector (ignored)))
    (at ignored (halt))))
```

# Appendix B

# ECOLI Implementation of Secretion FSM

## B.1 The FSM Definition

```
(define secretion-fsm
  (let ((dict-1
        (make-dictionary
         (((quote secrete) id-record extent count)
          (cond ((< count extent) (add-count id-record count) 1)
                ((= count extent)
                 (add-count id-record count)
                 (record.set-min!
                  id-record
                  (+ 1 (apply min (cdr (record.vals id-record)))))
                 3)
                (else 0)))
         (((quote update) id-record extent old-count new-count)
          old-count
          (cond ((< new-count extent) (add-count id-record new-count) 1)
                ((= new-count extent)
                 (add-count id-record new-count)
                 (record.set-min!
                  id-record
                  (+ 1 (apply min (cdr (record.vals id-record)))))
                 3)
                (else 0)))
         (((quote restore) . args) 0)
         (((quote timeout) . args) 0)))

       (dict-2
        (make-dictionary
         (((quote secrete) id-record extent count)
          extent
          (add-count id-record count)
          1)
         (((quote update) id-record extent old-count new-count)
          (replace-count id-record old-count new-count)
          1)
         (((quote restore) . args) 1)
         (((quote timeout) id-rec proc)
          (let ((new-min (+ 1 (apply min (cdr (record.vals id-rec))))))
```

```
     (record.set-min! id-rec new-min)
     (proc (quote secrete) new-min)
     2))))

(dict-3
 (make-dictionary
  (((quote secrete) id-rec extent count)
   (add-count id-rec count)
   (if (< (+ count 1) (record.min id-rec))
       4
       2))
  (((quote update) id-rec extent old-count new-count)
   (replace-count id-rec old-count new-count)
   (if (< (+ new-count 1) (record.min id-rec))
       4
       2))
  (((quote restore) id-rec count)
   (if (>= count (record.min id-rec))
       3
       2))
  (((quote timeout) id-rec proc) proc 3)))

(dict-4
 (make-dictionary
  (((quote secrete) id-rec extent count)
   (add-count id-rec count)
   (if (< (+ count 1) (record.min id-rec))
       4
       2))
  (((quote update) id-rec extent old-count new-count)
   (replace-count id-rec old-count new-count)
   (if (< (+ new-count 1) (record.min id-rec))
       4
       2))
  (((quote restore) id-rec count)
   (if (>= count (record.min id-rec))
       3
       3))
  (((quote timeout) id-rec proc)
   (let ((new-min (+ 1 (apply min (cdr (record.vals id-rec))))))
     (proc (quote restore) new-min)
     ()))))

(dict-5
 (make-dictionary
  (((quote secrete) id-record extent count)
   extent
   (add-count id-record count)
   4)
  (((quote update) id-record extent old-count new-count)
   (replace-count id-record old-count new-count)
   4)
  (((quote restore) . args) 4)
  (((quote timeout) id-rec proc)
   (let ((new-min (+ 1 (apply min (cdr (record.vals id-rec))))))
     (proc (quote update) (record.min id-rec) new-min)
     (record.set-min! id-rec new-min)
     2))))
```

```
    (dict-6
     (make-dictionary
      (((quote secrete) id-record extent count)
       extent
       (add-count id-record count)
       5)
      (((quote update) id-record extent old-count new-count)
       (replace-count id-record old-count new-count)
       5)
      (((quote restore) id-rec count)
       (if (>= count (record.min id-rec))
           6
           5))
      (((quote timeout) . args) 5)))

    (dict-7
     (make-dictionary
      (((quote secrete) id-record extent count)
       extent
       (add-count id-record count)
       5)
      (((quote update) id-record extent old-count new-count)
       (replace-count id-record old-count new-count)
       5)
      (((quote restore) id-rec count)
       (if (>= count (record.min id-rec))
           6
           6))
      (((quote timeout) id-rec proc) proc ())))))

(vector
 (make-state (quote state:default) dict-1 (lambda (action) action ()))
 (make-state (quote state:t-secrete)
             dict-2
             (lambda (action) (delayed-perform secretion-delay action)))
 (make-state
  (quote state:wait)
  dict-3
  (let ((limit (* 12 secretion-delay)))
    (lambda (action)
      (delayed-perform limit action))))
 (make-state
  (quote state:t-restore)
  dict-4
  (let ((limit (* 4 secretion-delay)))
    (lambda (action)
      (delayed-perform limit action))))
 (make-state (quote state:t-update)
             dict-5
             (lambda (action) (delayed-perform secretion-delay action)))
 (make-state (quote state:src-default) dict-6 (lambda (action) action ()))
 (make-state
  (quote state:src-t-restore)
  dict-7
  (let ((limit (* 4 secretion-delay)))
    (lambda (action)
      (delayed-perform limit action))))))
```

## B.2 Implementation of secrete

```
(define max-secretion-process
  (let ((id-records secretion-id-records) (table pheromone-table))

    (define (2d-get table key1 key2)
      (let ((entry (get table key1)))
        (and entry
             (get entry key2))))

    (define (2d-put table key1 key2 val)
      (let ((entry (get table key1)))
        (if entry
            (put entry key2 val)
            (put
             table
             key1
             (entries->table (list (make-entry key2 val)) (quote ids) ())))))

    (define (commit name key id init-val)
      (lambda ()
        (let ((name-id-records (get id-records name)))
          (let ((current-pair (get table key))
                (id-record (get name-id-records id)))
            (if (not current-pair)
                (error "No lock was set during process" name key))
            (remove! name-id-records id)
            (if (table/empty? name-id-records)
                (remove-register table key name))
            (if (eqv? id (get-var (quote id)))
                (send-to-self (quote done))
                (set-pheromone-val!
                 current-pair
                 (max-combiner (cadr current-pair)
                               init-val
                               (cdr (record.vals id-record)))))))))

    (make-process
     (quote max-secretion-process)
     (make-dictionary
      (define (timeout-proc name key id init-val extent)
        (lambda (msg-tag . rest-msg)
          (apply send
                 (quote max-secretion-process)
                 msg-tag
                 name
                 key
                 id
                 init-val
                 extent
                 rest-msg)))

      (((quote secrete) name key id init-val extent count)
       (let ((id-rec
              (or (2d-get id-records name id)
                  (let ((new-rec (make-default-record)))
                    (2d-put id-records name id new-rec)
                    new-rec))))
         (register-update table key name)
```

```
   (id-transition
    secretion-fsm
    id-rec
    (quasiquote
     (secrete (unquote id-rec) (unquote extent) (unquote count)))
    (commit name key id init-val)
    (timeout-proc name key id init-val extent))))

(((quote update) name key id init-val extent old-count new-count)
 (let ((id-rec
        (or (2d-get id-records name id)
            (let ((new-rec (make-default-record)))
              (2d-put id-records name id new-rec)
              new-rec))))
   (register-update table key name)
   (id-transition
    secretion-fsm
    id-rec
    (quasiquote
     (update (unquote id-rec)
             (unquote extent)
             (unquote old-count)
             (unquote new-count)))
    (commit name key id init-val)
    (timeout-proc name key id init-val extent))))

(((quote restore) name key id init-val extent count)
 (let ((id-rec (2d-get id-records name id)))
   (if id-rec
       (begin
        (id-transition
         secretion-fsm
         id-rec
         (quasiquote (restore (unquote id-rec) (unquote count)))
         (commit name key id init-val)
         (timeout-proc name key id init-val extent))))))

(((quote activate) name key extent init-val . args)
 args
 (if (> extent 0)
     (let ((myid (get-var (quote id))))
       (action.start
        (make-action
         (if (2d-get id-records name myid)
             (repeat)
             (begin
              (register-update table key name)
              (let ((current-pair (get table key)))
                (set-pheromone-val!
                 current-pair
                 (max-combiner (get-pheromone-val current-pair)
                               init-val
                               (quote ())))
                (2d-put id-records name myid (make-source-record))
                (send (quote max-secretion-process)
                      (quote secrete)
                      name
                      key
                      myid
```

```
                  init-val
                  extent
                  0)))))))
(send-to-self (quote done))))))))
```

## B.3 FSMs in ECOLI

This section presents the support code for building FSMs as well as the program that was used to generate the FSM for the `secrete` and `secrete+` commands (they both used the same FSM).

### B.3.1 Support code

```
(define (make-fsm states transition-table)
  ;; STATES is a list of states, thereby specifying each state's type.
  ;; TRANSITION-TABLE has message names for row-titles (explicit),
  ;;  state numbers for column headings (implicit in ordering),
  ;;  and a list of a function and possible destination states as entries.
  ;;  The function takes destination states as arguments, and returns a
  ;;   procedure that takes msg args and returns one of the states.
  ;; #f as an entry means that the message is ignored in the indicated state

  #| ;
  eg:
  `((SECRETE (,f0s 1) (,f1s 1) (,f2s 2 3) (,f3s 3))
    (UPDATE  (,f0u 1) (,f1u 1) (,f2u 2 3) (,f3u 3))
    (TIMEOUT #f       (,f1t 2) (,f2t 0)   (,f3t 2)))
  |#

  (let ((xdicts (g-map state/xdict states))
(states-vec (list->vector states)))
    (let row-loop ((rows transition-table))
      (if (null? rows)
  states-vec
  (let* ((row (car rows))
 (msg-name (car row)))
    (let col-loop ((entries (cdr row)) (dicts xdicts) (state-num 0))
      (if (null? entries)
  (row-loop (cdr rows))
  (let ((transition (car entries)) (dict (car dicts)))
    (if transition
(dict.bind dict msg-name
   (apply (car transition)
  (cdr transition)))
(dict.bind dict msg-name
   (make-handler args state-num)))
    (col-loop (cdr entries) (cdr dicts)
      (+ state-num 1))))))))))

(define (make-state name xdict entry-proc)
  ;; XDICT (transition-dictionary) is a dictionary in which
  ;; every handler returns a state
  (vector name xdict entry-proc))

(define-integrable (state/name state) (vector-ref state 0))
(define-integrable (state/xdict state) (vector-ref state 1))
(define-integrable (state/entry-proc state) (vector-ref state 2))

(define (state.enter fsm state . inputs)
  (apply (state/entry-proc (vector-ref fsm state)) inputs))

#|
(define (state.exit fsm state)
```

```
  ((state/exit-thunk (vector-ref fsm state))))
|#

(define (state.transition fsm state input)
  ;; current state is maintained outside of fsm, to allow code sharing
  (eval-msg input (state/xdict (vector-ref fsm state))))

(define (make-default-state name)
  (make-state name
      (make-empty-dictionary)
      (lambda (action) action #f)))

(define (make-shared-timeout-state name limit)
  (make-state name
      (make-empty-dictionary)
      (lambda (action)
(delayed-perform
 limit
 action))))

(define (add-count id-record count)
  (let ((vals (record.vals id-record)))
    (set-cdr! vals (cons count (cdr vals)))))

(define (replace-count id-record old-count new-count)
  (let ((vals (record.vals id-record)))
    (let loop ((prev vals) (current (cdr vals)))
      (cond ((null? current)
     (add-count id-record new-count))
    ((= (car current) old-count)
     (set-cdr! prev (cons new-count (cdr current))))
    (else
     (loop current (cdr current)))))))
```

### B.3.2 Generator for Secretion FSM

```
(define secretion-fsm
  ;; formerly (make-secretion-fsm process-name delay)
  (let ((process-name 'secretion-process)
        (delay secretion-delay))
    (let ((restore-delay (* 4 delay))
          ;; max idle delay. Large to give RESTORE a chance to work normally
          (wait-restore-delay (* 8 delay))
          (state:default 0)
          (state:t-secrete 1)
          (state:wait 2)
          (state:t-restore 3)
          (state:t-update 4)
          (state:src-default 5)
          (state:src-t-restore 6))

      (define (compute-min id-record)
        (+ 1 (apply min (cdr (record.vals id-record)))))

      (define (correct-min state1 state2)
        ;; if new min for id go to state2 else return to state1.
        (lambda (id-record new-count)
          (if (< (+ new-count 1) (record.min id-record))
              state2
              state1)))

      (define (secrete:init state1 state2 state3)
        (lambda (id-record extent count)
          (cond ((< count extent)
                 (add-count id-record count)
                 state1)
                ((= count extent)
                 (add-count id-record count)
                 (record.set-min! id-record (compute-min id-record))
                 state2)
                (else
                 state3))))

      (define (secrete:simple state1)
        (lambda (id-record extent count)
          init-val extent              ;ignored
          (add-count id-record count)
          state1))

      (define (secrete:correction? state1 state2)
        (lambda (id-rec extent count)
          (add-count id-rec count)
          ((correct-min state1 state2)
           id-rec count)))

      (define (update:init state1 state2 state3)
        (lambda (id-record extent old-count new-count)
          old-count                    ;ignored
          (cond ((< new-count extent)
                 (add-count id-record new-count)
                 state1)
                ((= new-count extent)
                 (add-count id-record new-count)
```

```
          (record.set-min! id-record (compute-min id-record))
          state2)
         (else
          state3))))

(define (update:simple state1)
  (lambda (id-record extent old-count new-count)
    (replace-count id-record old-count new-count)
    state1))

(define (update:correction? state1 state2)
  (lambda (id-rec extent old-count new-count)
    (replace-count id-rec old-count new-count)
    ((correct-min state1 state2)
     id-rec new-count)))

(define (timeout:secrete state1)
  (lambda (id-rec name key id init-val extent)
    (let ((new-min (compute-min id-rec)))
      (record.set-min! id-rec new-min)
      (send process-name 'SECRETE name key id init-val extent new-min)
      state1)))

(define (timeout:update state1)
  (lambda (id-rec name key id init-val extent)
    (let ((new-min (compute-min id-rec)))
      (send process-name 'UPDATE name key id init-val extent
            (record.min id-rec) new-min)
      (record.set-min! id-rec new-min)
      state1)))

(define (timeout:wait state1)
  (lambda (id-rec name key id init-val extent)
    id-rec name key id init-val extent ;ignored
    state1))

(define (timeout:restore state1)
  (lambda (id-rec name key id init-val extent)
    extent                         ;ignored
    (let ((new-min (compute-min id-rec)))
      (send process-name 'RESTORE name key id init-val extent new-min)
      state1)))

(define (timeout:done state1)
  (lambda (id-rec name key id init-val extent)
    id init-val extent             ;ignored
    ;;(send-to-self 'DONE)
    ;; exit fsm. cleanup of state and DONE msg happens externally
    state1))

(define (restore:relevant? state1 state2)
  (lambda (id-rec count)
    (if (>= count (record.min id-rec))
        state1
        state2)))

(make-fsm
 (list (make-default-state 'state:default)
       (make-shared-timeout-state 'state:t-secrete delay)
```

```
      (make-shared-timeout-state 'state:wait wait-restore-delay)
      (make-shared-timeout-state 'state:t-restore restore-delay)
      (make-shared-timeout-state 'state:t-update delay)
      (make-default-state 'state:src-default)
      (make-shared-timeout-state 'state:src-t-restore restore-delay))
`(
  (SECRETE (,secrete:init ,state:t-secrete ,state:t-restore
                         ,state:default)
           (,secrete:simple ,state:t-secrete)
           (,secrete:correction? ,state:wait ,state:t-update)
           (,secrete:correction? ,state:wait ,state:t-update)
           (,secrete:simple ,state:t-update)
           (,secrete:simple ,state:src-default)
           (,secrete:simple ,state:src-default))
  (UPDATE  (,update:init ,state:t-secrete ,state:t-restore
                        ,state:default)
           (,update:simple ,state:t-secrete)
           (,update:correction? ,state:wait ,state:t-update)
           (,update:correction? ,state:wait ,state:t-update)
           (,update:simple ,state:t-update)
           (,update:simple ,state:src-default)
           (,update:simple ,state:src-default))
  (RESTORE #f
           #f
           (,restore:relevant? ,state:t-restore ,state:wait)
           (,restore:relevant? ,state:t-restore ,state:t-restore)
           #f
           (,restore:relevant? ,state:src-t-restore ,state:src-default)
           (,restore:relevant? ,state:src-t-restore
                               ,state:src-t-restore))
  (TIMEOUT #f
           (,timeout:secrete ,state:wait)
           (,timeout:wait ,state:t-restore)
           (,timeout:restore #f)
           (,timeout:update ,state:wait)
           #f
           (,timeout:done #f))))))
```

## B.4 Support code for pheromone maintenance

```
;;; -*- Scheme -*-

(declare (usual-integrations))
(declare (integrate-external "ecoli2" "grow-language2"))

#|
;; secrete spec

(secrete <extent> <pheromone>)

<extent> = Natural Number

<pheromone> = Identifier
|#

(define (lookup-pheromone table pheromone-key free-proc block-proc)
  (let ((result (get table pheromone-key)))
    (cond ((not result) (free-proc #f))
  ((null? (car result))
   (free-proc (cadr result)))
  (else
   (block-proc (car result))))))

(define-integrable (make-locks-val-pair locks val)
  (list locks val))

(define-integrable (set-pheromone-val! locks-val-pair val)
  (set-car! (cdr locks-val-pair) val))

(define-integrable (get-pheromone-val locks-val-pair)
  (cadr locks-val-pair))

(define-integrable (get-pheromone-locks locks-val-pair)
  (car locks-val-pair))

(define-integrable (set-pheromone-locks! locks-val-pair new-locks)
  (set-car! locks-val-pair new-locks))

(define (register-update table tag lock-name)
  (let ((result (get table tag)))
    (if (not result)
(put table tag (make-locks-val-pair (list lock-name) 0.))
(let ((locks (get-pheromone-locks result)))
  (if (not (memq lock-name locks))
      (set-pheromone-locks! result (cons lock-name locks)))))))

(define (remove-register table tag lock-name)
  (let ((result (get table tag)))
    (if (not result)
(begin
  (report `(warning: No lock on ,tag in ,table by ,lock-name))
  #f)
(set-car! result (delq lock-name (car result))))))

(define (compute-new-level nbr-values)
  (let ((n (vector-ref nbr-values 0)))
    (if (= n 0)
```

```
(begin
  (report 'potential-disaster: nbr-values)
  (- (vector-ref nbr-values 2)))
(- (/ (vector-ref nbr-values 1) n)
   (vector-ref nbr-values 2)))))

;; upon receipt of SECRETE:
;; open record for id
;; reset timeout for propagating
;; listen-to-send (max-count - 1)
;; listen-to-commit
;;  if any updates
;;     then cancel commit,
;;         if update > max-count
;;            then listen-to-send update
;;                 repeat listen-to-commit
;;            else replace update, but do not propagate
;;     else commit record, drop info for id

(define (make-record state timer min-val nbr-vals)
  (vector state timer min-val nbr-vals))

(define-integrable (record.state rec) (vector-ref rec 0))
(define-integrable (record.timer rec) (vector-ref rec 1))
(define-integrable (record.min rec) (vector-ref rec 2))
(define-integrable (record.vals rec) (vector-ref rec 3))

(define-integrable (record.set-state! rec state)
  (vector-set! rec 0 state))

(define-integrable (record.set-timer! rec timer)
  (vector-set! rec 1 timer))

(define-integrable (record.set-min! rec min)
  (vector-set! rec 2 min))

(define-integrable (record.set-vals! rec vals)
  (vector-set! rec 3 vals))

(define (id-transition fsm id-rec input commit-thunk proc)
  (let ((timer (record.timer id-rec)))
    (if timer
(begin
  (action.stop timer)
  (record.set-timer! id-rec #f)))

    ;;(report `(input: ,input))
    (record.set-state!
     id-rec
     (state.transition fsm (record.state id-rec) input))
    (if (record.state id-rec)
(record.set-timer!
 id-rec
 (state.enter fsm (record.state id-rec)
      (make-action
       (id-transition fsm id-rec (list 'TIMEOUT id-rec proc)
      commit-thunk proc))))
(commit-thunk))
    id-rec))
```

```
(define (make-default-record)
  ;; 0 is the default initial state
  (make-record 0 #f #f (list 'vals)))

(define (make-source-record)
  ;; 5 is the initial state for sources
  (make-record 5 #f 0 (list 'vals)))

(define (register-secretion name)
  (put secretion-id-records
       name (make-table 'id-records #f))
  name)
```

## B.5 Runtime Utilities

```
;;; -*- Scheme -*-

(declare (usual-integrations))

;;;; Procedures used at runtime

;;; Code for pheromone level combination

(define (arithmetic-mean lst)
  (if (null? lst)
      0.
      (let loop ((sum (car lst)) (n 1) (rest (cdr lst)))
(if (null? rest)
    (exact->inexact (/ sum n))
    (loop (+ sum (car rest)) (+ n 1) (cdr rest))))))

(define (geometric-mean lst)
  ;; pseudo geometric mean.  Correction of +1 to product before taking root.
  (if (null? lst)
      0.
      (let loop ((product (car lst)) (n 1) (rest (cdr lst)))
(cond ((null? rest)
       (exact->inexact (expt product (/ 1. n))))
      ((= (car rest) 0)
       (loop product n (cdr rest)))
      (else
       (loop (* product (car rest)) (+ n 1) (cdr rest)))))))

(define max-combiner
  (lambda (current-val init-val nbr-vals)
    (let ((dist (arithmetic-mean nbr-vals)))
      ;; 7 is a convenient upper bound on 2pi
      (let ((result
     (max current-val
  (if (= dist 0)
      init-val
      (/ init-val 7. hop-radius dist)))))
;;(report (e-time) 'max current-val init-val nbr-vals '--> result)
result))))

(define +-combiner
  (lambda (current-val init-val nbr-vals)
    (let ((dist (arithmetic-mean nbr-vals)))
      ;; 7 is a convenient upper bound on 2pi
      ;;(report (e-time) '+ current-val init-val nbr-vals)
      (+ current-val
 (if (= dist 0)
     init-val
     (/ init-val 7. hop-radius dist))))))

;;; Code for growing-point propagation

(define (make-sel-predicate name dict)
  (lambda (sender-id target-names)
    (let loop ((results '()) (reqd-pheromones target-names))
      (if (null? reqd-pheromones)
  (begin
```

```
    (dict.bind dict 'CONF
       (make-conf-handler sender-id name))
    (reverse results))
  (lookup-pheromone pheromone-table (car reqd-pheromones)
    (lambda (level)
      (if level
  (loop (cons level results)
(cdr reqd-pheromones))
  'wait-retry))
    (lambda (locks)
      'wait-retry))))))

(define (make-conf-handler src-id name)
  (lambda (dest-id sender-id last-result . args)
    (and (= sender-id src-id)
 (= dest-id (get-var 'id))
 (apply send-to-self name 'ACTIVATE last-result args))))

(define (make-widen name size material-list)
  (let ((count (- size 1)))
    (make-handler
     (id)
     (if (>= count 0)
 (begin
   (g-for-each (lambda (material)
 (send material (- size count)))
       material-list)
   (set! count (- count 1))
   (delayed-perform growth-delay
    (make-action
     (send-to-self name 'WIDEN id))))))))

(define (reset buffer) (set-cdr! buffer '()))

(define (convert-results result-list)
  (g-map (lambda (result-pair)
   (cons (car result-pair)
 (list->vector (cdr result-pair))))
 result-list))

(define (compute-my-result target-names)
  (list->vector
   (g-map (lambda (name)
    (lookup-pheromone pheromone-table name
      (lambda (level) level)
      (lambda (locks) #f)))
  target-names)))

(define (check-my-result my-result)
  (let loop ((i (- (vector-length my-result) 1)))
    (if (>= i 0)
(begin
  (if (vector-ref my-result i)
      (loop (- i 1))
      #f))
#t)))

(define (pick-exp-random lst)
  ;; returns an element from lst with exponential PMF with base .5
```

```
  (cond ((null? lst) #f)
((null? (cdr lst)) (car lst))
((> (random 1.) .5) (car lst))
(else (pick-exp-random (cdr lst)))))

;;; Code for depositing material

(define (make-new-material-handler material count)
  (def-response material
    (make-handler (new-count)
     (if (= new-count (+ count 1))
 (begin
   (make-new-material-handler material new-count)
   (delayed-send nbr-estimate material new-count))))))

(define (enable-materials material-lst)
  (g-for-each
   (lambda (material)
     (def-response material
       (make-handler (count)
     (material-type+ material)
     (make-new-material-handler material count))))
   material-lst))

(define (is-material? material)
  (let ((material-list (get-var 'material)))
    (and material-list
 (subset? material material-list))))

(define (material-type+ material)
  (let ((material-list (get-var 'material)))
    (cond ((not material-list)
   (set-var 'material (list material)))
  ((member material material-list)
   #f)
  (else
   (set-var 'material (cons material material-list))))
    (colour-material (get-var 'material))))

(define (colour-material material-list)
  (let ((colour-table (get-var 'colour-table)))
    (if colour-table
(let ((colour? (assm material-list colour-table)))
  (if colour?
      (color-me (cadr colour?)))))))

(define (assm el-list list-assocs)
  (cond ((null? list-assocs) #f)
((subset? (caar list-assocs) el-list)
 (car list-assocs))
(else (assm el-list (cdr list-assocs)))))

(define (subset? lst1 lst2)
  (cond ((null? lst1) #t)
((null? lst2) #f)
(else (and (memq (car lst1) lst2)
   (subset? (cdr lst1) lst2)))))

;;; Code for secretions
```

```
(define (make-fsm states transition-table)
  ;; STATES is a list of states, thereby specifying each state's type.
  ;; TRANSITION-TABLE has message names for row-titles (explicit),
  ;;  state numbers for column headings (implicit in ordering),
  ;;  and a list of a function and possible destination states as entries.
  ;;  The function takes destination states as arguments, and returns a
  ;;   procedure that takes msg args and returns one of the states.
  ;; #f as an entry means that the message is ignored in the indicated state

  #| ;
  eg:
  `((SECRETE (,f0s 1) (,f1s 1) (,f2s 2 3) (,f3s 3))
    (UPDATE  (,f0u 1) (,f1u 1) (,f2u 2 3) (,f3u 3))
    (TIMEOUT #f       (,f1t 2) (,f2t 0)   (,f3t 2)))
  |#

  (let ((xdicts (g-map state/xdict states))
(states-vec (list->vector states)))
    (let row-loop ((rows transition-table))
      (if (null? rows)
  states-vec
  (let* ((row (car rows))
 (msg-name (car row)))
    (let col-loop ((entries (cdr row)) (dicts xdicts) (state-num 0))
      (if (null? entries)
  (row-loop (cdr rows))
  (let ((transition (car entries)) (dict (car dicts)))
    (if transition
(dict.bind dict msg-name
   (apply (car transition)
  (cdr transition)))
(dict.bind dict msg-name
   (make-handler args state-num)))
    (col-loop (cdr entries) (cdr dicts)
      (+ state-num 1))))))))))

(define (make-state name xdict entry-proc)
  ;; XDICT (transition-dictionary) is a dictionary in which
  ;; every handler returns a state
  (vector name xdict entry-proc))

(define-integrable (state/name state) (vector-ref state 0))
(define-integrable (state/xdict state) (vector-ref state 1))
(define-integrable (state/entry-proc state) (vector-ref state 2))

(define (state.enter fsm state . inputs)
  (apply (state/entry-proc (vector-ref fsm state)) inputs))

#|
(define (state.exit fsm state)
  ((state/exit-thunk (vector-ref fsm state))))
|#

(define (state.transition fsm state input)
  ;; current state is maintained outside of fsm, to allow code sharing
  (eval-msg input (state/xdict (vector-ref fsm state))))

(define (make-default-state name)
```

```
  (make-state name
      (make-empty-dictionary)
      (lambda (action) action #f)))

(define (make-shared-timeout-state name limit)
  (make-state name
      (make-empty-dictionary)
      (lambda (action)
(delayed-perform
 limit
 action))))

(define (add-count id-record count)
  (let ((vals (record.vals id-record)))
    (set-cdr! vals (cons count (cdr vals)))))

(define (replace-count id-record old-count new-count)
  (let ((vals (record.vals id-record)))
    (let loop ((prev vals) (current (cdr vals)))
      (cond ((null? current)
     (add-count id-record new-count))
    ((= (car current) old-count)
     (set-cdr! prev (cons new-count (cdr current))))
    (else
     (loop current (cdr current)))))))

;;; Stable Insertion sort.  Result lists are small, so it's OK.
(define (g-sort lst pred)
  (define (insert! elt sorted-lst)
    (if (pair? sorted-lst)
(if (pred elt (car sorted-lst))
    (cons elt sorted-lst)
    (let loop ((prev sorted-lst) (rest (cdr sorted-lst)))
      (if (or (null? rest) (pred elt (car rest)))
  (begin
    (set-cdr! prev (cons elt rest))
    sorted-lst)
  (loop (cdr prev) (cdr rest)))))
(cons elt '())))

  (if (pair? lst)
      (let loop ((unsorted (cdr lst)) (sorted (list (car lst))))
(if (null? unsorted)
    sorted
    (loop (cdr unsorted) (insert! (car unsorted) sorted))))
      '()))

(define (union lst1 lst2)
  (cond ((null? lst1) lst2)
((member (car lst1) lst2)
 (union (cdr lst1) lst2))
(else
 (union (cdr lst1) (cons (car lst1) lst2)))))
```

# Appendix C

# Implementation of the GPL illustrator

## C.1 Interpreter code

This section provides the complete code listing for the GPL interpreter.

### C.1.1 Top Level commands

**top-level.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(define id-range 65536) ; 16 bits of gp-instance-id
(define with-loc-symbol '**GPL-entry-point**)
(define top-level-env (make-global-env))
(define quiescent-action-exists? #f)
(define colour-table '())
(define material-table (make-material-table pt:=))
(define backtracking-table (make-empty-table pt:=))
(define initial-points '())

(define (illustrate program-name . pt-sets)
  (let ((program (read-gpl-file program-name)))
    (let ((env (parse-gpl-program program)))
      (let ((entry-proc (lookup-var env with-loc-symbol)))
(if (not entry-proc)
    (error "GPL Program must contain a with-locations statement"))
(D:reset *D*) ; reset the Domain
(let ((point-sets (map (lambda (pt-set)
 (map (lambda (pt) (D:inject *D* pt))
      pt-set))
       pt-sets)))
  (set! initial-points point-sets)
  (apply entry-proc point-sets))))))

(define (parse-gpl-program program)
  ;;(set! *identifier-count* 0)
  (set! quiescent-action-exists? #f)
```

```
  (set! top-level-env (make-global-env))
  (set! colour-table '())
  (set! material-table (make-material-table pt:=))
  (set! backtracking-table (make-empty-table pt:=))
  (reset-agenda)

  (eval-top-level-seq program top-level-env))

(define (eval-top-level-seq top-level-exps env)
  (cond ((null? top-level-exps)
 env)
((null? (cdr top-level-exps))
 (eval-top-level (car top-level-exps) env))
(else
 (eval-top-level-seq (cdr top-level-exps)
     (eval-top-level (car top-level-exps)
     env)))))

(define (eval-top-level top-level-exp env)
  (if (not (pair? top-level-exp))
      (error "Badly formed top level statement" top-level-exp))
  (case (car top-level-exp)
    ((DEFINE-GROWING-POINT DEFINE-GP)
     (eval-define-growing-point (cdr top-level-exp) env))
    ((DEFINE-NETWORK)
     (eval-define-network (cdr top-level-exp) env))
    ((CONSTANT)
     (eval-constant (cdr top-level-exp) env))
    ((COLOR)
     (set! colour-table (cdr top-level-exp))
     env)
    ((SHOW-PHEROMONE) ; ignore for now
     (compile-show-pheromone (cadr top-level-exp)))
    ((WITH-LOCATIONS WITH-INITIAL-LOCS)
     (eval-with-locations (cdr top-level-exp) env))
    ((INCLUDE)
     (eval-include (cdr top-level-exp) env))
    (else
     (eval-top-level-seq top-level-exp env))))

(define (eval-constant const-exp env)
  (let ((var (car const-exp)))
    (if (symbol? var)
(add-binding env var (eval-exp (cadr const-exp) env #f))
(error "Bad Constant identifier" var))))

(define (eval-include files env)
  (if (null? files)
      env
      (eval-include (cdr files)
    (eval-top-level-seq (read-gpl-file (car files))
env))))

(define (eval-define-growing-point dgp-exps env)
  (let ((name+params (car dgp-exps)))
    (let ((name (car name+params))
  (params (cdr name+params))
  (material? (assq 'material dgp-exps))
  (tropism (assq 'tropism dgp-exps))
```

```
  (size? (assq 'size dgp-exps))
  (instructions (or (assq 'for-each-step dgp-exps)
    (assq 'actions dgp-exps)))
  (avoids-anything? (assq 'avoids dgp-exps))
  (targets (list 'pheromone-names)))
      (let ((material (and material? (cdr material?)))
    (size-code (if size?
   (let ((size-exp (cadr size?)))
     (lambda (point args)
       (eval-exp size-exp
 (extend-env env params args) point)))
   (lambda (point args) 0)))
    (tropism-code (and tropism
       (analyse-tropism (cadr tropism) targets)))
    (body-code (make-body
(if instructions
    (cdr instructions)
    '((TERMINATE)))
env
params))
    (poisons (and avoids-anything?
  (cdr avoids-anything?)))
    )
(add-binding env name
     (vector name
     (length params)
     material
     size-code
     body-code
     (list->vector (names+indexes->list targets))
     tropism-code
     (or poisons '())))))))

(define-integrable (gp.name gp) (vector-ref gp 0))
(define-integrable (gp.arity gp) (vector-ref gp 1))
(define-integrable (gp.material gp) (vector-ref gp 2))
(define-integrable (gp.size gp) (vector-ref gp 3))
(define-integrable (gp.body gp) (vector-ref gp 4))
(define-integrable (gp.targets gp) (vector-ref gp 5))
(define-integrable (gp.tropism gp) (vector-ref gp 6))
(define-integrable (gp.poisons gp) (vector-ref gp 7))

(define (eval-define-network exp env)
  (let ((header (car exp))
(body (cdr exp)))
    (let ((name (car header))
  (inputs (cadr header))
  (outputs (caddr header)))
      (if (there-exists? inputs (lambda (input) (memq input outputs)))
  (error "Input and Output terminals must be distinct"
 inputs outputs))
      (add-binding env name
   (make-network inputs outputs body)))))

(define (eval-with-locations w-l-exp env)
  (let ((params (car w-l-exp))
(body (cdr w-l-exp)))
    (add-binding
     env with-loc-symbol
```

```
     (lambda points
       (let ((pt-bindings (match-up params points))
     (new-env (extend-env env params points)))
 ;; setup initial point agenda
 (for-each
  (lambda (terminal)
    (for-each
     (lambda (point)
       (fork (lambda ()
       (project-seq terminal body new-env point
    (lambda ()
      (newline)
      (display `(,terminal done))
      (process-next-point))
    'stop
    #f))))
     (lookup-var new-env terminal)))
  params)
 (process-next-point))))))

;;;; Point continuations
;;;(net continuations and growing point continuations)

(define (call-pt-cont pt-cont point)
  ((cdr pt-cont) point))

(define (backtrack id)
  (cons 'gp-continuation
(lambda (point)
  (fork (lambda ()
  (let ((prev-point (get-backtracking-pt point id)))
    (cond ((procedure? prev-point) ; the backtracking cont
   (prev-point))
  ((not prev-point)
   (error "Backtracking beyond history at" point))
  (else
   (write-line `(,id backtracking from ,point
     to ,prev-point))
   (call-pt-cont (backtrack id) prev-point)
   (process-next-point)))))))))

(define (make-gp-continuation exps env cont stop id last-result size gp)
  (cons 'gp-continuation
(lambda (point)
  (fork (lambda ()
  (illustrate-seq exps env point cont stop
  id last-result size gp))))))

(define (make-net-continuation exps env cont stop id)
  (cons 'net-continuation
(lambda (point)
  (lambda (terminal)
    (fork (lambda ()
    (project terminal exps env point cont stop id)))))))

(define (make-backtracking-net-continuation exps env cont stop id new-id)
  (cons 'net-continuation
(lambda (point)
  (fork (lambda ()
```

```
  (call-pt-cont (backtrack new-id) point)
  (process-next-point)))
  (lambda (terminal)
    (fork (lambda ()
    (project terminal exps env point cont stop id)))))))

(define (net-cont? cont)
  (and (pair? cont)
       (eq? 'net-continuation (car cont))))

(define (gp-cont? cont)
  (and (pair? cont)
       (eq? 'gp-continuation (car cont))))

;;;; Utilities

(define (read-gpl-file filename)
  (let ((input-filename (if (not (pathname-type filename))
    (pathname-new-type filename "gpl")
    filename)))
    (read-file input-filename)))

(define (match-up name-list val-list)
  (if (not (= (length name-list) (length val-list)))
      (error "Wrong number of point types passed to WITH-LOCATIONS"))
  (extend-env '() name-list val-list))

(define (param->index param param-list)
  (let loop ((i 1) (params param-list))
    (cond ((null? params) #f)
  ((eq? param (car params)) i)
  (else (loop (+ i 1) (cdr params))))))

(define (names+indexes->list targets)
  (let ((result-vec (make-vector (length (cdr targets)))))
    (let loop ((lst (cdr targets)))
      (if (null? lst)
  (vector->list result-vec)
  (let ((name+index (car lst)))
    (vector-set! result-vec
 (entry/value name+index) (entry/tag name+index))
    (loop (cdr lst))))))))

(define (show-initial-points)
  (show-points *D* (mark-x 3) (flatten initial-points)))
```

### C.1.2 Growing point commands

**body.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(define debug:print-tropism-info #f)
(define debug:print-propagate-info #f)
(define show-trajectory? #f)           ; no longer supported

;;; Code to illustrate expressions in actions clause

(define (make-body per-step-instructions env params)
  ;; return a procedure that invokes the illustrator when given context.
  (lambda (point cont stop id last-result size gp args)
    (illustrate-seq (append per-step-instructions '((HALT)))
                    (extend-env env params args)
                    point cont stop id last-result size gp))
  )

(define (illustrate-seq commands env point cont stop id last-result size gp)
  (cond ((null? commands) (cont))      ; like a HALT
        ((null? (cdr commands))        ; tail recursion
         (illustrate-command (car commands) env point
                             cont stop id last-result size gp))
        (else
         (let ((cont* (lambda ()
                        ;; give other particles a chance between commands
                        (fork (lambda ()
                                (illustrate-seq (cdr commands) env point
                                                cont stop
                                                id last-result size gp)))
                        (process-next-point))))
           (illustrate-command (car commands) env point
                               cont* stop id last-result size gp)))))

(define (eval-exp exp env point)
  (cond ((symbol? exp)                 ; variable
         (lookup-var env exp))
        ((not (pair? exp))             ; constant
         exp)
        ((eq? (car exp) 'quote)
         (cadr exp))
        ((eq? (car exp) 'sensing?)
         (is-material? material-table point
                       (map (lambda (material-exp)
                              (eval-name-exp material-exp env point))
                            (cdr exp))))
        ((eq? (car exp) 'detecting?)
         (pheromone-present? (eval-name-exp (cadr exp) env point)
                             point))
        (else
         (let ((op (eval-exp (car exp) env point))
               (operands (map (lambda (operand)
                                (eval-exp operand env point))
                              (cdr exp))))
           (apply op operands)))))
```

```
(define (eval-name-exp exp env point)
  (cond ((not (pair? exp))                      ; unquoted name
         exp)
        ((eq? (car exp) ':)
         (lookup-var env (cadr exp)))
        (else
         (eval-exp exp env point))))

(define (illustrate-command exp env point cont stop id last-result size gp)
  ;; return a list of compiled expressions/statements
  (if (not (pair? exp))
      (error "Command must be enclosed in parentheses" exp)
      (case (car exp)
        ((SECRETE)
         (if (check-secrete (cdr exp))
             (illustrate-secrete (cdr exp) max env point cont id gp)
             (error "Bad SECRETE syntax:" exp)))
        ((SECRETE+)
         (if (check-secrete (cdr exp))
             (illustrate-secrete (cdr exp) + env point cont id gp)
             (error "Bad SECRETE+ syntax:" exp)))
        ((WHEN)
         (if (check-when (cdr exp))
             (illustrate-when (cdr exp) env point cont stop
                              id last-result size gp)
             (error "Bad WHEN syntax:" exp)))
        ((START-GROWING-POINT START-GP)
         (if (check-start-growing-point (cdr exp))
             (illustrate-start-growing-point (cdr exp) env point cont stop id)
             (error "Bad START-GROWING-POINT syntax:" exp)))
        ((PROPAGATE)
         (cond ((not (gp.tropism gp))
                (error "Propagate form illegal without tropism declaration"
                       exp))
               ((not (check-propagate (cdr exp)))
                (error "Bad PROPAGATE syntax:" exp))
               ((not (= (gp.arity gp) (length (cdr exp))))
                (error "PROPAGATE called with wrong number of arguments"
                       exp))
               (else (illustrate-propagate (cdr exp) env point cont stop
                                           id last-result size gp))))
        ((CONNECT -->)
         (if (check-connect (cdr exp))
             (illustrate-connect (cdr exp) env point cont stop
                                 id last-result size gp)
             (error "Bad CONNECT syntax:" EXP)))
        ((SERIALIZE)
         (if (check-serialize (cdr exp))
             (illustrate-serialize (cdr exp) env point cont stop
                                   id last-result size gp)
             (error "Bad SERIALIZE syntax:" exp)))
        ((->OUTPUT)
         (if (check-->output (cdr exp))
             (illustrate->output (cdr exp) env point cont stop)
             (error "->OUTPUT requires at least one name" exp)))
        ((HALT)
         (if (check-halt (cdr exp))
             (cont)                     ;(process-next-point)
```

```
             (error "Bad HALT syntax:" exp)))
        ((TERMINATE)
         (if (check-terminate (cdr exp))
             (illustrate-terminate point cont stop)
             (error "Bad TERMINATE syntax:" exp)))
        (else (error "Unrecognised Growing Point command" exp)))))

(define (illustrate-start-growing-point exp env point cont stop id)
  (let ((new-gp (eval-exp (car exp) env point))
        (args (map (lambda (arg-exp)
                     (eval-exp arg-exp env point))
                   (cdr exp))))
    (let ((materials (gp.material new-gp)))
      (let ((size* ((gp.size new-gp) point args)))
        ((gp.body new-gp)
         point
         (lambda ()
           ;;(add-materials material-table point materials)
           (widen point size* new-gp)
           (cont))
         ;; id = #f unless commands are serialised
         stop id #f size* new-gp args)))))

(define (illustrate-propagate exp env point cont stop
                              id last-result size gp)
  (let ((args (map (lambda (arg) (eval-exp arg env point))
                   exp))
        (materials (gp.material gp))
        (filter-gen (gp.tropism gp))
        (targets (gp.targets gp)))

    (define (process-point new-point last-result)
      (if show-trajectory?
          (draw-line point new-point colour-table materials))
      (newline)
      (display `(,(gp.name gp) ,id propagated from
                               ,(D:point->pt point) to
                               ,(D:point->pt new-point)))
      ;;(put (gp.history gp) point #t)
      (and id (add-backtracking-pt new-point id point))
      (widen point size gp)
      (fork (lambda ()
              ((gp.body gp)
               new-point
               (lambda ()
                 ;;(add-materials material-table new-point materials)
                 (widen new-point size gp)
                 (process-next-point))
               stop id last-result size gp args))
            cont)
      (process-next-point))

    (define (stuck)
      (write-line
       `(,(gp.name gp) ,id stuck: giving up
                       at ,point))
      (illustrate-terminate point cont stop))

    (define (try-to-get-next-point n)
```

```
      (define (retry)
        ;; block and try again
        (fork (lambda ()
                (try-to-get-next-point (- n 1))))
        (process-next-point))

      (let ((current-result
             (make-entry point (D:get-values *D* targets point)))
            (neighbourhood-values
             (D:get-nbd-vals *D* targets point (gp.poisons gp))))
        (let ((last-result (or last-result current-result)))
          (let ((empty-case (if (> n 0) retry stuck))
                (choose-the-first
                 (lambda (pset)
                   (if debug:print-propagate-info
                       (begin
                         (pp `(values: ,current-result ,last-result))
                         (write-line 'choices:)
                         (pp pset)))
                   (process-point (D:first-point *D* pset)
                                  last-result)))
                (missing-pheromone (missing-p? current-result)))

            (cond (missing-pheromone
                   (write-line `(,(gp.name gp)
                                 ,id
                                 waiting for pheromone
                                 ,(vector-ref targets missing-pheromone)
                                 at ,point))
                   (fork (lambda () (try-to-get-next-point n)))
                   (process-next-point))
                  (else
                   (let ((last-result (or last-result current-result)))
                     (if debug:print-tropism-info
                         (begin
                           (write-line `(,(gp.name gp)
                                         applying tropism at ,point (n: ,n)))
                           (pp `(values: ,current-result ,last-result))
                           (write-line 'nbd-values:)
                           (pp neighbourhood-values)))
                     (((filter-gen empty-case choose-the-first)
                       current-result last-result)
                      neighbourhood-values))))))))

    (try-to-get-next-point 3)))

(define (illustrate-serialize exp env point cont stop id last-result size gp)
  ;; create a new id for each spawning command except for the last.
  (if (null? (cdr exp))
      (illustrate-command (car exp) env point cont stop
                          id last-result size gp)
      (let ((next (car exp)))
        (if (spawn-command? next)
            (let ((new-id (random id-range)))
              (add-backtracking-pt
               point new-id
               (lambda ()
                 ;; need to remove backtracking point for new-id here.
```

```
                (illustrate-seq `(SERIALIZE ,@(cdr exp)) env point
                                process-next-point stop
                                id last-result size gp)))
             (illustrate-command next env point cont
                                 (backtrack new-id)
                                 new-id last-result size gp))
           (illustrate-seq next env point
                           (lambda ()
                             (illustrate-seq `(SERIALIZE ,@(cdr exp))
                                             env point cont stop
                                             id last-result size gp))
                           stop id last-result size gp)))))

(define (illustrate-connect exp env point cont stop id last-result size gp)
  ;; create a gp-continuation of rest of exps and pass it along
  (let ((start-command (car exp))
        (end-commands (cdr exp)))
    (illustrate-command start-command env point cont
                        (make-gp-continuation end-commands env
                                              process-next-point
                                              stop id last-result size gp)
                        id last-result size gp)))

(define (illustrate-terminate point cont stop)
  ;; A point may be processing more than one GP, so we must call (cont)
  ;; instead of process-next-point
  (cond ((eq? stop 'stop)
         (cont))                        ; not connected to another gp
        ((net-cont? stop)
         ((call-pt-cont stop point) #f) ; terminate all pending continuations
         (cont))
        (else
         (write-line `(CONNECTING at ,point))
         (call-pt-cont stop point)
         (cont))))

(define (illustrate->output exp env point cont stop)
  ;;(pp `(illustrate->output: ,exp ,point ,cont ,stop))
  (cond ((eq? stop 'stop)
         (cont))
        ((net-cont? stop)
         (for-each (call-pt-cont stop point) exp)
         (cont))
        (else
         (error `(->output ,@exp) "appears in illegal context"))))

(define (illustrate-when exp env point cont stop id last-result size gp)
  ;; assume exp is not empty
  (let loop ((clauses exp))
    (if (null? clauses)
        (error "WHEN statement does not have have DEFAULT clause")
        (let ((clause (car clauses)))
          (let ((predicate (if (eq? (car clause) 'default)
                               #t
                               (eval-exp (car clause) env point))))
            (if predicate
                (illustrate-seq (cdr clause) env point cont stop
                                id last-result size gp)
                (loop (cdr clauses))))))))
```

```scheme
(define (illustrate-secrete secrete-exp combiner env point cont id gp)
  (let ((extent (eval-exp (car secrete-exp) env point))
        (rest-args (map (lambda (exp) (eval-exp exp env point))
                        (cddr secrete-exp))))
    (let ((pheromone (pheromone-name (cadr secrete-exp))))
      (newline)
      (if gp
          (display `(,(gp.name gp) ,id secreted ,pheromone at ,point
                                    with extent ,extent))
          (display `(top-level secretion of ,pheromone at ,point
                                with extent ,extent)))
      (D:record-secretion *D* pheromone point extent combiner)
      (cont))))

;;; Syntax checkers

(define (check-secrete exp)
  (and (pair? exp)
       (pair? (cdr exp))))

(define (check-when exp)
  (for-all? exp pair?))

(define (check-start-growing-point exp)
  (pair? exp))

(define (check-terminate exp)
  (null? exp))

(define (check-halt exp)
  (null? exp))

(define (check-propagate exp)
  #t)

(define (check-connect exp)
  (and (pair? exp)
       (or (check-start-growing-point (car exp))
           (check-connect (car exp)))))

(define (check-serialize exp)
  (pair? exp))

(define (check-->output exp)
  (pair? exp))

(define (spawn-command? command)
  ;; spawn commands terminate at locations other than the current one.
  ;; used in the context of serialize command, perhaps could be used to
  ;; expand allowed context for connect.
  (and (pair? command)
       (memq (car command)
             '(start-growing-point start-gp propagate connect))))

;;; Helpers

(define (pheromone-present? ph-name point)
  (> (or (D:get-pheromone-value *D* ph-name point) 0)
```

```
     pheromone-threshold))

(define (widen point size gp)
  (let ((materials (gp.material gp))
        (nbd (g-filter
              (lambda (nbr)
                (not (there-exists?
                      (gp.poisons gp)
                      (lambda (poison) (pheromone-present? poison nbr)))))
              (D:get-nbd point size empty-point-table))))
    (add-materials material-table point materials)
    (for-each (lambda (pt)
                (add-materials material-table pt materials))
              nbd)
    (D:display-domain *D*)
    ;;(D:widen *D* point size colour-table
    ;;         (get-materials material-table point))
    ))

(define (add-materials table point materials)
  (put (table.pt table) point materials)
  (let ((colour? (assm (get-materials table point) colour-table)))
    (if colour?
        (D:update-point point (cadr colour?)))))

(define (update-material-col point col-table)
  (let ((colour? (assm (get-materials material-table point) col-table)))
    (if colour?
        (D:update-point point (cadr colour?)))))

(define (show-materials col-table)
  (iter-points *D*
               (lambda (point)
                 (update-material-col point col-table)))
  (d:display-domain *D*))

(define (draw-line point new-point colour-table materials)
  (let ((colour? (assm materials colour-table)))
    (if colour?
        (D:draw-line *D* point new-point (cadr colour?)))))

(define (pheromone-name pheromone-exp)
  ;; used to analyse sensing? expression.
  (let loop ((exp pheromone-exp) (anti? #f))
    (if (not (pair? exp))
        exp
        (case (car exp)
          ((inhibitor)
           (if anti?
               `(anti ,(cadr exp))
               (cadr exp)))
          ((activator)
           (if anti?
               `(anti ,(cadr exp))
               (cadr exp)))
          ((anti) (loop (cadr exp) (not anti?)))
          (else (error "Unrecognized pheromone specifier" exp))))))

(define (pheromone-name->key name)
```

```
  (if (not (pair? name))
      name
      (cadr name)))

(define primitives `((= ,=) (> ,>) (< ,<) (<= ,<=) (>= ,>=)
                           (+ ,+) (- ,-) (* ,*) (/ ,/)
                           (or ,(lambda (x y) (or x y)))
                           (and ,(lambda (x y) (and x y)))))

(define *identifier-count* 0)

(define (->id name)
  (set! *identifier-count* (+ *identifier-count* 1))
  (symbol-append name '-
                 (string->symbol (number->string *identifier-count*))))
#|
(define (id-cont val)
  ;; identity continuation
  val)
|#
```

### C.1.3 Network commands

**networks.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(define (project-seq terminal seq env point cont stop id)
  (cond ((null? seq) (cont))
        ((null? (cdr seq))
         (project terminal (car seq) env point cont stop id))
        (else
         (project terminal (car seq) env point
                  (lambda ()
                    (project-seq terminal (cdr seq) env point cont stop id))
                  stop
                  id))))

(define (project terminal exp env point cont stop id)
  ;;(pp `(projecting over ,terminal ,exp ,cont ,stop))
  (cond ((not (pair? exp))
         (error "Unrecognised network command" exp))
        ((not terminal)
         (illustrate-terminate point cont stop))
        ((at-exp? exp)
         (project-at terminal (cadr exp) (cddr exp) env point cont stop id))
        ((location-binding? exp)
         (project-location-binding terminal exp env point cont stop id))
        ((cascade? exp)
         (project-cascade terminal (cdr exp) env point cont stop id))
        (else
         (project-seq terminal exp env point cont stop id))))

(define (project-at terminal at-terminal at-body env point cont stop id)
  (if (eq? terminal at-terminal)
      (illustrate-seq at-body env point cont stop id #f 0 #f)
      (cont)))

(define (project-cascade terminal exp env point cont stop id)
  (let ((inputs (car exp))
        (rest-cascade (cddr exp)))
    (if (memq terminal inputs)
        (if (null? rest-cascade)
            (illustrate->output
             (list (alias terminal inputs (cadr exp)))
             env point cont stop)
            (let ((first-net (eval-exp (cadr exp) env point)))
              (if (network? first-net)
                  (begin
                    ;;(write-line `(projecting over ,terminal ,first-net))
                    (project (alias terminal inputs (net.inputs first-net))
                             (net.body first-net)
                             env point cont
                             (make-net-continuation
                              (make-cascade-exp (net.outputs first-net)
                                                rest-cascade)
                              env process-next-point stop id)
                             id))
```

```
                (error "The object" first-net
                       "passed to ==> is not a network"))))
        (cont))))

(define (project-location-binding terminal exp env point cont stop id)
  ;;(pp `(projecting let-locs over ,terminal ,exp))
  (let ((binding-vars (loc-binding-terminals exp))
        (binding-exps (loc-binding-exps exp))
        (body (loc-binding-body exp)))
    (let ((new-id (random id-range))
          (used-in-binding? (memq terminal (exp.inputs binding-exps)))
          (used-in-body? (and (not (memq terminal binding-vars))
                              (memq terminal (exp.inputs body)))))
      (if used-in-binding?
          (if used-in-body?
              (project terminal binding-exps env point
                       (lambda ()
                         (pp `(deferring ,body at ,point))
                         (add-backtracking-pt
                          point new-id
                          (lambda ()
                            (project-seq terminal body env point
                                         process-next-point stop id)))
                         (cont))
                       (make-backtracking-net-continuation
                        body env process-next-point stop id new-id)
                       new-id)
              (project terminal binding-exps env point cont
                       (make-net-continuation body env process-next-point
                                              stop id)
                       id))
          (if used-in-body?
              (project-seq terminal body env point cont stop id)
              (cont)))
      )))

(define (exp.inputs exp)
  (cond ((not (pair? exp))
         '())
        ((at-exp? exp)
         (list (cadr exp)))
        ((cascade? exp)
         (cadr exp))
        ((location-binding? exp)
         (diff (union (exp.inputs (loc-binding-exps exp))
                      (exp.inputs (loc-binding-body exp)))
               (loc-binding-terminals exp)))
        (else (append (exp.inputs (car exp))
                      (exp.inputs (cdr exp))))))

(define (exp.outputs exp)
  (cond ((not (pair? exp))
         '())
        ((eq? (car exp) '->OUTPUT)
         (cdr exp))
        ((at-exp? exp)
         (append (exp.outputs (caddr exp))
                 (exp.outputs (cdddr exp))))
        ((cascade? exp)
```

```
         (car (last-pair exp)))
        ((location-binding? exp)
         (diff (union (exp.outputs (loc-binding-exps exp))
                      (exp.outputs (loc-binding-body exp)))
               (loc-binding-terminals exp)))
        (else (append (exp.outputs (car exp))
                      (exp.outputs (cdr exp))))))

(define (terminal->index terminal-list terminal)
  (let loop ((i 0) (lst terminal-list))
    (cond ((null? lst) #f)
          ((eq? terminal (car lst)) i)
          (else (loop (+ i 1) (cdr lst))))))

(define (alias terminal terminals1 terminals2)
  (let loop ((lst1 terminals1) (lst2 terminals2))
    (cond ((null? lst1) #f)
          ((null? lst2) (error "Incompatible terminal coupling"
                               terminals1 terminals2))
          ((eq? terminal (car lst1))
           (car lst2))
          (else (loop (cdr lst1) (cdr lst2))))))

(define (make-network inputs outputs body)
  ;; no environment because outputs are dynamically bound.
  (list 'network
        inputs
        outputs
        body))

(define (network? net) (and (pair? net)
                            (eq? (car net) 'network)
                            (= (length net) 4)))

(define (net.inputs network) (cadr network))
(define (net.outputs network) (caddr network))
(define (net.body network) (cadddr network))

(define (loc-binding-terminals exp)
  (fold-right append '() (every-other (cadr exp))))

(define (loc-binding-exps exp)
  (fold-right cons '() (every-other (cdr (cadr exp)))))

(define (loc-binding-body exp)
  (cddr exp))

;;; Syntax

(define (at-exp? exp)
  (and (pair? exp)
       (eq? (car exp) 'AT)))

(define (location-binding? exp)
  (and (pair? exp)
       (eq? (car exp) 'LET-LOCS)
       (pair? (cdr exp))
       (list? (cadr exp))
       (pair? (cddr exp))))
```

```
(define (cascade? exp)
  (and (pair? exp)
       (or (eq? (car exp) '==>)
           (eq? (car exp) 'cascade))
       (pair? (cdr exp))
       (list? (cadr exp))))

(define (make-cascade-exp inputs rest-exp)
  `(==> ,inputs ,@rest-exp))
```

### C.1.4 Tropism Implementation

**tropism.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

;;; Part of growing point language.  Implementation of tropism parsing

(define (analyse-tropism tropism-exp targets)
  (let ((filter-gen-maker (tropism->filter tropism-exp targets)))
    (let ((sort-gen (pred-gen->sorter-gen
                     (tropism->pred-gen tropism-exp targets))))
      (lambda (empty-case non-empty-case)
        (lambda (my-result last-result)
          (let ((sorter (sort-gen my-result last-result)))
            ((filter-gen-maker
              empty-case
              (lambda (results)
                (non-empty-case (sorter results))))
             my-result last-result)))))))

(define (pred-gen->sorter-gen pred-gen)
  (lambda (my-result last-result)
    (let ((make-predicate (pred-gen my-result last-result)))
      (lambda (result-list)
        (g-sort result-list (make-predicate result-list))))))

(define (tropism->pred-gen exp targets)
  ;; returns a result of the form:
  ;; (lambda (my-result last-result)
  ;;   (lambda (result-list)
  ;;     (g-sort ... result-list)))
  (if (not (pair? exp))
      (error "Bad TROPISM syntax: " exp)
      (let ((op (car exp)))
        (case op
          ((ortho+)
           (make-ortho-pred-gen > (get-index (cadr exp) targets)))
          ((ortho-)
           (make-ortho-pred-gen < (get-index (cadr exp) targets)))
          ((plagio+ plagio-)
           (make-plagio-pred-gen (get-index (cadr exp) targets)))
          ((dia)
           (make-dia-pred-gen (get-index (cadr exp) targets)))
          (else
           (cond ((null? (cdr exp))
                  (error "Ill formed tropism" exp))
                 ((null? (cddr exp))
                  (tropism->pred-gen (cadr exp) targets))
                 (else
                  (compose-pred-gen
                   (tropism->pred-gen (cadr exp) targets)
                   (tropism->pred-gen (cons op (cddr exp))
                                      targets)))))))))

(define (compose-pred-gen gen1 gen2)
  (lambda (my-result last-result)
```

```
    (let ((make-pred1 (gen1 my-result last-result))
          (make-pred2 (gen2 my-result last-result)))
      (lambda (result-list)
        (let ((pred1 (make-pred1 result-list))
              (pred2 (make-pred2 result-list)))
          (lambda (elt1 elt2)
            (or (pred1 elt1 elt2)
                (and (not (pred1 elt2 elt1))
                     (pred2 elt1 elt2)))))))))

(define (make-ortho-pred-gen comparator index)
  (define ref (make-ref index))
  (let ((predicate (make-2-predicate comparator ref)))
    (lambda (my-result last-result)
      (lambda (result-list)
        my-result last-result           ;ignored for
        result-list                     ;straightforward < or >
        predicate))))

(define (make-dia-pred-gen index)
  (define ref (make-ref index))
  (lambda (my-result last-result)
    (let ((predicate (make-proximity-pred ref (ref last-result))))
      (lambda (result-list)
        my-result result-list           ; ignored when last-result is target
        predicate))))

(define (make-plagio-pred-gen index)
  (define ref (make-ref index))
  (lambda (my-result last-result)
    my-result last-result               ; ignored
    (lambda (result-list)
      (find-min-max
       (map ref result-list)
       (lambda (lo hi)
         (let ((current (ref my-result)))
           (if (< hi current)
               (make-proximity-pred ref (/ (+ lo current) 2))
               (make-proximity-pred ref (/ (+ current hi) 2)))))))))

(define (make-proximity-pred ref base-val)
  (lambda (elt1 elt2)
    (< (abs (- (ref elt1) base-val))
       (abs (- (ref elt2) base-val)))))

(define (make-ref index)
  ;; values of #f count numerically as 0
  (lambda (elt)
    (or (vector-ref (entry/value elt) index)
        0)))

(define (make-1-predicate comparator index my-result)
  (define ref (make-ref index))
  (let ((bv (ref my-result)))
    (lambda (result-element)
      (comparator (ref result-element) bv))))

(define (make-2-predicate comparator ref)
  ;; values of #f count numerically as 0 except for = comparison
```

```
  (lambda (elt1 elt2)
    (let ((val1 (or (ref elt1) 0))
          (val2 (or (ref elt2) 0)))
      (comparator val1 val2))))

(define (find-min-max val-lst k)
  (if (null? val-lst)
      (error "Cannot find extrema of empty list--FIND-MIN-MAX")
      (let loop ((lst val-lst) (lo (car val-lst)) (hi (car val-lst)))
        (if (null? lst)
            (k lo hi)
            (let ((first (car lst)))
              (cond ((< first lo) (loop (cdr lst) first hi))
                    ((> first hi) (loop (cdr lst) lo first))
                    (else (loop (cdr lst) lo hi)))))))))

(define (find-near target tolerance ref lst)
  (let ((ok-lo (- target tolerance))
        (ok-hi (+ target tolerance)))
    (let loop ((lst lst))
      (cond ((null? lst) '())
            ((<= ok-lo (ref (car lst)) ok-hi)
             (cons (car lst) (loop (cdr lst))))
            (else (loop (cdr lst)))))))

(define (tropism->filter exp targets)
  ;; returns a procedure of the form
  ;; (lambda (empty-case unique-case multiple-case)
  ;;   (lambda (my-result last-result)
  ;;     (lambda (result-list) ...)))
  ;; my-result, like each element of result-list, is a pair of the
  ;; point id and a vector of the pheromone levels specified by targets

  (if (not (pair? exp))
      (error "Bad TROPISM syntax: " exp)
      (let ((op (car exp)))
        (case op
          ((ortho+ ortho- plagio+ plagio- dia)
           (D:make-primitive-filter exp targets))
          ((and)
           (make-and-filter (cdr exp) targets))
          ((~and)
           (make-~and-filter (cdr exp) targets))
          ((or)
           (make-or-filter (cdr exp) targets))
          ((~or)
           (make-~or-filter (cdr exp) targets))
          (else (error "Bad TROPISM syntax: " exp))))))

(define (D:make-primitive-filter exp targets)
  (let ((index (get-index (cadr exp) targets)))
    (case (car exp)
      ((ortho+)
       (make-extreme-filter > index))
      ((ortho-)
       (make-extreme-filter < index))
      ((plagio+)
       (make-filter > index))
      ((plagio-)
```

```
       (make-filter < index))
      ((dia)
       (make-eqv-filter index))
      (else (error "Unknown primitive tropism")))))

(define (make-filter comparator index)
  (define ref (make-ref index))

  (lambda (empty-case non-empty-case)
    (lambda (my-result last-result)
      (let ((1-predicate (make-1-predicate comparator index my-result)))
        (lambda (result-list)
          (find-min-max
           (map (lambda (result) (or (ref result) 0)) result-list)
           (lambda (lo hi)
             (let ((filtered-results (g-filter 1-predicate result-list)))
               (if (null? filtered-results)
                   (empty-case)
                   (let ((my-val (ref my-result)))
                     (let ((final-results
                            (if (comparator lo my-val)
                                (find-near (/ (+ lo my-val) 2)
                                           (* .5 (- my-val lo))
                                           ref
                                           filtered-results)
                                (find-near (/ (+ my-val hi) 2)
                                           (* .5 (- hi my-val))
                                           ref
                                           filtered-results))))
                       (if (null? final-results)
                           (empty-case)
                           (non-empty-case final-results))))))))))))

(define (make-extreme-filter comparator index)
  (define ref (make-ref index))

  (lambda (empty-case non-empty-case)
    (lambda (my-result last-result)
      (let ((1-predicate (make-1-predicate comparator index my-result)))
        (lambda (result-list)
          (let ((good-results (g-filter 1-predicate result-list)))
            (if (null? good-results)
                (empty-case)
                (non-empty-case good-results))))))))

(define (make-eqv-filter index)
  (define ref (make-ref index))

  (lambda (empty-case non-empty-case)
    (lambda (my-result last-result)
      (let ((base-val (ref last-result))
            (current-val (ref my-result)))

        (define (make-eqv-1-predicate range)
          (lambda (result-element)
            (let ((val (ref result-element)))
              (< (abs (- val base-val))
                 range                  ; (* .1 base-val)
                 ))))
```

```
        (lambda (result-list)
          (if (null? result-list)
              (empty-case)
              (find-min-max
               (map ref result-list)
               (lambda (lo hi)
                 #|
                 (newline)
                 (display "DIA: (lo hi):")
                 (display (list lo hi))
                 (newline)
                 |#
                 (let ((filtered-results
                        (g-filter (make-eqv-1-predicate (/ (- hi lo) 2))
                                  result-list)))
                   (if (null? filtered-results)
                       (empty-case)
                       (non-empty-case filtered-results))))))))))

(define (make-and-filter clauses targets)
  (if (null? clauses)
      (error "No clauses in TROPISM (and) form: " exp)
      (let ((filters (map (lambda (clause)
                            (tropism->filter clause targets))
                          clauses)))
        (let loop ((my-filters filters))
          (if (null? (cdr my-filters))
              (car my-filters)
              (let ((rest-filter (loop (cdr my-filters))))
                (lambda (empty-case non-empty-case)
                  (lambda (my-result last-result)
                    (let ((cont ((rest-filter
                                  empty-case non-empty-case)
                                 my-result last-result)))
                      (((car my-filters) empty-case cont)
                       my-result last-result))))))))))

(define (make-~and-filter clauses targets)
  (if (null? clauses)
      (error "No clauses in TROPISM (~and) form: " exp)
      (let ((filters (map (lambda (clause)
                            (tropism->filter clause targets))
                          clauses)))
        (let loop ((my-filters filters))
          (if (null? (cdr my-filters))
              (car my-filters)
              (let ((rest-filter (loop (cdr my-filters))))
                (lambda (empty-case non-empty-case)
                  (lambda (my-result last-result)
                    (let ((cont (lambda (results)
                                  (((rest-filter
                                     (lambda () (non-empty-case results))
                                     non-empty-case)
                                    my-result last-result)
                                   results))))
                      (((car my-filters) empty-case cont)
                       my-result last-result))))))))))
```

```
(define (make-or-filter clauses targets)
  (if (null? clauses)
      (error "No clauses in TROPISM (or) form: " exp)
      (let ((filters (map (lambda (clause)
                            (tropism->filter clause targets))
                          clauses)))
        (let loop ((my-filters filters))
          (if (null? (cdr my-filters))
              (car my-filters)
              (let ((rest-filter (loop (cdr my-filters))))
                (lambda (empty-case non-empty-case)
                  (lambda (my-result last-result)
                    (let ((cont ((rest-filter
                                   empty-case non-empty-case)
                                  my-result last-result)))
                      (lambda (results)
                        ((((car my-filters) (lambda () (cont results))
                                            non-empty-case)
                          my-result last-result)
                         results))))))))))

(define (make-~or-filter clauses targets)
  (if (null? clauses)
      (error "No clauses in TROPISM (~or) form: " exp)
      (let ((filters (map (lambda (clause)
                            (tropism->filter clause targets))
                          clauses)))
        (let loop ((my-filters filters))
          (if (null? (cdr my-filters))
              (car my-filters)
              (let ((rest-filter (loop (cdr my-filters))))
                (lambda (empty-case non-empty-case)
                  (lambda (my-result last-result)
                    (lambda (result-list)
                      (let ((empty-cont
                             (lambda ()
                               (((rest-filter
                                   empty-case non-empty-case)
                                  my-result last-result)
                                 result-list)))
                            (non-empty-cont
                             (lambda (results)
                               (((rest-filter
                                   (lambda () (non-empty-case results))
                                   (lambda (rest-results)
                                     (non-empty-case
                                      (union results rest-results))))
                                  my-result last-result)
                                 result-list))))
                        ((((car my-filters)
                           empty-cont non-empty-cont)
                          my-result last-result)
                         result-list)))))))))))

(define (get-index pheromone-name pairings)
  (let loop ((lst (cdr pairings)) (largest -1))
    (if (null? lst)
        (let ((new-largest (+ largest 1)))
          (set-cdr! pairings
```

```
                     (cons (make-entry pheromone-name new-largest)
                           (cdr pairings)))
           new-largest)
         (let ((name+index (car lst)))
           (if (eq? (entry/tag name+index) pheromone-name)
               (entry/value name+index)
               (loop (cdr lst) (max largest (entry/value name+index)))))))

(define (names+indexes->list targets)
  (let ((result-vec (make-vector (length (cdr targets)))))
    (let loop ((lst (cdr targets)))
      (if (null? lst)
          (vector->list result-vec)
          (let ((name+index (car lst)))
            (vector-set! result-vec (cdr name+index) (car name+index))
            (loop (cdr lst)))))))

(define (test-tropism exp)
  (let ((targets (list 'targets)))
    (let ((filter-gen-gen (analyse-tropism exp targets)))
      (let ((targets-vec (list->vector (names+indexes->list targets))))
        (let ((filter-gen (filter-gen-gen (lambda () 'empty)
                                          (lambda (lst) lst))))
          (lambda (prev-result current-result)
            (lambda (result-list)
              ((filter-gen current-result prev-result)
               result-list))))))))

(define (test-tropism-on-points exp)
  (let ((targets (list 'targets)))
    (let ((filter-gen-gen (analyse-tropism exp targets)))
      (let ((targets-vec (list->vector (names+indexes->list targets))))
        (let ((filter-gen (filter-gen-gen (lambda () 'empty)
                                          (lambda (lst) lst))))
          (lambda (prev current)        ; points
            (let ((prev-result
                    (make-entry prev (D:get-values *D* targets-vec prev)))
                   (current-result
                    (make-entry current
                                (D:get-values *D* targets-vec current))))
              (write-line prev-result)
              (write-line current-result)
              ((filter-gen current-result prev-result)
               (D:get-nbd-vals *D* targets-vec current '()))))))))
```

## C.2 Domain Implementation

This section provides the code listing for drawing the effects of interpreted GPL code on the screen. There are three files involved: `layouts.scm`, `placement.scm` and `domains2.scm`. The first two files provide a number of procedures for generating random distributions of particles. The file `domains2.scm` contains the code that actually draws on the screen, maintains data structures for the whole system of particles, and even maintains some of the state like pheromone concentrations.

**domains2.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(load "~newts/gunk/toolbox/neighbours")
(load-option 'hash-table)

;;; Domains maintain the point set keeping track of points and their
;;; neighbours.  A domain supplies a method for obtaining the neighbourhood
;;; of a given point.

;;; Top level user functions:
;;;  (display-domain D)
;;;    show all points of D, unless D is discrete integer lattice
;;;    or continuous domain (when they are implemented).
;;;  (set-active-domain! D)
;;;    causes the illustrator to use D as its domain.
;;;  (win->domain win ds)
;;;    returns a domain with win as its window and ds as the step size.
;;;    The points are taken as all points on the square lattice of pixels in
;;;    the window.
;;;  (layout->domain layout ds win)
;;;    returns a domain with the points taken from layout, step size ds and
;;;    window win.  If win is #f, a new window is created and initialized.

;;; Functions called by illustrator:
;;;  (D:reset D)
;;;    called once before GPL program is interpreted.
;;;  (D:get-values D targets point)
;;;    returns the pheromone values at point of those specified by name
;;;    in targets
;;;  (D:get-nbd-vals D targets point)
;;;    returns the values of the pheromones specified in targets at all
;;;    points in the neighbourhood of point
;;;  (D:get-pheromone-value D pheromone point)
;;;    returns the value of pheromone at point
;;;  (D:record-secretion D pheromone point extent combiner)
;;;    updates p-table.
;;;  (D:first-point D pt-list)
;;;    returns the first element of pt-list

;;; Debugging/Utility functions:
;;;  (show-pheromone D pheromone)
;;;   shades the locations that have non-zero values for pheromone

(define preferred-window-size 300)
(define default-background "#CCCCCC")
```

```
(define default-foreground "wheat")
(define pt-pixel-size 5) ; radius of circle rep. point
(define margin-pix (* 2 pt-pixel-size)) ; so particles don't get clipped

(define pheromone-threshold 1) ; depends on secretion implementation

;;(define D:metric #f)

;;;; Variables that get set when a domain is created/initialised
(define *D* #f) ; the active domain
;;(define p-table #f) ; the pheromone-table
(define ds 5) ; step size
(define pt-size 2) ; size of point

(define AVG-NBD-SIZE 10)
;;; These procedures get defined after *D* gets set.
;;; They represent cached values of lookups of fields in *D*.

(define D:get-nbd
  ;; method for getting list of nbrs
  (lambda (point radius exclusions)
    point radius exclusions
    #f))

(define %D:update-point
  ;; cache of update method
  (lambda (point colour)
    point colour
    #f))

(define D:point->pt
  ;; convert points to coordinate rep.
  (lambda (point)
    point
    #f))

;;; Setup Procedures

(define (new-domain domain)
  (set-active-domain! domain)
  (reset-display domain default-background) ; background colour
  (init-display domain default-foreground) ; colour of particles
  (D:display-domain domain)
  unspecific)

(define (set-active-domain! domain)
  (set! *D* domain)
  (if (eq? (domain.type domain) 'discrete-int-lattice)
      (use-euclidean-secretions)
      (use-path-secretions))
  (set! D:point->pt (domain.point->pt domain))
  (set! %D:update-point ((domain.display-mgr domain) 'update))
  (set! D:get-nbd (domain.nbd-getter domain))
  (set! ds (domain.ds domain))
  (set! AVG-NBD-SIZE (ceiling->exact (nbd-mean domain)))
  'active-domain-set)

(define (ensure-domain domain)
  (if (domain? domain)
```

```
      domain
      (let ((result (win->domain (make-window preferred-window-size
      preferred-window-size)
 4.)))
(set-active-domain! result)
result)))

(define (D:reset domain)
  (let ((domain (ensure-domain domain)))
    (set-domain.p-table! domain (make-empty-table eq?))
    (reset-display domain default-background) ; background colour
    (init-display domain default-foreground) ; colour of particles
    (set! ds (domain.ds domain))
    (set! pt-size (domain.pt-size domain))
    (set! D:get-nbd (domain.nbd-getter domain))
    (set! D:point->pt (domain.point->pt domain))
    (set! %D:update-point ((domain.display-mgr domain) 'update))
    ;;(set! p-table (domain.p-table domain))
    (D:display-domain domain)
    ))

;;;; Various display methods.
;;; The first is very slow.
;;; The version that uses the draw-points graphics operation does not work
;;; on the Debian/XFree86 S3virge Xserver (Oct. 11, 1998).
;;; The image version, implemented in the display manager, is probably the
;;; fastest, but is only recently functional because it took considerable
;;; effort to implement.

(define (display-domain domain)
  (let ((win (domain.win domain))
(point->pt (domain.point->pt domain)))
    ;; Every point in the domain has unique coordinates
    (graphics-clear win)
    (if (eq? (domain.type domain) 'discrete-int-lattice)
'done
(with-values
    (lambda () (pixels->units win))
  (lambda (xscale yscale)
    (let ((coords-cache (fill-circle-coords pt-pixel-size)))
      (let ((x-cache (map (lambda (xy) (* xscale (car xy)))
  coords-cache))
    (y-cache (map (lambda (xy) (* yscale (cdr xy)))
  coords-cache))
    (x-list '())
    (y-list '()))
(iter-points domain
     (lambda (point)
       (let ((pt-x (pt.x (point->pt point)))
     (pt-y (pt.y (point->pt point))))
 (set! x-list
       (append (map (lambda (x) (+ x pt-x))
    x-cache)
       x-list))
 (set! y-list
       (append (map (lambda (y) (+ y pt-y))
    y-cache)
       y-list)))))
(graphics-operation win 'set-foreground-color "wheat")
```

```
(graphics-operation win 'draw-points
    (flo:list->vector x-list)
    (flo:list->vector y-list)))))))))
;;;; Updating and Accessing Secretions

(define (euclidean:record-secretion domain p-name point ext combiner)
  ;; distribute pheromone p-name
  (put (domain.p-table domain) p-name (make-tuple point (phi ext) combiner)))

(define (euclidean:get-pheromone-value domain p-name point)
  ;; returns concentration of p-name at pt
  (let ((pt-list (get (domain.p-table domain) p-name)))
    ;; entries are stored in reverse order, so fold right to combine in
    ;; same order as secretions were produced.
    (and pt-list
 (fold-right (lambda (tuple result)
       ((tuple.combiner tuple)
((tuple.f tuple)
 (D:euclidean-metric domain point (tuple.pt tuple)))
result))
     0
     pt-list))))

(define (consolidate table1 table2 combiner)
  (for-each (lambda (t2-entry)
      (let ((key (car t2-entry))
    (t2-value (cdr t2-entry)))
(let ((t1-value (get table1 key)))
  (if t1-value
      (replace table1 key (combiner t1-value t2-value))
      (put table1 key (car t2-value))))))
    (table.entries table2))
  table1)

(define (consolidate-hash-tbl table1 table2 combiner)
  (hash-table/for-each
   table2
   (lambda (key t2-value)
     (let ((t1-value (get table1 key)))
       (if t1-value
   (replace table1 key (combiner t1-value t2-value))
   (put table1 key t2-value)))))
  table1)

(define (path:record-secretion domain p-name point ext combiner)
  ;; distribute pheromone p-name

  (define (add lst point val)
    (let ((entry? (assoc point lst)))
      (if entry?
  (begin (set-cdr! entry? (+ (cdr entry?) val))
 lst)
  (cons (cons point val) lst))))

  (let ((get-nbd (domain.nbd-getter domain))
(values (make-equal-hash-table))
(marks (make-equal-hash-table))
(p-threshold 1)
(u0 (expt AVG-NBD-SIZE ext))
```

```
(alpha (/ 1. AVG-NBD-SIZE)))
    (let loop ((front (list (cons point u0))))
      ;;(write-line front)
      ;;(write-line marks)
      (for-each (lambda (point+val)
  (let ((front-pt (car point+val)))
    (hash-table/put! values front-pt
     (+ (cdr point+val)
(hash-table/get values front-pt 0)))))
front)
      ;;(fold-table-values! marks (lambda (val result) result val) #t)
      ;;(fold-table-values! values + 0)
      (if (null? front)
  (let* ((p-name-table-lst (get (domain.p-table domain) p-name))
 (updated-p-table (consolidate-hash-tbl
   (if p-name-table-lst
       (car p-name-table-lst)
       (make-empty-table equal?)) ; point=?
   values
   (lambda (v1-lst v2)
     (combiner (car v1-lst) v2)))))
    (replace (domain.p-table domain)
     p-name
     updated-p-table))
  (loop
   (fold-right (lambda (point+val result)
 (let ((point (car point+val))
       (val (cdr point+val)))
   (hash-table/put! marks point #t)
   (fold-right (lambda (nbr final-result)
 (add final-result nbr
      (* alpha val)))
       result
       (get-nbd point 1 empty-point-table))))
       '()
       (g-filter (lambda (point+val)
   (and (not (hash-table/get
      marks
      (car point+val)
      #f))
(> (cdr point+val) p-threshold)))
 front)))))))

(define (path:record-secretion2 domain p-name point ext combiner)
  (let ((get-nbd (domain.nbd-getter domain))
(values-tbl (make-equal-hash-table))
(p-threshold 1)
(alpha (/ 1. AVG-NBD-SIZE)))

    (define (expand-secretion extent value prev-front current-front)
      ;; for each element p in current-front:
      ;;   add p to interior
      ;;   add neighbours of p not in interior to front
      (if (or (= extent 0)
      (null? current-front)
      (<= value p-threshold))
  (update-ph-table)
  (expand-secretion (- extent 1)
    (* alpha value)
```

```
    current-front
    (update-front current-front
  value
  prev-front))))

    (define (update-front front current-value prev-front)
      ;; update values for neighbourhood of front
      ;; return union of neighbourhood of front
      (fold-right (lambda (front-point result)
    (expand-point front-point current-value
  prev-front front result))
  '()
  front))

    (define (expand-point point current-value prev current partial-next)
      ;; prev current and partial-next are all fronts.
      ;; returns updated partial-next
      (fold-right (lambda (nbr new-front)
    (hash-table/put! values-tbl nbr
     (+ current-value
(hash-table/get values-tbl nbr 0)))
    (if (or (memv nbr new-front)
    (memv nbr prev)
    (memv nbr current))
new-front
(cons nbr new-front)))
  partial-next
  (get-nbd point 1 empty-point-table)))

    (define (update-ph-table)
      (let* ((p-name-table-lst (get (domain.p-table domain) p-name))
     (updated-p-table (consolidate-hash-tbl
       (if p-name-table-lst
   (car p-name-table-lst)
   (make-empty-table equal?)) ; point=?
       values-tbl
       (lambda (v1-lst v2)
 (combiner (car v1-lst) v2)))))
(replace (domain.p-table domain)
 p-name
 updated-p-table)))

    (if (>= ext 0)
(let ((init-value (expt AVG-NBD-SIZE ext)))
  (hash-table/put! values-tbl point init-value)
  (expand-secretion ext init-value '() (list point)))
(error "Negative ranged secretions are not allowed" p-name ext))))

(define (binomial n k)
  (define (loop n k)
    (if (= k 0)
1
(* (/ (- n k -1) k)
   (loop n (- k 1)))))
  (cond ((> n k (/ n 2)) (binomial n (- n k)))
((< k 0) (error "BINOMIAL called with illegal second argument" k))
(else (loop n k))))

(define (path:record-secretion3 domain p-name point ext combiner)
```

```
  ;; this currently is wrong!!
  ;; there needs to be a sensible update rule for values.
  ;; the current value serves as a convenient representation of the path count
  ;; but it needs to be normalised, which involves making a "standard" scale
  ;; factor alpha0, which I incorrectly determined here (based on the
  ;; secretion result).

  (let ((get-nbd (domain.nbd-getter domain))
(values-tbl (make-eqv-hash-table))
(front-tbl (make-eqv-hash-table))
(p-threshold 1.)
(beta (- (/ (sqrt 2) (- (sqrt 2) 1)))))

    (define (alpha-n n count)
      (expt count (/ beta n)))

    (define (increment-value hash-tbl key amount)
      (hash-table/put! hash-tbl key
       (+ amount
  (hash-table/get hash-tbl key 0))))

    (let ((alpha0 (alpha-n ext
   (binomial ext (quotient ext 2)))))

      (define (expand-secretion prev-front current-front n front-val)
(if (null? current-front)
    (update-ph-table)
    (expand-secretion current-front
      (update-front prev-front
    current-front
    n
    front-val)
      (+ n 1)
      (* alpha0 front-val))))

      (define (update-front prev-front front n front-val)
;; update values for neighbourhood of front
;; return neighbourhood of front not already processed [ie next-front]
(let loop ((rest front)
   (values (empty-front-table! front)) ; modifies hash tables
   (new-front '()))
  (if (null? rest)
      new-front
      (let ((point (car rest))
    (point-value (car values)))
(let ((next-value (* (alpha-n n (/ point-value front-val))
     point-value)))
  (if (> point-value p-threshold)
      (loop (cdr rest) (cdr values)
    (expand-point point point-value
  prev-front front new-front))
      (loop (cdr rest) (cdr values)
    new-front)))))))

      (define (empty-front-table! front)
;; extract values for front in front-tbl, update values-tbl,
;; clear front-tbl.  return the values extracted.
(let ((front-values
       (map (lambda (point)
```

```
      (hash-table/lookup
       front-tbl point
       (lambda (v) v)
       (lambda ()
 ;; should never get here
 (error "RECORD-SECRETION: No value for"
point))))
    front)))
  (hash-table/for-each
   front-tbl
   (lambda (point value)
     (increment-value values-tbl point value)))
  (hash-table/clear! front-tbl)
  front-values))

      (define (expand-point point current-value prev current partial-next)
;; prev current and partial-next are all fronts.
;; returns updated partial-next
(fold-right (lambda (nbr new-front)
      (cond ((memv nbr new-front)
     (increment-value front-tbl nbr current-value)
     new-front)
    ((or (memv nbr prev)
 (memv nbr current))
     (increment-value values-tbl nbr current-value)
     new-front)
    (else
     (increment-value front-tbl nbr current-value)
     (cons nbr new-front))))
    partial-next
    (get-nbd point 1 empty-point-table)))

      (define (update-ph-table)
(let* ((p-name-table-lst (get (domain.p-table domain) p-name))
       (updated-p-table (consolidate-hash-tbl
 (if p-name-table-lst
     (car p-name-table-lst)
     (make-empty-table equal?)) ; point=?
 values-tbl
 (lambda (v1-lst v2)
   (combiner (car v1-lst) v2)))))
  (replace (domain.p-table domain)
   p-name
   updated-p-table)))

      (if (>= ext 0)
  (let ((init-value (expt alpha0 (- ext))))
    (hash-table/put! front-tbl point init-value)
    (expand-secretion '() (list point) 1 init-value))
  (error "Negative ranged secretions are not allowed" p-name ext)))))

(define (path:get-pheromone-value domain p-name point)
  ;; returns concentration of p-name at pt
  (let ((pt-table-lst (get (domain.p-table domain) p-name)))
    (and pt-table-lst
 (let ((val (get (car pt-table-lst) point)))
   (and val (car val))))))

(define (path:get-pheromone-value2 domain p-name point)
```

```
  ;; returns log of concentration of p-name at pt
  (let ((pt-table-lst (get (domain.p-table domain) p-name)))
    (and pt-table-lst
 (let ((val (get (car pt-table-lst) point)))
   (and val (log (car val)))))))

;;; Default secretion method is path, but to speed things up, can change to
;;; euclidean

(define D:record-secretion path:record-secretion2)
(define d:get-pheromone-value path:get-pheromone-value2)

(define (use-euclidean-secretions)
  (set! D:record-secretion euclidean:record-secretion)
  (set! D:get-pheromone-value euclidean:get-pheromone-value))

(define (use-path-secretions)
  (set! D:record-secretion path:record-secretion2)
  (set! D:get-pheromone-value path:get-pheromone-value2))

(define (D:get-values domain targets point)
  (vector-map
   targets
   (lambda (p-name)
     (D:get-pheromone-value domain p-name point))))

(define (get-nbd-vals domain lookahead targets point poison-pheromones)
  ;; first get a list of the non-poisoned neighbours
  (let ((get-nbd (domain.nbd-getter domain)))
    (let ((nbd (g-filter
(lambda (pt)
  (let ((poison-levels
 (map (lambda (poison)
(or (D:get-pheromone-value domain poison pt)
    0))
      poison-pheromones)))
    (for-all? poison-levels (lambda (level)
      (< level pheromone-threshold)))))
(get-nbd point 1 empty-point-table))))

      (define (expand-nbd nbr lookahead)
(let loop ((nbrs (list nbr)) (count lookahead))
  (if (= count 0)
      nbrs
      (loop (g-filter (lambda (nbr*)
(not (or (member nbr* nbd)
 (member nbr* nbrs))))
      (flat-map (lambda (nbr*)
  (get-nbd nbr* 1 empty-point-table))
nbrs))
    (- count 1)))))
      ;; this is not correct for lookaheads > 1 because it uses DFS.
      ;; correct solution needs a BFS tagging of points.
      (permute
       (flat-map (lambda (nbr)
   (map (lambda (nbr*)
  (make-entry nbr (D:get-values domain targets nbr*)))
(expand-nbd nbr lookahead)))
 nbd)))))
```

```
(define (D:get-nbd-vals-with-lookahead domain targets point poison-pheromones)
  (get-nbd-vals domain 1 targets point poison-pheromones))

(define (D:get-nbd-vals-no-lookahead domain targets point poison-pheromones)
  (get-nbd-vals domain 0 targets point poison-pheromones))

;; default is no lookahead
(define D:get-nbd-vals D:get-nbd-vals-no-lookahead)

(define (set-propagate-lookahead! val)
  (if (= val 0)
      (set! D:get-nbd-vals D:get-nbd-vals-no-lookahead)
      (set! D:get-nbd-vals D:get-nbd-vals-with-lookahead)))

(define (D:first-point domain filtrate)
  domain ;ignored for discrete domains
  (entry/tag (car filtrate)))

(define (D:inject domain pt)
  ((domain.pt->point domain) pt))

(define (D:display-domain domain)
  (((domain.display-mgr domain) 'show)))

(define (D:update-point point colour)
  (%D:update-point (D:point->pt point) colour))

(define-integrable (update-point domain point colour)
  (((domain.display-mgr domain) 'update) ((domain.point->pt domain) point)
 colour)
  ;;(D:display-point domain point colour)
  )

(define (D:display-point domain point colour)
  (update-point domain point colour)
  (D:display-domain domain))

(define (show-points domain marker points)
  (let ((mark-1-point (marker (domain.win domain)))
(point->pt (domain.point->pt domain)))
    (for-each (lambda (point)
(let ((pt (point->pt point)))
  (mark-1-point (pt.x pt) (pt.y pt))))
      points)))

(define (display-point-no-update domain point colour)
  (let ((w (domain.win domain))
(pt ((domain.point->pt domain) point)))
    (graphics-operation w 'set-foreground-color colour)
    (graphics-operation w 'fill-circle (pt.x pt) (pt.y pt)
(domain.pt-size domain))))

(define (show-pheromone domain p-name #!optional background)
  (let ((max-level 0)
(levels '())
(background (if (default-object? background) "blue" background)))
    (iter-points domain
 (lambda (point)
```

```
   (let ((level (D:get-pheromone-value domain p-name point)))
     (if (and level (> level max-level))
 (set! max-level level))
     (set! levels
   (cons (cons point level) levels)))))
    (let ((log-max-level (log max-level)))
      (reset-display domain background)
      (for-each (lambda (point+level)
  (let ((point (car point+level))
(level (cdr point+level)))
    (update-point
     domain point
     (cond ((not level) "wheat")
   ((= level 0) "blue")
   (else (number->grey level 0
       max-level))))))
levels)
      (D:display-domain domain))))

(define (D:draw-line domain pt1 pt2 colour)
  (let ((w (domain.win domain))
(pt1 ((domain.point->pt domain) pt1))
(pt2 ((domain.point->pt domain) pt2)))
    (graphics-operation w 'set-foreground-color colour)
    (graphics-draw-line w (pt.x pt1) (pt.y pt1)
(pt.x pt2) (pt.y pt2))))

;;;; Constructing Domains
;;; A discrete domain is a data structure containing the step-size,
;;; a procedure for obtaining a list of the neighbours of a point,
;;; a procedure for enumerating the points in the domain,
;;; a pheromone table of associations with points in the domain
;;; the display object and some information required by it.

(define-structure (domain (conc-name domain.)
  (constructor %make-domain
       (type ds size nbd-getter pt-ref
     point->pt pt->point
     win display-mgr pt-size
     p-table)))
  (type) ; arbitrary points, int-lattice etc
  (ds 5.) ; step-size
  (size) ; number of particles
  (nbd-getter) ; proc: point -> list of points
  (pt-ref) ; index -> point
  (point->pt) ; point -> coordinates
  (pt->point) ; coordinates -> point
  (win) ; display window
  (display-mgr) ; maintainer of window image
  (pt-size) ; particle size in window coordinates
  (p-table) ; pheromone table
  )

(define (make-domain type ds size nbd-getter pt-ref
     point->pt pt->point window display-mgr pt-size)
  (let ((d (%make-domain type ds size nbd-getter pt-ref
     point->pt pt->point window display-mgr pt-size
     (make-empty-table eq?))))
    d))
```

```
(define (win->domain window ds)
  (with-values
      (lambda () (graphics-device-coordinate-limits window))
    (lambda (xlo ylo xhi yhi)
      (let ((max-x (abs (- xhi xlo)))
    (max-y (abs (- yhi ylo))))
(graphics-set-coordinate-limits window 0 0 max-x max-y)
(let ((width (+ max-x 1))
      (height (+ max-y 1)))
  (make-domain 'discrete-int-lattice
       ds
       (* width height)
       (make-int-lattice-nbd-getter width height ds)
       (make-int-lattice-ref width height)
       (lambda (point) point)
       (lambda (pt) (make-pt (inexact->exact (pt.x pt))
     (inexact->exact (pt.y pt))))
       window
       (make-display-mgr window 0)
       1))))))

(define (layout->domain layout ds window)
  (let ((xs (vector-ref layout 0))
(ys (vector-ref layout 1))
(zs (vector-ref layout 2)))

    (let ((nbd-vec (point-neighbours xs ys zs (exact->inexact ds))))
      (let ((result (make-domain
     'discrete-layout
     ds
     (vector-length xs)
     (nbd-vec->getter2 nbd-vec)
     (lambda (index) index) ; index is point
     (lambda (point)
       (make-pt (vector-ref xs point)
(vector-ref ys point)))
     (lambda (pt) (find-point-nearest pt xs ys))
     #f
     #f ; will be set to dummy
     (min .5 pt-size))))

(define (dummy-display-mgr msg)
  ;; install the real display-mgr
  (2d-points->win xs ys window (finish-domain result))
  ((domain.display-mgr result) msg) ; service msg
  )

(set-domain.display-mgr! result dummy-display-mgr)
result))))

(define (finish-domain domain)
  (lambda (win pix->coords)
    (let ((pt-size (* pix->coords pt-pixel-size)))
      (let ((display-mgr (make-display-mgr
  win
  (if (> pt-size .5)
      (/ .5 pix->coords)
      pt-pixel-size))))
```

```
(set-domain.win! domain win)
(set-domain.display-mgr! domain display-mgr)))))

(define (2d-points->win xs ys window k)
  ;; k = (lambda (win pixel-scale) ...)
  (with-bounding-box
   xs ys
   (lambda (xlo ylo xhi yhi)
     (let* ((width (abs (- xhi xlo)))
    (height (abs (- yhi ylo)))
    (2margin-pix (* 2 margin-pix)))
       (if window
   (with-values
       (lambda () (graphics-device-coordinate-limits window))
     (lambda (dev-xlo dev-ylo dev-xhi dev-yhi)
       (let ((dev-width (+ (- dev-xhi dev-xlo) 1))
     (dev-height (+ (- dev-yhi dev-ylo) 1)))
 (let ((pix->coords
(max
 (/ height (- dev-height 2margin-pix))
 (/ width (- dev-width 2margin-pix)))))
   (let ((margin (* pix->coords margin-pix)))
     (graphics-set-coordinate-limits
      window
      (- xlo margin) (- ylo margin)
      (+ xhi margin) (+ yhi margin))
     (k window pix->coords))))))
   (let ((corrected-window-size (- preferred-window-size
   2margin-pix)))
     (if (>= width height)
 (let ((pix->coords (/ height corrected-window-size)))
   (let ((margin (* pix->coords margin-pix)))
     (k (make-window (ceiling->exact
      (* preferred-window-size
 (/ width height)))
     preferred-window-size
     `(,(- xlo margin)
       ,(- ylo margin)
       ,(+ xhi margin)
       ,(+ yhi margin)))
pix->coords)))
 (let ((pix->coords (/ width corrected-window-size)))
   (let ((margin (* pix->coords margin-pix)))
     (k (make-window preferred-window-size
     (ceiling->exact
      (* preferred-window-size
 (/ height width)))
     `(,(- xlo margin)
       ,(- ylo margin)
       ,(+ xhi margin)
       ,(+ yhi margin)))
pix->coords)))))))))

(define (reset-display domain background)
  ;; background is any valid X colour string
  (((domain.display-mgr domain) 'reset) background))

(define (init-display domain colour)
  ;; in a discrete-int-lattice every pixel is a particle
```

```
  (if (eq? (domain.type domain) 'discrete-int-lattice)
      (reset-display domain colour)
      (let ((display-updater ((domain.display-mgr domain) 'update))
    (point->pt (domain.point->pt domain)))
(iter-points
 domain
 (lambda (point)
   (display-updater (point->pt point) colour))))))

(define (find-point-nearest pt xs ys)
  (let ((pt-x (pt.x pt))
(pt-y (pt.y pt))
(n (vector-length xs)))
    (let loop ((index 0)
       (nearest 0)
       (min-dist (L2norm pt-x pt-y
 (vector-ref xs 0) (vector-ref ys 0))))
      (if (= index n)
  nearest
  (let ((dist (L2norm pt-x pt-y
      (vector-ref xs index) (vector-ref ys index))))
    (if (< dist min-dist)
(loop (+ index 1) index dist)
(loop (+ index 1) nearest min-dist)))))))

(define (with-bounding-box xs ys k)
  (find-min&max
   xs
   (lambda (xlo xhi)
     (find-min&max
      ys
      (lambda (ylo yhi)
(k xlo ylo xhi yhi))))))

(define (make-display-mgr window particle-size)
  (with-values
      (lambda () (graphics-device-coordinate-limits window))
    (lambda (hlo vlo hhi vhi)
      (let ((dev-width (+ (abs (- hhi hlo)) 1))
    (dev-height (+ (abs (- vhi vlo)) 1)))
(let ((x-image (image/descriptor
(graphics-operation window 'create-image
    (+ dev-width 1) (+ dev-height 1))))
      (colour-map (graphics-operation window 'get-colormap))
      (coords-cache (fill-circle-coords particle-size)))

  (define (reset-image background)
    (let ((col (colour->index colour-map background)))
      (for-each-index
       0 (- dev-width 1) 1
       (lambda (x)
 (for-each-index
  0 (- dev-height 1) 1
  (lambda (y)
    (x-image/set-pixel x-image x y col)))))))

  (with-values
      (lambda () (graphics-coordinate-limits window))
    (lambda (xlo ylo xhi yhi)
```

```
      (let ((coord-width (- xhi xlo))
    (coord-height (- yhi ylo)))
;; initialize image to grey background
(reset-image "#CCCCCC")
(lambda (msg)
  (case msg
    ((UPDATE)
     ;; convert point to image coordinates
     ;;  (y-coordinate needs reflection)
     ;; convert colour to rgb values, then get colormap index
     (lambda (pt colour)
       (let ((img-x (floor->exact (* (/ (- (pt.x pt) xlo)
coord-width)
     dev-width)))
     (img-y (floor->exact
     (* (- 1 (/ (- (pt.y pt) ylo)
coord-height))
dev-height))))
 (for-each
  (lambda (xy)
    (x-image/set-pixel x-image
       (+ (car xy) img-x)
       (+ (cdr xy) img-y)
       (colour->index colour-map
      colour)))
  coords-cache))))
    ((SHOW)
     (lambda ()
       (x-image/draw x-image xlo yhi)))
    ((RESET)
     (lambda (background)
       (newline)
       (display ";Resetting Display...")
       (newline)
       (reset-image background)))
    (else
     (error "Unrecognised Domain Display Manager Message"
    msg))))))))))))

(define colour->index
  (let ((index-table '()))
    (lambda (colourmap colour)
      (let ((rgb (x-color->rgb colour)))
(let ((index-entry (assoc rgb index-table)))
  (if index-entry
      (cadr index-entry)
      (let ((index (apply x-colormap/allocate-color colourmap
  (->colormap-scale rgb))))
(if (not index)
    (error "COLOUR->INDEX: Could not allocate new color"
   colour)
    (begin
      (set! index-table (cons (list rgb index) index-table))
      index)))))))))

(define (->colormap-scale rgb-list)
  (map (lambda (8bit-level)
 (fix:* 8bit-level 256))
       rgb-list))
```

```
(define (make-int-lattice-nbd-getter width height ds)
  (lambda (point nbd-radius exclusions)
    ;; returns a list of points not in exclusions, but in neighbourhood
    ;; centred at point, with radius in ds units
    (let ((pt-x (pt.x point))
  (pt-y (pt.y point))
  (r (* nbd-radius ds)))
      (let ((hlo (max 0 (- pt-x r)))
    (hhi (min (- width 1) (+ pt-x r)))
    (vlo (max 0 (- pt-y r)))
    (vhi (min (- height 1) (+ pt-y r))))
(let v-loop ((v vlo) (result '()))
  (if (> v vhi)
      result
      (let h-loop ((u hlo) (h-result result))
(cond ((> u hhi)
       (v-loop (+ v 1) h-result))
      ((> (L2norm u v pt-x pt-y) r)
       (h-loop (+ u 1) h-result))
      ((and (= u pt-x) (= v pt-y))
       (h-loop (+ u 1) h-result))
      ((get exclusions (make-pt u v))
       (h-loop (+ u 1) h-result))
      (else
       (h-loop (+ u 1) (cons (make-pt u v) h-result))))))))))))

(define (nbd-vec->getter nbd-vec)
  (define (get-nearest-nbrs point nbd-radius)
    ;; points are indexes.
    (let ((pt-nbd (vector->list (vector-ref nbd-vec point))))
      (if (= nbd-radius 1)
  pt-nbd
  (fold-right
   union '()
   (map (lambda (nbr)
  (delete point (get-nearest-nbrs nbr (- nbd-radius 1))))
pt-nbd)))))

  (lambda (point nbd-radius exclusions)
    ;; ignore exclusions, present only for backwards compatibility
    exclusions
    (if (>= nbd-radius 1)
(get-nearest-nbrs point nbd-radius)
'())))

(define (nbd-vec->getter2 nbd-vec)
  (define (get-nearest-nbrs nbd-radius front interior)
    ;; for each element p in front:
    ;;   add p to interior
    ;;   add neighbours of p not in interior to front
    (cond ((= nbd-radius 0)
   (set+ front interior))
  ((null? front)
   interior)
  (else
   (let ((new-interior (set+ front interior))
 (new-front (fold-right
     (lambda (front-pt result)
```

```
      (set+ (vector->set
     (vector-ref nbd-vec front-pt))
    result))
    empty-set
    front)))
    (get-nearest-nbrs (- nbd-radius 1)
      (set- new-front new-interior)
      new-interior)))))

  (lambda (point nbd-radius exclusions)
    ;; points are indexes.
    ;; ignore exclusions, present only for backwards compatibility
    exclusions
    (if (>= nbd-radius 0)
(set->list (set- (get-nearest-nbrs nbd-radius (elt->set point)
   empty-set)
 (elt->set point)))
'())))

(define (make-int-lattice-ref width height)
  (lambda (i)
    (make-pt (modulo i width)
     (modulo (quotient i width) height))))

(define (make-pt-ref xs ys)
  (lambda (i)
    (make-pt (vector-ref xs i) (vector-ref ys i))))

(define (iter-points domain proc)
  ;; used to enumerate all the points in terms of their coordinates
  (let ((pt-ref (domain.pt-ref domain)))
    (for-each-index 0 (- (domain.size domain) 1) 1
    (lambda (index)
      (proc (pt-ref index))))))

(define (L2norm x1 y1 x2 y2)
  (let ((sq (lambda (x) (* x x))))
    (sqrt (+ (sq (- x2 x1)) (sq (- y2 y1))))))

(define (Euclidean-metric p-pt q-pt)
  ;; assume p-pt and q-pt are coordinates
  (L2norm (pt.x p-pt) (pt.y p-pt) (pt.x q-pt) (pt.y q-pt)))

;; this needs testing
(define (shortest-path-metric p q)
  ;; use variant of Djikstra's algorithm to find shortest path from p to q
  ;; problem is simpler because all weights are equal to 1.
  ;; optimize by using physical coordinates as pruning indicator of progress.
  (let ((p-pt (D:point->pt p))
(q-pt (D:point->pt q)))
    (let ((p.x (pt.x p-pt))
  (p.y (pt.y p-pt))
  (q.x (pt.x q-pt))
  (q.y (pt.y q-pt)))
      (let loop ((nbrs (list p)) (dist 0))
(if (member q nbrs)
    dist
    (loop (fold-right (lambda (pt result)
(union (D:get-nbd pt 1 empty-point-table)
```

```
       result))
      '()
      (g-filter
       (lambda (nbr)
 (let ((nbr-pt (D:point->pt nbr)))
   (or (between? p.x q.x (pt.x nbr-pt))
       (between? p.y q.y (pt.y nbr-pt)))))
       nbrs))
  (+ dist 1)))))))

(define (D:Euclidean-metric domain p q)
  ;; default metric is Euclidean
  (Euclidean-metric ((domain.point->pt domain) p)
    ((domain.point->pt domain) q)))

(define (square x) (* x x))


(define (phi ext)
  (let* ((r0 1e-7) ; r0 is in hops
 (alpha -.5))
    (lambda (r*)
      (let ((r (/ r* ds)))
(cond ((> r ext) 0)
      ((< r r0) (* alpha (log (/ r0 ext))))
      (else (* alpha (log (/ r ext))))))))))

;;;; Statistics on domains without having to store all raw data at once.

(define (nbd-mean domain)
  (find-mean (make-nbd-size-producer domain)))

(define (nbd-variance domain)
  (find-variance (make-nbd-size-producer domain)))

(define (sep-mean domain)
  (find-sep-mean  (make-nbd-pt-producer domain)))

(define (sep-variance domain)
  (find-sep-variance (make-nbd-pt-producer domain)))

;;; A Bunch of producers
(define (make-nbd-producer domain)
  (let ((get-nbd (domain.nbd-getter domain))
(last (- (domain.size domain) 1)))
    (lambda (index consumer)
      (if (<= 0 index last)
  (consumer index (get-nbd index 1 empty-point-table))
  (consumer index 'stop)))))

(define (make-nbd-size-producer domain)
  (let ((nbd-producer (make-nbd-producer domain)))
    (lambda (index consumer)
      (nbd-producer index
    (lambda (next nbd)
      (if (eq? nbd 'stop)
  (consumer next 'stop)
  (consumer next (length nbd))))))))
```

```
(define (make-nbd-pt-producer domain)
  (let ((get-nbd (domain.nbd-getter domain))
(point->pt (domain.point->pt domain))
(last (- (domain.size domain) 1)))
    (lambda (index consumer)
      (if (<= 0 index last)
  (consumer index
    (point->pt index)
    (map point->pt (get-nbd index 1 empty-point-table)))
  (consumer index
    (point->pt index)
    'stop)))))

;;; These consumers do the real work

(define (find-mean producer)
  (define (tally-one n sum)
    (lambda (index x)
      (if (eq? x 'stop)
  (if (< n 1)
      (list n sum)
      (exact->inexact (/ sum n)))
  (producer (+ index 1)
    (tally-one (+ n 1) (+ sum x))))))

  (producer 0 (tally-one 0 0)))

(define (find-variance producer)
  (define (tally-one n sum sq-sum)
    (lambda (index x)
      (if (eq? x 'stop)
  (if (< n 1)
      (list n sum sq-sum)
      (let ((mean (/ sum n)))
(exact->inexact (- (/ sq-sum n) (* mean mean)))))
  (producer (+ index 1)
    (tally-one (+ n 1) (+ sum x) (+ sq-sum (* x x)))))))

  (producer 0 (tally-one 0 0 0)))

;; There is probably a way to neatly capture the common patterns...
;; but no time!

(define (find-sep-mean producer)
  (define (tally-lst n sum)
    (lambda (index index-pt lst)
      (if (eq? lst 'stop)
  (if (< n 1)
      (list n sum)
      (exact->inexact (/ sum n)))
  (let loop ((rest lst) (n* n) (sum* sum))
    (if (null? rest)
(producer (+ index 1) (tally-lst n* sum*))
(let ((x (euclidean-metric index-pt (car rest))))
  (loop (cdr rest) (+ n* 1) (+ sum* x))))))))
  (producer 0 (tally-lst 0 0)))

(define (find-sep-variance producer)
  (define (tally-lst n sum sq-sum)
```

```
    (lambda (index index-pt lst)
      (if (eq? lst 'stop)
  (if (< n 1)
      (list n sum sq-sum)
      (let ((mean (/ sum n)))
(exact->inexact (- (/ sq-sum n) (* mean mean)))))
  (let loop ((rest lst) (n* n) (sum* sum) (sq-sum* sq-sum))
    (if (null? rest)
(producer (+ index 1) (tally-lst n* sum* sq-sum*))
(let ((x (euclidean-metric index-pt (car rest))))
  (loop (cdr rest) (+ n* 1)
(+ sum* x) (+ sq-sum* (* x x)))))))))
  (producer 0 (tally-lst 0 0 0)))

;;; Load syncsim/tools to use these plotting utilities

(define (D:plot-f domain src radius f width height)
  (let ((point->pt (domain.point->pt domain)))
    (let ((nbrs ((domain.nbd-getter domain) src radius '()))
  (pt (point->pt src)))
      (plot (list->vector
     (map (lambda (nbr) (Euclidean-metric (point->pt nbr) pt))
  nbrs))
    (list->vector (map f nbrs))
    (mark-x 3)
    width height))))

(define (D:plot-levels+logs domain src radius p-name width height)
  (let ((point->pt (domain.point->pt domain)))
    (let ((nbrs ((domain.nbd-getter domain) src radius '()))
  (pt (point->pt src)))
      (let ((data (fold-right
   (lambda (nbr result-lsts)
     (let ((val (D:get-pheromone-value domain p-name nbr)))
       (if val
   (list (cons (Euclidean-metric (point->pt nbr) pt)
       (car result-lsts))
 (cons val (cadr result-lsts)))
   result-lsts)))
   (list '() '())
   nbrs)))
(let ((distances (list->vector (car data)))
      (values (list->vector (cadr data))))
  ;; first plot the secretion levels
  (plot distances values (mark-x 3) width height)
  ;; then plot the secretions on a log scale
  (plot distances (vector-map values log) (mark-x 3) width height))))))

(define (D:plot-log-levels domain src radius p-name width height)
  (D:plot-f domain src radius
    (lambda (nbr)
      (let ((val (D:get-pheromone-value domain p-name nbr)))
(if val
    (log val)
    0)))
    width height))

(define (D:plot-levels domain src radius p-name width height)
  (D:plot-f domain src radius
```

```
    (lambda (nbr)
      (or (D:get-pheromone-value domain p-name nbr) 0))
    width height))

;;;; Implementation of pt. (These should have been called coordinates)

(define make-pt cons)
(define (pt.x pt) (car pt))
(define (pt.y pt) (cdr pt))

(define (pt? pt)
  (and (pair? pt)
       (number? (pt.x pt))
       (number? (pt.y pt))))

(define pt:= equal?)

(define (pt:< pt1 pt2)
  (or (< (pt.x pt1) (pt.x pt2))
      (and (= (pt.x pt1) (pt.x pt2))
   (< (pt.y pt1) (pt.y pt2)))))

;;; Dummy table to indicate no exclusions for D:get-nbd
(define empty-point-table (make-empty-table pt:=))

;;;; Pheromone tuples

(define-integrable make-tuple vector)
(define-integrable (tuple.pt tuple) (vector-ref tuple 0))
(define-integrable (tuple.f tuple) (vector-ref tuple 1))
(define-integrable (tuple.combiner tuple) (vector-ref tuple 2))

;;;; Trashed code

(define (layout->domain2 layout ds window)
  (let ((xs (vector-ref layout 0))
(ys (vector-ref layout 1))
(zs (vector-ref layout 2)))

    (define (construct-domain win pix->coords)
      (let ((nbd-vec (point-neighbours xs ys zs (exact->inexact ds)))
    (pt-size (* pix->coords pt-pixel-size)))
(let ((display-mgr (make-display-mgr
    win
    (if (> pt-size .5)
(/ .5 pix->coords)
pt-pixel-size))))
  (let ((result (make-domain
 'discrete-layout
 ds
 (vector-length xs)
 (nbd-vec->getter2 nbd-vec)
 (lambda (index) index) ; index is point
 (lambda (point)
   (make-pt (vector-ref xs point)
    (vector-ref ys point)))
 (lambda (pt) (find-point-nearest pt xs ys))
 win
 display-mgr
```

```
 (min .5 pt-size))))
    ;;(init-display result "wheat")
    ;;(D:display-domain result)
    result))))

    (with-bounding-box
     xs ys
     (lambda (xlo ylo xhi yhi)
       (let* ((width (abs (- xhi xlo)))
      (height (abs (- yhi ylo)))
      (2margin-pix (* 2 margin-pix)))
 (if window
     (with-values
 (lambda () (graphics-device-coordinate-limits window))
       (lambda (dev-xlo dev-ylo dev-xhi dev-yhi)
 (let ((dev-width (+ (- dev-xhi dev-xlo) 1))
       (dev-height (+ (- dev-yhi dev-ylo) 1)))
   (let ((pix->coords
  (max
   (/ height (- dev-height 2margin-pix))
   (/ width (- dev-width 2margin-pix)))))
     (let ((margin (* pix->coords margin-pix)))
       (graphics-set-coordinate-limits
window
(- xlo margin) (- ylo margin)
(+ xhi margin) (+ yhi margin))
       (construct-domain window pix->coords))))))
     (let ((corrected-window-size (- preferred-window-size
     2margin-pix)))
       (if (>= width height)
   (let ((pix->coords (/ height corrected-window-size)))
     (let ((margin (* pix->coords margin-pix)))
       (construct-domain
(make-window (ceiling->exact
      (* preferred-window-size
 (/ width height)))
     preferred-window-size
     `(,(- xlo margin)
       ,(- ylo margin)
       ,(+ xhi margin)
       ,(+ yhi margin)))
pix->coords)))
   (let ((pix->coords (/ width corrected-window-size)))
     (let ((margin (* pix->coords margin-pix)))
       (construct-domain
(make-window preferred-window-size
     (ceiling->exact
      (* preferred-window-size
 (/ height width)))
     `(,(- xlo margin)
       ,(- ylo margin)
       ,(+ xhi margin)
       ,(+ yhi margin)))
pix->coords))))))))))
```

**layouts.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

;;; Routines for generating point layouts.  Extracted from
;;;  /zu/newts/pc-home/gunk/hlsim2/extensions.scm

#|

This file (along with those it loads) provides procedures for defining
particle distributions.  In all cases the result returned is a vector
of three vectors: the x,y and z coordinates in that order.

For example, if L is the result of one of these procedures, then
  (list (vector-ref (vector-ref L 0) 0)
(vector-ref (vector-ref L 1) 0)
(vector-ref (vector-ref L 2) 0))

is a list of the (x,y,z) coordinates of the first point.

Procedures provided:

(make-random-points N WIDTH HEIGHT)
 returns a distribution with exactly N points.  Each point is assigned random
 coordinates uniformly from (0,0,0) to (WIDTH, HEIGHT, 0).
 Note that if WIDTH and HEIGHT are given as exact numbers, then the coordinates
 will be restricted to integers.

(make-non-overlapping-pts N WIDTH HEIGHT [DEPTH])
 Assigns as many points (not more than N) uniformly random coordinates in the
 region (0, 0, 0) to (WIDTH, HEIGHT, DEPTH) without overlapping the particles.
 The particles are assumed to have a diameter of 1 unit (in the same units
 as the given bounds).  If DEPTH is not given, then the boundary of the region
 is (0, 0, 1) to (WIDTH, HEIGHT, 1).

(make-binomial-points N WIDTH HEIGHT [P])
 Returns a distribution of expected size N with coordinates in the range (0,0)
 to (WIDTH, HEIGHT).  The region is divided into enough squares so that when
 a particle is assigned to each square with probability P, the expected total
 number of particles is N.  The coordinates of a particle within a square are
 assigned randomly from a uniform distribution over the boundaries of the
 square. If P is not specified, it defaults to .8.  The lower the value of P
 the higher the variance in the density.

(make-rect-grid-points width height h-spacing v-spacing)

(make-triangle-grid-points width height dx dy)

(make-2d-hcp-points n spacing)

(show-pts WIN PTS)
 Displays the distribution contained in PTS in the graphics-device WIN

|#

(load "~newts/gunk/gpl-illustrator/placement")
```

```
;; defines (make-non-overlapping-pts N WIDTH HEIGHT !optional DEPTH)
;; and  (%make-non-overlapping-pts N WIDTH HEIGHT !optional DEPTH)

(define-integrable (make-coord x y z) (list x y z))
(define-integrable (xcor coord) (car coord))
(define-integrable (ycor coord) (cadr coord))
(define-integrable (zcor coord) (caddr coord))

;;;; Procedures for making and manipulating coordinate configurations.

(define (make-random-points n width height)
  ;; this is essentially a Poisson distribution
  (let ((x-fun (if (zero? width)
   (lambda (i) i 0.)
   (lambda (i) i (random width))))
(y-fun (if (zero? height)
   (lambda (i) i 0.)
   (lambda (i) i (random height))))
)
    (vector
     (make-initialized-vector n x-fun)
     (make-initialized-vector n y-fun)
     (make-vector n 0.))))

(define (make-2d-hcp-points n spacing)
  ;; hexagonally close packs n discs each of unit diameter.
  (let ((rt3/2 (/ (sqrt 3) 2)))
    (let ((xs (make-vector n 0.))
  (ys (make-vector n 0.))
  (dxs (vector -.5  -1.   -.5       .5      1.  .5))
  (dys (vector rt3/2 0. (- rt3/2) (- rt3/2) 0. rt3/2)))

    ;; first point is correctly placed, so start from 1 (second pt)
    (let loop ((i 1) (radius 1) (dir 0) (side-count 1) (x 1.) (y 0.))
      (cond ((>= i n)
     (vector (translate (scale xs spacing) radius)
     (translate (scale ys spacing) radius)
     (make-vector n 0.)))
    ((= dir 6)
     (loop i (+ radius 1) 0 1 (flo:+ x 1.) 0.))
    (else
     (let ((newx (flo:+ x (vector-ref dxs dir)))
   (newy (flo:+ y (vector-ref dys dir))))
       (vector-set! xs i x)
       (vector-set! ys i y)
       (if (= side-count radius)
   (loop (+ i 1) radius (+ dir 1) 1 newx newy)
   (loop (+ i 1) radius dir (+ side-count 1) newx newy)))))))))

(define (make-triangle-grid-points width height dx dy)
  ;; produces points on a regular triangular grid in a rectangle of
  ;; WIDTH x HEIGHT.  dx is base length, and dy is the height of a triangle
  (let ((m (floor->exact (/ width dx)))
(n (floor->exact (/ height (* 2 dy)))))
    (let ((size (+ (* 2 (* m n)) ; internal nodes
   (* 1/2 (* 2 (+ m n))) ; edge nodes
   (* 1/4 4)))) ; corner nodes
      (let ((xs (make-vector size 0.))
    (ys (make-vector size 0.))
```

```
    (dx/2 (/ dx 2.)))
(let i-loop ((i 0) (y 0.) (skew? #f))
  (if (fix:= i size)
      (vector xs ys (make-vector size 0.))
      (let j-loop ((j i) (x (if skew? dx/2 0.)))
(if (> x width)
    (i-loop j (+ y dy) (not skew?))
    (begin
      (vector-set! xs j x)
      (vector-set! ys j y)
      (j-loop (+ j 1) (+ x dx))))))))))

(define (make-rect-grid-points width height h-spacing v-spacing)
  ;;produces points on a rectangular grid in a rectangle of WIDTH x HEIGHT
  (let ((m (+ (floor->exact (/ width h-spacing)) 1))
(n (+ (floor->exact (/ height v-spacing)) 1)))
    (let* ((size (* m n))
   (xs (make-vector size 0.))
   (ys (make-vector size 0.)))
      (let i-loop ((i 0) (y 0.))
(if (fix:= i size)
    (vector xs ys (make-vector size 0.))
    (let j-loop ((j i) (x 0.))
      (if (> x width)
  (i-loop j (+ y v-spacing))
  (begin
    (vector-set! xs j x)
    (vector-set! ys j y)
    (j-loop (+ j 1) (+ x h-spacing)))))))))

(define (make-binomial-points n width height #!optional p)
  ;; p small --> poisson
  ;; larger p --> less particle overlap (default assumption)
  ;; use p > .7 for "nice" distributions
  ;; N   = total # of cells
  ;; A   = total area = width*height
  ;; da  = area of 1 cell = A/N
  ;; n   = Np (given)
  ;; lam = n/A = Np/A = p/da (for poisson process parameter)
  (let ((p (if (default-object? p) .8 p)))
    (let* ((N (ceiling->exact (/ n p)))
   (da (/ (* width height) N))
   (ds (sqrt da))
   (nx (ceiling->exact (/ width ds)))
   (ny (ceiling->exact (/ height ds))) ; N = nx * ny
   )
      (let ((xs (make-vector n 0.))
    (ys (make-vector n 0.))
    (zs (make-vector n 0.))) ; zs never modified
(let yloop ((y 0.) (results '()))
  (if (> y height)
      (coords->pts results)
      (let xloop ((x 0.) (results* results))
(if (> x width)
    (yloop (+ y ds) results*)
    (if (<= (random 1.) p)
(xloop (+ x ds)
       (cons (make-coord (+ x (random ds))
 (+ y (random ds))
```

```
 0.)
     results*))
(xloop (+ x ds) results*))))))))))

(define (coords->pts coords-lst)
  ;; coords are assumed to be in 3d
  (vector (list->vector (map xcor coords-lst))
  (list->vector (map ycor coords-lst))
  (list->vector (map zcor coords-lst))))

(define (translate v delta)
  ;; adds delta to each component of v
  (let loop ((i (- (vector-length v) 1)))
    (if (fix:< i 0)
v
(begin
  (vector-set! v i (+ (vector-ref v i) delta))
  (loop (fix:- i 1))))))

(define (scale v k)
  ;; scales each entry of v by k
  (let loop ((i (- (vector-length v) 1)))
    (if (fix:< i 0)
v
(begin
  (vector-set! v i (* (vector-ref v i) k))
  (loop (fix:- i 1))))))

(define (show-pts win pts)
  (let ((xs (vector-ref pts 0))
(ys (vector-ref pts 1))
(zs (vector-ref pts 2)))
    (graphics-operation win 'set-background-color "#CCCCCC")
    (graphics-operation win 'set-foreground-color "black")
    (graphics-clear win)
    (for-each-index 0 (- (vector-length xs) 1) 1
    (lambda (index)
      (graphics-operation win 'fill-circle
  (vector-ref xs index)
  (vector-ref ys index)
  .5)))))
```

**placement.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))
(declare (integrate-external "extensions"))

#|
Problem:
Given a number of cubes (processors), their size and the cuboid in
which they should be placed, assign coordinates to the
processors so that the cubes do not overlap each other.

Solution:
The algorithm works by maintaining the available space.  When a cube
is placed inside a cuboid, the remaining space in the cuboid can be
partitioned into 6 sub-cuboids.  To place a second cube in the
original cuboid, first choose one of the 6 sub-cuboids with
probability proportional to their volumes and then choose a random
coordinate within the chosen sub-cuboid.

Starting with an initial cuboid defined by the specified volume,
recursively apply that idea.  Bookkeeping needs to be done to maintain
the geometric relationships between cuboids and to aid in the random
selection of the cuboids with the correct associated probabilities.

I believe this algorithm is O(n lg n) where n is the number of
cubes. If it's any worse, it is not really worth the trouble.  One
advantage of it over any of the other methods so far, is that it should be
able to detect when there is no more space left.

|#

(define-structure (cuboid
   (conc-name cuboid.)
   (constructor %make-cuboid (xlo ylo zlo xhi yhi zhi))
   (print-procedure
    (standard-unparser-method
     'CUBOID
     (lambda (cuboid port)
       (write-char #\space port)
       (write (cuboid.remaining-volume cuboid) port)
       (write-char #\space port)
       (write
(list
 (list (cuboid.xlo cuboid)
       (cuboid.ylo cuboid)
       (cuboid.zlo cuboid))
 (list (cuboid.xhi cuboid)
       (cuboid.yhi cuboid)
       (cuboid.zhi cuboid)))
port)
       (write-char #\space port)
       (write (cuboid.split-pt cuboid) port)))))
  ;; all coordinates must be flonums
  (xlo 0.)
  (ylo 0.)
  (zlo 0.)
```

```
  (xhi 0.)
  (yhi 0.)
  (zhi 0.)
  (remaining-volume 0.) ; volume available for placing points.
  (split-pt #f) ; if split, coords of splitting pt.
  (parent #f) ; containing cuboid
  (fragments #f) ; if split, resulting sub-cuboids
  ;; fragment idexes assigned as follows:
  ;; |----+--+----------|
  ;; |    |  |          |
  ;; |    |1 |          |
  ;; |    +--+          |
  ;; | 0  |  |  2       |  4 below, 5 above
  ;; |    +--+          |
  ;; |    |  |          |
  ;; |    | 3|          |
  ;; |----+--+----------|
  )

(define (make-cuboid xlo ylo zlo xhi yhi zhi)
  (let ((cuboid (%make-cuboid xlo ylo zlo xhi yhi zhi)))
    (set-cuboid.remaining-volume! cuboid
  (* (- xhi xlo) (- yhi ylo) (- zhi zlo)))
    cuboid))

;;; Procedures to handle geometry of interaction of cube and cuboid
(define-integrable padded-cube-volume 8.)
(define-integrable cube-volume 1.)

(define (flo:<= x y)
  (or (flo:< x y) (flo:= x y)))

(define (volume xlo ylo zlo xhi yhi zhi)
  (flo:* (flo:- xhi xlo)
 (flo:* (flo:- yhi ylo)
(flo:- zhi zlo))))

(define (inside? cuboid xlo ylo zlo xhi yhi zhi)
  ;; return true if cube defined by coordinates is entirely inside cuboid.
  (if (not (cuboid.parent cuboid)) ; at top level (largest) cuboid
      #t
      (and (flo:<= (cuboid.xlo cuboid) xlo)
   (flo:<= xhi (cuboid.xhi cuboid))
   (flo:<= (cuboid.ylo cuboid) ylo)
   (flo:<= yhi (cuboid.yhi cuboid))
   (flo:<= (cuboid.zlo cuboid) zlo)
   (flo:<= zhi (cuboid.zhi cuboid)))))

(define (smallest-containing-cuboid cuboid xlo ylo zlo xhi yhi zhi)
  ;; assumes cuboid described intersects cuboid
  (if (inside? cuboid xlo ylo zlo xhi yhi zhi)
      cuboid
      (smallest-containing-cuboid (cuboid.parent cuboid)
  xlo ylo zlo xhi yhi zhi)))

(define (intersecting-fragments cuboid xlo ylo zlo xhi yhi zhi)
  ;; assumes cuboid described by coordinates lies entirely within CUBOID.
  ;; return a list of leaf cuboids that intersect cuboid described.
```

```
  (let ((fragments (cuboid.fragments cuboid)))

    (define (fragments-0-to-3 xlo ylo zlo xhi yhi zhi)
      ;; assumes that cuboid was split and
      ;; split-zlo <= zlo <= zhi <= split-zhi
      (let ((split-pt (cuboid.split-pt cuboid)))
(let ((split-xlo (max (flo:- (xcor split-pt) 1.) (cuboid.xlo cuboid)))
      (split-xhi (min (flo:+ (xcor split-pt) 1.) (cuboid.xhi cuboid))))
  (cond ((flo:< xhi split-xlo) ; is all in 0
 (intersecting-fragments (vector-ref fragments 0)
 xlo ylo zlo xhi yhi zhi))

((flo:< xlo split-xlo) ; part is in 0
 (append
  (intersecting-fragments (vector-ref fragments 0)
  xlo ylo zlo split-xlo yhi zhi)
  ;; and the other part is
  ;; can't assume only two overlap parts b/c of external splits
  (fragments-0-to-3 split-xlo ylo zlo xhi yhi zhi)))

((flo:<= xhi split-xhi) ; entirely within 1 or 3
 (fragment-1-or-3 xlo ylo zlo xhi yhi zhi))

((flo:< xlo split-xhi) ; part in 2
 (append
  (fragment-1-or-3 xlo ylo zlo split-xhi yhi zhi)
  (intersecting-fragments (vector-ref fragments 2)
  split-xhi ylo zlo xhi yhi zhi)))

(else ; all in 2
 (intersecting-fragments (vector-ref fragments 2)
 xlo ylo zlo xhi yhi zhi))))))

    (define (fragment-1-or-3 xlo ylo zlo xhi yhi zhi)
      ;; assumes that cuboid was split and that
      ;; split-xlo <= xlo <= xhi <= split-xhi
      (let ((split-pt (cuboid.split-pt cuboid)))
(let ((split-ylo (max (flo:- (ycor split-pt) 1.) (cuboid.ylo cuboid)))
      (split-yhi (min (flo:+ (ycor split-pt) 1.) (cuboid.yhi cuboid))))
  (cond ((flo:> ylo split-yhi) ; in 1
 ;;(display "frag 1")
 (intersecting-fragments
  (vector-ref fragments 1)
  xlo (max ylo split-yhi) zlo xhi yhi zhi))
((flo:> yhi split-yhi) ; in 1
 ;;(display "frags 1+3")
 (append
  (intersecting-fragments
   (vector-ref fragments 1)
   xlo (max ylo split-yhi) zlo xhi yhi zhi)
  (intersecting-fragments
   (vector-ref fragments 3)
   xlo (min ylo split-ylo) zlo
   xhi split-ylo zhi)))
((and (flo:= yhi split-yhi)
      (flo:> ylo split-ylo)) ; in splitting cube
 '())
(else ; in 3
 ;;(display "frag 3 ")
```

```
 (intersecting-fragments
  (vector-ref fragments 3)
  xlo ylo zlo xhi (min yhi split-ylo) zhi))))))

    (let ((split-pt (cuboid.split-pt cuboid)))
      (cond ((flo:= (volume xlo ylo zlo xhi yhi zhi) 0.) '())
    ((not split-pt)
     (if (flo:= (cuboid.remaining-volume cuboid) 0.)
 '()
 (list (list cuboid xlo ylo zlo xhi yhi zhi))))
    (else
     (let ((split-zlo (max (flo:- (zcor split-pt) 1.)
   (cuboid.zlo cuboid)))
   (split-zhi (min (flo:+ (zcor split-pt) 1.)
   (cuboid.zhi cuboid))))
       (cond ((flo:< zhi split-zlo) ; is all of it in 4
      (intersecting-fragments (vector-ref fragments 4)
      xlo ylo zlo xhi yhi zhi))

     ((flo:< zlo split-zlo) ; is part of it in 4
      (append
       (intersecting-fragments (vector-ref fragments 4)
       xlo ylo zlo xhi yhi split-zlo)
       (intersecting-fragments cuboid
       xlo ylo split-zlo xhi yhi zhi)))

     ((flo:<= zhi split-zhi) ; is it at split height
      (fragments-0-to-3 xlo ylo zlo xhi yhi zhi))

     ((flo:< zlo split-zhi) ; is part in 5
      (append
       (fragments-0-to-3 xlo ylo zlo xhi yhi split-zhi)
       (intersecting-fragments (vector-ref fragments 5)
    xlo ylo split-zhi xhi yhi zhi)))

     (else ; all is in 5
      (intersecting-fragments (vector-ref fragments 5)
   xlo ylo zlo xhi yhi zhi)))))))))

(define (split-intersecting-cuboids cuboid x y z)
  ;; coordinates are centre of cube
  (let ((xlo (max (flo:- x 1.) 0.))
(xhi (flo:+ x 1.))
(ylo (max (flo:- y 1.) 0.))
(yhi (flo:+ y 1.))
(zlo (max (flo:- z 1.) 0.))
(zhi (flo:+ z 1.)))
    (let ((ancestor (smallest-containing-cuboid cuboid
xlo ylo zlo xhi yhi zhi)))
      ;; need to make a boundary check if ancestor is top level cuboid
      (let ((leaves
     (if (not (cuboid.parent ancestor))
 (intersecting-fragments ancestor
 (max xlo (cuboid.xlo ancestor))
 (max ylo (cuboid.ylo ancestor))
 (max zlo (cuboid.zlo ancestor))
 (min xhi (cuboid.xhi ancestor))
 (min yhi (cuboid.yhi ancestor))
 (min zhi (cuboid.zhi ancestor)))
```

```
 (intersecting-fragments ancestor xlo ylo zlo xhi yhi zhi))))
;; remove or split each leaf.
;;(pp leaves)
(for-each (lambda (leaf)
    (let ((cuboid (car leaf))
  (overlap-vol (apply volume (cdr leaf))))
      (apply split leaf)
      (set-cuboid.split-pt! cuboid (make-coord x y z))
      (let loop ((c cuboid))
(if (not c)
    unspecific
    (let ((new-vol (flo:- (cuboid.remaining-volume c)
  overlap-vol)))
      ;;compensate for flonum roundoff errors
      (if (flo:< (flo:abs new-vol) 1e-12)
  (set-cuboid.remaining-volume! c 0.)
  (set-cuboid.remaining-volume! c new-vol))
      (loop (cuboid.parent c)))))))
  leaves)))))

(define (split cuboid xlo ylo zlo xhi yhi zhi)
  ;; assume the coordinates describe a cuboid that intersects CUBOID.
  (let ((cuboid-xlo (cuboid.xlo cuboid))
(cuboid-xhi (cuboid.xhi cuboid))
(cuboid-ylo (cuboid.ylo cuboid))
(cuboid-yhi (cuboid.yhi cuboid))
(cuboid-zlo (cuboid.zlo cuboid))
(cuboid-zhi (cuboid.zhi cuboid)))
    ;; need to compute intersection limits
    (let ((int-xlo (max xlo cuboid-xlo))
  (int-xhi (min xhi cuboid-xhi))
  (int-ylo (max ylo cuboid-ylo))
  (int-yhi (min yhi cuboid-yhi))
  (int-zlo (max zlo cuboid-zlo))
  (int-zhi (min zhi cuboid-zhi)))
      (let ((fragments
     (vector
      ;; fragments 0 - 3 are only as thick as the cube
      (make-cuboid cuboid-xlo cuboid-ylo int-zlo
   int-xlo cuboid-yhi int-zhi)
      (make-cuboid int-xlo int-yhi int-zlo
   int-xhi cuboid-yhi int-zhi)
      (make-cuboid int-xhi cuboid-ylo int-zlo
   cuboid-xhi cuboid-yhi int-zhi)
      (make-cuboid int-xlo cuboid-ylo int-zlo
   int-xhi int-ylo int-zhi)

      ;; fragments 4 and 5 are bounded by cuboid z components
      (make-cuboid cuboid-xlo cuboid-ylo cuboid-zlo
   cuboid-xhi cuboid-yhi int-zlo)
      (make-cuboid cuboid-xlo cuboid-ylo int-zhi
   cuboid-xhi cuboid-yhi cuboid-zhi))))

(for-each-vector-element fragments
 (lambda (fragment)
   (set-cuboid.parent! fragment cuboid)))
(set-cuboid.fragments! cuboid fragments)
;;(set-cuboid.split-pt! cuboid (make-coord x y z))
;; should only subtract portion that overlaps with cuboid.
```

```
;;(set-cuboid.remaining-volume! cuboid
;;   (flo:- (cuboid.remaining-volume cuboid)
;;       int-vol))
cuboid))))

(define (interval-random low high)
  (+ low (random (- high low))))

(define (adjust-volume! cuboid)
  (if (not cuboid)
      #f
      (begin
(set-cuboid.remaining-volume!
 cuboid
 (let ((sum-vol 0.))
   (for-each-vector-element
    (cuboid.fragments cuboid)
    (lambda (fragment)
      (set! sum-vol (+ sum-vol (cuboid.remaining-volume fragment)))))
   sum-vol))
(adjust-volume! (cuboid.parent cuboid)))))

(define (find-selected-cuboid cuboid selector start)
  ;; Assume that selector is valid. ie selector < remaining-volume of cuboid
  (if (not (cuboid.split-pt cuboid))
      cuboid
      (let ((fragments (cuboid.fragments cuboid)))
(let loop ((fragment (vector-ref fragments start))
   (sel selector)
   (next (modulo (fix:+ start 1) 6)))
  (let ((fragment-vol (cuboid.remaining-volume fragment)))
    (cond ((< sel fragment-vol)
   (find-selected-cuboid fragment sel next))
  ((= next start)
   ;; volumes are not consistent. (flonum roundoff)
   (adjust-volume! cuboid)
   ;;(find-selected-cuboid parent selector start)
   #f)
  (else
   (loop (vector-ref fragments next)
 (flo:- sel fragment-vol)
 (modulo (fix:+ next 1) 6)))))))))

(define (add-unit-cube cuboid fx fy fz)
  (let ((selector (random (cuboid.remaining-volume cuboid))))
    (let ((chosen (find-selected-cuboid cuboid selector (random 6))))
      (if (not chosen)
  (if (> (cuboid.remaining-volume cuboid) 0.)
      (add-unit-cube cuboid fx fy fz) ;try again
      #f) ; can't add a cube
  (let ((x (fx (cuboid.xlo chosen) (cuboid.xhi chosen)))
(y (fy (cuboid.ylo chosen) (cuboid.yhi chosen)))
(z (fz (cuboid.zlo chosen) (cuboid.zhi chosen))))
    (split-intersecting-cuboids chosen x y z)
    cuboid)))))

(define (make-non-overlapping-pts n width height #!optional depth)
  (let ((fail (lambda (count result)
(newline)
```

```
(display "Could place only ")
(display count)
(display " processors")
(vector-map result
    (lambda (axis)
      (subvector axis 0 count))))))
    (if (default-object? depth)
(%make-non-overlapping-pts fail n width height)
(%make-non-overlapping-pts fail n width height depth))))

(define (%make-non-overlapping-pts fail n width height #!optional depth)
  ;; unit of distance is processor diameter.
  (let ((fz (if (default-object? depth)
(lambda (low high) low high 1.)
interval-random)) ; assume uniform distribution.
(depth (if (default-object? depth) 2. depth))
(result-x (make-vector n 0.))
(result-y (make-vector n 0.))
(result-z (make-vector n 1.)))
    (let loop ((count 0)
       (cuboid (make-cuboid 0. 0. 0.
    (exact->inexact width)
    (exact->inexact height)
    (exact->inexact depth))))
      ;;(show-cuboid/z win result)
      (if (< count n)
  (if (> (cuboid.remaining-volume cuboid) 0.)
      (let ((selector (random (cuboid.remaining-volume cuboid))))
(let ((chosen (find-selected-cuboid cuboid selector
    (random 6))))
  (if (not chosen)
      (loop count cuboid) ;try again with new volume
      (let ((x (interval-random (cuboid.xlo chosen)
(cuboid.xhi chosen)))
    (y (interval-random (cuboid.ylo chosen)
(cuboid.yhi chosen)))
    (z (fz (cuboid.zlo chosen) (cuboid.zhi chosen))))
(split-intersecting-cuboids chosen x y z)
(vector-set! result-x count x)
(vector-set! result-y count y)
(vector-set! result-z count z)
(loop (fix:+ count 1) cuboid)))))
      (fail count (vector result-x result-y result-z)))
  (vector result-x result-y result-z)) ;points format
      )))

;;;; Functions to help testing

(define test-non-overlapping-pts
  (let ((win #f)
(x-margin 1.)
(y-margin 1.))
    (lambda (n width height #!optional depth)
      ;; unit is processor diameter.
      ;; assume uniform distribution.
      (let ((fz (if (default-object? depth)
    (lambda (low high) low high 1.)
    interval-random))
    (depth (if (default-object? depth) 2. depth)))
```

```
(if (not win)
    (set! win (make-window 0. 0.
   (exact->inexact width)
   (exact->inexact height)
   x-margin y-margin))
    (begin
      (graphics-set-coordinate-limits
       win
       (- x-margin) (- y-margin)
       (+ width x-margin) (+ height y-margin))
      (graphics-clear win)))

(let ((init-cuboid (make-cuboid 0. 0. 0.
(exact->inexact width)
(exact->inexact height)
(exact->inexact depth))))
  (let loop ((count 0) (result init-cuboid))
    ;;(show-cuboid/z win result)
    (if (< count n)
(if (> (cuboid.remaining-volume result) 0.)
    (if (add-unit-cube result
 interval-random interval-random fz)
(loop (fix:+ count 1) result)
(loop count result))
    (begin
      (newline)
      (display "Could place only ")
      (display count)
      (display " processors")
      (show-cuboid-leaves/z win result)
      (show-cuboid-points/z win result)
      result))
(begin
  (show-cuboid-leaves/z win result)
  (show-cuboid-points/z win result)
  result)))))))

(define (make-window xlo ylo xhi yhi x-margin y-margin)
  (let ((g (make-graphics-device 'x #f "400x400")))
    (graphics-operation g 'set-foreground-color "green")
    (graphics-operation g 'set-background-color "black")
    (graphics-set-coordinate-limits g
    (- xlo x-margin) (- ylo y-margin)
    (+ xhi x-margin) (+ yhi y-margin))
    (graphics-clear g)
    g))

(define (draw-rectangle win xlo ylo xhi yhi)
  (graphics-draw-line win xlo ylo xhi ylo)
  (graphics-draw-line win xhi ylo xhi yhi)
  (graphics-draw-line win xhi yhi xlo yhi)
  (graphics-draw-line win xlo yhi xlo ylo))

(define (draw-x-in-rectangle win xlo ylo xhi yhi)
  (draw-rectangle win xlo ylo xhi yhi)
  (graphics-draw-line win xlo ylo xhi yhi)
  (graphics-draw-line win xlo yhi xhi ylo))

(define (show-cuboid/z win cuboid)
```

```
  (draw-rectangle win
  (cuboid.xlo cuboid) (cuboid.ylo cuboid)
  (cuboid.xhi cuboid) (cuboid.yhi cuboid))
  (if (not (cuboid.split-pt cuboid))
      'done
      (for-each-vector-element (cuboid.fragments cuboid)
       (lambda (fragment)
 (show-cuboid/z win fragment)))))

(define (show-cuboid-leaves/z win cuboid)
  (cond ((<= (cuboid.remaining-volume cuboid) 0.)
 unspecific)
((not (cuboid.split-pt cuboid))
 (draw-x-in-rectangle win
      (cuboid.xlo cuboid) (cuboid.ylo cuboid)
      (cuboid.xhi cuboid) (cuboid.yhi cuboid)))
(else
 (for-each-vector-element (cuboid.fragments cuboid)
  (lambda (fragment)
    (show-cuboid-leaves/z win fragment))))))

(define (show-cuboid-points/z win cuboid)
  (let ((split-pt (cuboid.split-pt cuboid)))
    (cond ((not split-pt)
   (if (> (cuboid.remaining-volume cuboid) 0.)
       (draw-x-in-rectangle win
(cuboid.xlo cuboid) (cuboid.ylo cuboid)
(cuboid.xhi cuboid) (cuboid.yhi cuboid))
       unspecific))
  (else
   (graphics-operation win 'fill-circle
       (xcor split-pt) (ycor split-pt) .5)
   (for-each-vector-element (cuboid.fragments cuboid)
    (lambda (fragment)
      (show-cuboid-points/z win fragment)))))))

(define (list-points cuboid)
  (define (merge pts1 pts2)
    (cond ((null? pts1) pts2)
  ((null? pts2) pts1)
  (else
   (let ((p1 (car pts1))
 (p2 (car pts2)))
     (cond ((lex:< p1 p2)
    (cons p1 (merge (cdr pts1) pts2)))
   ((lex:= p1 p2)
    (cons p1 (merge (cdr pts1) (cdr pts2))))
   (else
    (cons p2 (merge pts1 (cdr pts2)))))))))
  (let ((split-pt (cuboid.split-pt cuboid)))
    (if (not split-pt)
'()
(let ((fragments (vector->list (cuboid.fragments cuboid))))
  (fold-left merge (list split-pt)
     (map list-points fragments))))))

#|
(merge (list-points (vector-ref fragments 0))
       (merge (list-points (vector-ref fragments 3))
```

```
      (merge (list split-pt)
     (merge (list-points (vector-ref fragments 1))
    (merge (list-points
    (vector-ref fragments 2))
   (merge (list-points
   (vector-ref fragments 4))
  (list-points
   (vector-ref fragments 5))))))))
|#

(define (lex:< p1 p2)
  (or (< (xcor p1) (xcor p2))
      (and (= (xcor p1) (xcor p2))
   (or (< (ycor p1) (ycor p2))
       (and (= (ycor p1) (ycor p2))
    (< (zcor p1) (zcor p2)))))))

(define (lex:> p1 p2)
  (or (> (xcor p1) (xcor p2))
      (and (= (xcor p1) (xcor p2))
   (or (> (ycor p1) (ycor p2))
       (and (= (ycor p1) (ycor p2))
    (> (zcor p1) (zcor p2)))))))

(define (lex:= p1 p2)
  (and (= (xcor p1) (xcor p2))
       (= (ycor p1) (ycor p2))
       (= (zcor p1) (zcor p2))))

(define (vector:find-first pred v)
  (let loop ((i (fix:- (vector-length v) 1)))
    (cond ((< i 0) #f)
  ((pred (vector-ref v i))
   (vector-ref v i))
  (else (loop (fix:- i 1))))))

(define (overlap? c1 c2)
  ;; returns true if c1 and c2 have intersecting volumes
  (let ((int-xlo (max (cuboid.xlo c1) (cuboid.xlo c2)))
(int-xhi (min (cuboid.xhi c1) (cuboid.xhi c2)))
(int-ylo (max (cuboid.ylo c1) (cuboid.ylo c2)))
(int-yhi (min (cuboid.yhi c1) (cuboid.yhi c2)))
(int-zlo (max (cuboid.zlo c1) (cuboid.zlo c2)))
(int-zhi (min (cuboid.zhi c1) (cuboid.zhi c2))))
    (and (flo:< int-xlo int-xhi)
 (flo:< int-ylo int-yhi)
 (flo:< int-zlo int-zhi))))

(define (overlapping-fragments? cuboid)
  (if (not (cuboid.split-pt cuboid))
      #f
      (let ((fragments (cuboid.fragments cuboid)))
(let i-loop ((i 0))
  (if (> i 4)
      (vector:find-first overlapping-fragments? fragments)
      (let j-loop ((j (fix:+ i 1)))
(cond ((> j 5) (i-loop (fix:+ i 1)))
      ((overlap? (vector-ref fragments i)
 (vector-ref fragments j))
```

```
       (cons (vector-ref fragments i)
     (vector-ref fragments j)))
      (else (j-loop (fix:+ j 1))))))))))

(define (all-fragments-contained? cuboid)
  (if (not (cuboid.split-pt cuboid))
      #t
      (let ((fragments (cuboid.fragments cuboid)))
(let ((violator (vector:find-first
 (lambda (fragment)
   (not (inside? cuboid
 (cuboid.xlo fragment)
 (cuboid.ylo fragment)
 (cuboid.zlo fragment)
 (cuboid.xhi fragment)
 (cuboid.yhi fragment)
 (cuboid.zhi fragment))))
 fragments)))
  (if (not violator)
      (let loop ((i 0))
(or (> i 5)
    (and (all-fragments-contained? (vector-ref fragments i))
 (loop (fix:+ i 1)))))
      violator)))))

(define (leaves->list cuboid)
  ;; returns a list of leaf cuboids of CUBOID
  (cond ((= (cuboid.remaining-volume cuboid) 0.) '())
((not (cuboid.split-pt cuboid)) (list cuboid))
(else
 (let ((fragments (cuboid.fragments cuboid)))
   (let loop ((i 5) (result '()))
     (if (< i 0)
 result
 (loop (- i 1)
       (append (leaves->list (vector-ref fragments i))
       result)))))))))

(define (sq x) (* x x))

(define (overlapping-pts? points)
  ;; Assuming radius of 1, decide whether any discs with centres given in
  ;; POINTS will overlap each other.
  (define (discs-overlap? x1 y1 z1 x2 y2 z2)
    (< (+ (sq (- x2 x1)) (sq (- y2 y1)) (sq (- z2 z1)))
       1))

  ;; don't care about efficiency for now
  (let ((xs (vector-ref points 0))
(ys (vector-ref points 1))
(zs (vector-ref points 2)))
    (let i-loop ((i (- (vector-length xs) 1)))
      (if (< i 0)
  #f
  (let ((xi (vector-ref xs i))
(yi (vector-ref ys i))
(zi (vector-ref zs i)))
    (let j-loop ((j (fix:- i 1)))
      (cond ((< j 0) (i-loop (fix:- i 1)))
```

```
((discs-overlap? xi yi zi
 (vector-ref xs j)
 (vector-ref ys j)
 (vector-ref zs j))
 (list i j))
(else (j-loop (fix:- j 1))))))))))
```

## C.3 Support code

These are generally useful subroutines, or utilities written especially for the illustrator that do not have much to do with GPL *per se.*

**queue.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

;;;; Message queues

(define (make-empty-queue tag)
  (let ((data (list tag)))
    (cons data data)))

(define-integrable (queue/items q) (car q))
(define-integrable (queue/last q) (cdr q))
(define-integrable (set-queue/last! q item)
  (set-cdr! q item))

(define (queue/empty? q)
  (eq? (queue/items q) (queue/last q)))

(define (queue/add! q item)
  ;; make sure this executes atomically
  (let ((last-element (list item)))
    (set-cdr! (queue/last q) last-element)
    (set-queue/last! q last-element)
    q))

(define (queue/remove! q)
  ;; make sure this executes atomically
  (let ((data (queue/items q)))
    (let ((msg (cadr data)))
      (if (eq? (cdr data) (queue/last q))
  (set-queue/last! q data))
      (set-cdr! data (cddr data))
      msg)))

(define (queue/length q)
  ;; subtract 1 for the tag
  (- (length (queue/items q)) 1))
```

**../syncsim/tools.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(load-option 'format)

(define pixels/xinterval 25) ; 1 xunit = 25 pixels
(define pixels/yinterval 25) ; 1 yunit = 25 pixels
(define preferred-size 300) ; pixel height or width of window

(define (plot x-values y-values mark-proc width height #!optional font)
  (let ((font (if (default-object? font) "6x9" font)))
    (find-min&max
     x-values
     (lambda (xlo xhi)
       (find-min&max
y-values
(lambda (ylo yhi)
  (win-plot x-values y-values mark-proc
    (make-graph-win width height font xlo xhi ylo yhi))))))))

(define (int-plot x-values y-values mark-proc x-interval y-interval
  #!optional font)
  (let ((font (if (default-object? font) "6x9" font)))
    (find-min&max
     x-values
     (lambda (xlo xhi)
       (find-min&max
y-values
(lambda (ylo yhi)
  (let ((win (make-graph-win (* pixels/xinterval
(/ (* 1.2 (- xhi xlo)) x-interval))
     (* pixels/yinterval
(/ (* 1.2 (- yhi ylo)) y-interval))
     font
     xlo xhi ylo yhi)))
    (win-plot x-values y-values mark-proc win))))))))

(define (win-plot x-values y-values mark-proc win)
  (let ((mark (mark-proc win)))
    (let loop ((i (- (vector-length x-values) 1)))
      (if (>= i 0)
  (begin
    (mark (vector-ref x-values i)
  (vector-ref y-values i))
    (loop (- i 1)))
  win))))

(define (make-graph-win width height font-name xlo xhi ylo yhi)
  ;; lo and hi are the min and max values for the indicated axes.
  (let ((w (make-window width height)))
    (let ((font (graphics-operation w 'font-structure font-name)))
      (if (not font)
  (error "Unknown Font Requested" font-name)
  (graphics-operation w 'set-font font-name))
```

```
      (let ((bounds (x-font-structure/max-bounds font)))
(let ((max-width (x-character-bounds/width bounds))
      (ascent (x-character-bounds/ascent bounds))
      (descent (x-character-bounds/descent bounds)))
  ;; the axis multiplier is never more than 7 characters.
  (let ((pix-x-margin (* 7 max-width))
(pix-y-margin (* 4 (+ ascent descent)))
(xlo (exact->inexact xlo))
(xhi (exact->inexact xhi))
(ylo (exact->inexact ylo))
(yhi (exact->inexact yhi)))

    (newline)
    (display `(pix-x-margin: ,pix-x-margin))
    (display `(pix-y-margin: ,pix-y-margin))
    (newline)

    (let* ((xunits/pixel (/ (- xhi xlo) (- width (* 2 pix-x-margin))))
   (yunits/pixel (/ (- yhi ylo) (- height (* 2 pix-y-margin))))
   (x-margin (* pix-x-margin xunits/pixel))
   (y-margin (* pix-y-margin yunits/pixel))
   (dx (* pixels/xinterval xunits/pixel))
   (max-chars (+ (max (+ (floor->exact
  (log10 (abs (/ xhi dx))))
 1)
      3)
 1)) ; one char for decimal point
   (pixels/xinterval (max pixels/xinterval
  (* (+ max-chars 1) max-width)))
   ;;(x-margin (* .1 (- xhi xlo)))
   ;;(y-margin (* .1 (- yhi ylo)))
   )
      (let ((x-range (+ (- xhi xlo) (* 2 x-margin)))
    (y-range (+ (- yhi ylo) (* 2 y-margin)))
    (l-margin x-margin)
    (r-margin x-margin)
    (t-margin y-margin)
    (b-margin y-margin))
(let ((x-interval (round-sig-fig (* xunits/pixel
    pixels/xinterval)
 1))
      (y-interval (round-sig-fig (* yunits/pixel
    pixels/yinterval)
 1)))
  (let ((axis-x (if (between? xlo xhi 0)
    0. ; no adjustment needed
    (let ((room (* .25 x-margin)))
      (set! r-margin (- r-margin room))
      (set! l-margin (+ l-margin room))
      (- xlo room))))
(axis-y (if (between? ylo yhi 0)
    0. ; no adjustment needed
    (let ((room (* .25 y-margin)))
      (set! t-margin (- t-margin room))
      (set! b-margin (+ b-margin room))
      (- ylo room)))))
    (graphics-set-coordinate-limits
     w
     (- xlo l-margin) (- ylo b-margin)
```

```
     (+ xhi r-margin) (+ yhi t-margin))
    ;; draw axes at the appropriate places
    (graphics-operation w 'set-foreground-color "gray100")
    (draw-axes w font-name axis-x axis-y x-interval y-interval
       xlo ylo xhi yhi)
    (graphics-operation w 'set-foreground-color "black")
    w)))))))))

(define (draw-axes w font-name x y dx dy xlo ylo xhi yhi)
  (let ((mark-len 5))
    (draw-horiz-axis w font-name mark-len x y dx xlo xhi)
    (draw-vert-axis w font-name mark-len x y dy ylo yhi)))

(define (draw-horiz-axis w font-name mark-len x y dx xlo xhi)
  (let* ((marker (mark-point w mark-len))
 (multiplier (expt 10 (floor->exact (log10 (abs xlo)))))
 (digits (max (+ (floor->exact (log10 (abs (/ xhi dx))))
 1)
      3))
 (horiz-labeler
  (label-point
   w font-name #f mark-len
   (lambda (x y)
     (fluid-let ((flonum-unparser-cutoff `(relative ,digits)))
       (format #f (string-append "~" (number->string (+ digits 1)) "A")
       (number->string (exact->inexact (/ x multiplier))))))))
 (height (font-height w font-name))
 (scale-labeler
  (label-point w font-name #f mark-len
       (scale-label multiplier))))
    (let ((final-vals
   (process-intervals
    w dx 0. xlo y xhi y
    (lambda (x y x-units y-units x-perp y-perp)
      (marker x y x-units y-units x-perp y-perp)
      (horiz-labeler x y x-units y-units x-perp y-perp)))))
      (graphics-draw-line w xlo y (car final-vals) y)
      (if (not (= multiplier 1))
  (apply scale-labeler
 (/ (+ xlo xhi) 2)
 (- (cadr final-vals) (* height (cadddr final-vals)))
 (cddr final-vals))))))

(define (draw-vert-axis w font-name mark-len x y dy ylo yhi)
  (let* ((marker (mark-point w mark-len))
 (multiplier (expt 10 (floor->exact (log10 (abs ylo)))))
 (digits (max (+ (floor->exact (log10 (abs (/ yhi dy))))
 1)
      3))
 (vert-labeler
  (label-point
   w font-name #t mark-len
   (lambda (x y)
     (fluid-let ((flonum-unparser-cutoff `(relative ,digits)))
       (format #f (string-append "~" (number->string (+ digits 1)) "A")
       (number->string (exact->inexact (/ y multiplier))))))))
 (height (font-height w font-name))
 (scale-labeler
  (label-point w font-name #t mark-len
```

```
       (scale-label multiplier))))
    (let ((final-vals
   (process-intervals
    w 0. dy x ylo x yhi
    (lambda (x y x-units y-units x-perp y-perp)
      (marker x y x-units y-units x-perp y-perp)
      (vert-labeler x y x-units y-units x-perp y-perp)))))
      (graphics-draw-line w x ylo x (cadr final-vals))
      (if (not (= multiplier 1))
  ;;(apply scale-labeler (car final-vals) yhi (cddr final-vals))
  (apply scale-labeler
 (car final-vals)
 (+ (cadr final-vals) (* height (cadddr final-vals)))
 (cddr final-vals))))))

(define (scale-label multiplier)
  (lambda (x y)
    (if (= multiplier 1)
""
(fluid-let ((flonum-unparser-cutoff '(relative 1 scientific)))
  (string-append "x" (number->string
      (if (> multiplier 1)
  multiplier
  (exact->inexact multiplier))))))))

(define (process-intervals win dx dy xstart ystart xstop ystop f)
  (let ((mark-angle (atan (- dx) dy)))
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((x-perp (cos mark-angle))
      (y-perp (sin mark-angle)))
  (let loop ((x xstart) (y ystart))
    (f x y x-units y-units x-perp y-perp)
    (if (and (between? xstart xstop x) (between? ystart ystop y))
(loop (+ x dx) (+ y dy))
(list x y x-units y-units x-perp y-perp))))))))

#|
(define (process-intervals win dx dy xstart ystart xstop ystop f)
  (let ((mark-angle (atan (- dx) dy)))
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((x-perp (cos mark-angle))
      (y-perp (sin mark-angle)))
  (let loop ((x xstart) (y ystart))
    (if (and (between? xstart xstop x) (between? ystart ystop y))
(begin
  (f x y x-units y-units x-perp y-perp)
  (loop (+ x dx) (+ y dy)))
(list x y x-units y-units x-perp y-perp))))))))
|#

(define (draw-intervals win mark-len dx dy xstart ystart xstop ystop)
  (process-intervals
   win dx dy xstart ystart xstop ystop
   (mark-point win mark-len)))
```

```
(define (mark-point win mark-len)
  (lambda (x y x-units y-units x-perp y-perp)
    (let ((mark-dx (* .5 (* mark-len x-perp x-units)))
  (mark-dy (* .5 (* mark-len y-perp y-units))))
      (graphics-draw-line win
  (- x mark-dx) (- y mark-dy)
  (+ x mark-dx) (+ y mark-dy)))))

(define (label-point win font-name left? dist label)
  (let ((font (graphics-operation win 'font-structure font-name)))
    (if (not font)
(error "Unknown Font Requested" font-name)
(graphics-operation win 'set-font font-name))
    (let ((bounds (x-font-structure/max-bounds font)))
      (let ((max-width (x-character-bounds/width bounds))
    (ascent (x-character-bounds/ascent bounds))
    (descent (x-character-bounds/descent bounds)))
(if left?
    (lambda (x y x-units y-units x-perp y-perp)
      (let ((text (label x y)))
(let ((width (* (string-length text) max-width)))
  (graphics-draw-text win
      (- x (* x-units
      (+ (* dist x-perp) width)))
      (- y (* y-units
      (+ (* dist y-perp) descent)))
      text))))
    (lambda (x y x-units y-units x-perp y-perp)
      (let ((text (label x y)))
(let ((width (* (string-length text) max-width)))
  (graphics-draw-text win
      (+ x (* x-units
      (- (* dist x-perp) (/ width 2))))
      (+ y (* y-units
      (- (* dist y-perp) ascent)))
      text)))))))))

(define (font-height win font-name)
  (let ((font (graphics-operation win 'font-structure font-name)))
    (if (not font)
(error "Unknown Font Requested" font-name))
    (let ((bounds (x-font-structure/max-bounds font)))
      (let ((max-width (x-character-bounds/width bounds))
    (ascent (x-character-bounds/ascent bounds))
    (descent (x-character-bounds/descent bounds)))
(+ ascent descent)))))

(define (mark-+ size)
  ;; size is in pixels
  (lambda (win)
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((xsize (* .5 (* size x-units)))
      (ysize (* .5 (* size y-units))))
  (lambda (x y)
    (graphics-draw-line win (- x xsize) y (+ x xsize) y)
    (graphics-draw-line win x (- y ysize) x (+ y ysize))))))))
```

```
(define (mark-x size)
  ;; size is in pixels
  (lambda (win)
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((xsize (* .5 (* size x-units)))
      (ysize (* .5 (* size y-units))))
  (lambda (x y)
    (graphics-draw-line win (- x xsize) (- y ysize)
(+ x xsize) (+ y ysize))
    (graphics-draw-line win (- x xsize) (+ y ysize)
(+ x xsize) (- y ysize)))))))

(define (pixels->units win)
  (with-values
      (lambda () (graphics-device-coordinate-limits win))
    (lambda (left bot right top)
      (with-values
  (lambda () (graphics-coordinate-limits win))
(lambda (xlo ylo xhi yhi)
  (values (/ (- xhi xlo) (+ (- right left) 1))
  (/ (- yhi ylo) (+ (- bot top) 1))))))))

(define (log10 v)
  (/ (log v) (log 10)))

(define (round-sig-fig v n)
  (let ((d (floor->exact (/ (log (abs v)) (log 10)))))
    (let ((v* (* v (expt 10 (- n 1 d)))))
      (let ((shift (expt 10 (- (+ d 1) n))))
(exact->inexact (* (round->exact v*) shift))))))

;;; These probably properly belong in utils.scm

(define (vector:for-each p v1 . vs)
  (let ((n (vector-length v1)))
    (cond ((null? vs)
   (for-each-vector-element v1 p))
  ((null? (cdr vs))
   (let ((v2 (car vs)))
     (let loop ((i 0))
       (if (< i n)
   (begin
     (p (vector-ref v1 i) (vector-ref v2 i))
     (loop (fix:+ i 1)))))))
  ((null? (cddr vs))
   (let ((v2 (car vs))
 (v3 (cadr vs)))
     (let loop ((i 0))
       (if (< i n)
   (begin
     (p (vector-ref v1 i) (vector-ref v2 i)
(vector-ref v3 i))
     (loop (fix:+ i 1)))))))
  (else
   (let loop ((i 0))
     (if (< i n)
 (begin
```

```
   (apply p
  (vector-ref v1 i)
  (map (lambda (v*) (vector-ref v* i))
       vs))
   (loop (fix:+ i 1)))))))))

(define (for-each-index start stop inc p)
  (let loop ((i start))
    (if (between? start stop i)
(begin
  (p i)
  (loop (+ i inc))))))

(define (map-vector f v1 . vs)
  (let ((n (vector-length v1)))
    (let ((result (make-vector n #f)))
      (cond ((null? vs)
     (vector-map v1 f))
    ((null? (cdr vs))
     (let ((v2 (car vs)))
       (let loop ((i 0))
 (if (< i n)
     (begin
       (vector-set! result i
    (f (vector-ref v1 i)
       (vector-ref v2 i)))
       (loop (fix:+ i 1)))
     result))))
    ((null? (cddr vs))
     (let ((v2 (car vs))
   (v3 (cadr vs)))
       (let loop ((i 0))
 (if (< i n)
     (begin
       (vector-set! result i
    (f (vector-ref v1 i)
       (vector-ref v2 i)
       (vector-ref v3 i)))
       (loop (fix:+ i 1)))
     result))))
    (else
     (let loop ((i 0))
       (if (< i n)
   (begin
     (vector-set!
      result i
      (apply f
     (vector-ref v1 i)
     (map (lambda (v*) (vector-ref v* i))
  vs)))
     (loop (fix:+ i 1)))
   result)))))))

(define (vector-acc combiner base vec)
  (let loop ((i (- (vector-length vec) 1)) (result base))
    (if (< i 0)
result
(loop (- i 1) (combiner (vector-ref vec i) result)))))
```

```
(define (vector:multi-acc combiners bases vec . rest)
  (if (null? rest)
      (let loop ((i (- (vector-length vec) 1)) (result bases))
(if (< i 0)
    result
    (let ((elt (vector-ref vec i)))
      (loop (- i 1)
    (map (lambda (f x) (f x elt))
 combiners
 result)))))
      (let ((vecs (cons vec rest)))
(let loop ((i (- (vector-length vec) 1)) (result bases))
  (if (< i 0)
      result
      (let ((elts (map (lambda (vec) (vector-ref vec i))
       vecs)))
(loop (- i 1)
      (map (lambda (f x) (apply f x elts))
   combiners
   result))))))))

;;; Statistics

(define (id x) x)
(define (sq x) (* x x))

(define (vector:sum-terms terms vec . rest)
  (if (null? rest)
      (vector:multi-acc (map (lambda (f)
       (lambda (result elt)
 (+ result (f elt))))
     terms)
(map (lambda (f) 0) terms)
vec)
      (apply vector:multi-acc
     (map (lambda (f)
    (lambda (result . elts)
      (+ result (apply f elts))))
  terms)
     (map (lambda (f) f 0) terms)
     vec rest)))

(define (mean vec)
  (let ((n (vector-length vec)))
    (if (= n 0)
0
(let ((sum-list (vector:sum-terms (list id) vec)))
  (exact->inexact (/ (car sum-list) n))))))

(define (variance vec)
  (let ((n (vector-length vec)))
    (if (= n 0)
0
(let ((result (vector:sum-terms (list id sq) vec)))
  (exact->inexact (- (/ (cadr result) n)
     (sq (/ (car result) n))))))))

(define (mean+st-dev vec)
  (let ((n (vector-length vec)))
```

```
    (if (= n 0)
0
(let ((result (vector:sum-terms (list id sq) vec)))
  (let ((mean (exact->inexact (/ (car result) n))))
    (list
     mean
     (sqrt (exact->inexact (- (/ (cadr result) n)
      (sq mean))))))))))

(define (st-dev vec)
  (sqrt (variance vec)))

(define (covariance x-vec y-vec)
  (let ((nx (vector-length x-vec))
(ny (vector-length y-vec)))
    (if (or (= nx 0) (not (= nx ny)))
(error "Unequal numbers of data points passed to COVARIANCE")
(let ((result (vector:sum-terms
       (list (lambda (x y) x)
     (lambda (x y) y)
     (lambda (x y) (* x y)))
       x-vec y-vec)))
  (exact->inexact (- (/ (caddr result) nx)
     (* (/ (car result) nx)
(/ (cadr result) ny))))))))

(define (correlation x-vec y-vec)
  (/ (covariance x-vec y-vec)
     (sqrt (* (variance x-vec)
      (variance y-vec)))))

#|
(define (mean value-vec)
  (let ((n (vector-length value-vec)))
    (if (= n 0)
0
(let loop ((i (- n 1)) (sum 0))
  (if (< i 0)
      (exact->inexact (/ sum n))
      (loop (- i 1) (+ sum (vector-ref value-vec i))))))))

(define (variance value-vec)
  (let ((n (vector-length value-vec)))
    (if (= n 0)
0
(let loop ((i (- n 1)) (sum 0) (sq-sum 0))
  (if (< i 0)
      (exact->inexact (- (/ sq-sum n) (sq (/ sum n))))
      (let ((x (vector-ref value-vec i)))
(loop (- i 1) (+ sum x) (+ sq-sum (* x x)))))))))
|#

;;; Simulation specific tools

(define (with-dists-and-state sim src-index state-fn k)
  (let ((src-x (vector-ref (sim.xs sim) src-index))
(src-y (vector-ref (sim.ys sim) src-index))
(src-z (vector-ref (sim.zs sim) src-index))
(sq (lambda (x) (* x x))))
```

```
    (let ((x-values (map-vector
     (lambda (x y z)
       (sqrt (+ (sq (- x src-x))
(sq (- y src-y))
(sq (- z src-z)))))
     (sim.xs sim) (sim.ys sim) (sim.zs sim)))
  (y-values (vector-map (sim.states sim) state-fn)))
      (k x-values y-values))))

(define (sim-plot-vs-dist sim src-index state-fn)
  (with-dists-and-state sim src-index state-fn
(lambda (x-values y-values)
  (plot x-values y-values (mark-x 5) 400 200))))

(define (count-property pred vec)
  (vector-acc (lambda (item result)
(if (pred item)
    (+ 1 result)
    result))
      0
      vec))

(define (show-counts sim counts)
  (let ((max-count (apply max (vector->list counts))))
    (map-vector (lambda (index count)
  (sim-show-particle sim index
     (if (= count 0)
 "wheat"
 (number->grey count 0 max-count))))
(make-initialized-vector (sim.size sim) (lambda (i) i))
counts)
    #f))

(define (show-property+ sim sim-pred colour)
  ;; sim-pred = (lambda (sim index) ...) --> bool
  ;; colour = (lambda (sim index) ...) --> X colour string
  (for-each-index 0 (- (sim.size sim) 1) 1
  (lambda (index)
    (if (sim-pred sim index)
(sim-show-particle sim index
   (colour sim index))))))
```

**mouse-utils.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(define (probe-pheromone p-name)
  ;; probe pheromone p-name in the active domain
  (pointer:probe
   *D*
   (lambda (point)
     (write-line (list point
       (D:get-pheromone-value *D* p-name point))))))

(define (pointer:probe domain proc)
  (let ((pt->point (domain.pt->point domain))
(win (domain.win domain)))
    (let loop ()
      (with-pointer
       win
       (lambda (x y)
 (let ((point (pt->point (make-pt x y))))
   (proc point)
   (loop)))
       (lambda (x y)
 (let ((point (pt->point (make-pt x y))))
   (display-point-no-update domain point "yellow")
   (proc point)
   (loop)))
       (lambda (x y)
 x y
 'probe-done)))))

(define (pointer:with-n-points n domain proc)
  (let ((pt->point (domain.pt->point domain))
(win (domain.win domain)))
    (if (not win)
(error "No window for domain" domain))
    (let loop ((n n) (lst '()))
      (if (= n 0)
  (proc (reverse lst))
  (with-pointer
   win
   (lambda (x y)
     (let ((point (pt->point (make-pt x y))))
       (display-point-no-update domain point "yellow")
       (loop (- n 1) (cons point lst)))))))))

(define (pointer:with-points domain proc)
  ;; Left click to select points.  Right click to apply proc to them
  ;; middle click to see points selected so far.

  (let ((win (domain.win domain))
(pt->point (domain.pt->point domain)))

    (define (adjoin x y lst)
      (let ((point (pt->point (make-pt x y))))
(if (memv point lst)
```

```
    lst
    (begin
      (display-point-no-update domain point "yellow")
      (cons point lst)))))

    (let loop ((points '()))
      (with-pointer win
    (lambda (x y)
      (loop (adjoin x y points)))
    (lambda (x y)
      (let ((new-points (adjoin x y points)))
(write-line new-points)
(loop new-points)))
    (lambda (x y)
      x y ; ignored
      (proc points))))))

(define (with-pointer win left #!optional middle right)

  (let ((middle (if (default-object? middle) list middle))
(right (if (default-object? right) list right)))
    (with-values
(lambda ()
  (graphics-coordinate-limits win))
      (lambda (xlo ylo xhi yhi)
(define (inside? x y)
  (and (between? xlo xhi x) (between? ylo yhi y)))

(define (loop)
  ((graphics-operation win 'query-pointer)
   (lambda (x y v)
     (cond ((< v 512) (loop))
   ((inside? x y)
    (button-down-loop v))
   (else
    (loop))))))

(define (button-down-loop v)
  ((graphics-operation win 'query-pointer)
   (lambda (x y new-v)
     (if (< new-v 512)
 (if (inside? x y)
     (cond ((= (fix:and v 512) 512) (left x y))
   ((= (fix:and v 1024) 1024) (middle x y))
   ((= (fix:and v 2048) 2048) (right x y))
   (else
    (newline)
    (display "No mouse button clicked!!")))
     (loop))
 (button-down-loop v)))))
(loop))))

(define (between? lo hi val)
  (or (<= lo val hi)
      (<= hi val lo)))
```

**utils.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

;;;; Environments
;;; A complex environment structure is not currently necessary for GPL
;;; since frames can be at most one level deep.  However, environments have
;;; been implemented in full generality here, special casing 0 and 1 level
;;; environments.

;;; An environment contains the parent in the first position of the
;;; aggregate data structure.
;;; Static environments are lists and contain symbols indicating the
;;; parameters and variables in each frame.
;;; Dynamic environments are vectors in one to one correspondence with the
;;; lists of the static environments.  Dynamic environments contain values
;;; to which the corresponding symbols are bound.

(define the-empty-env '())
(define-integrable (env:empty? env) (null? env))
(define (make-global-env)
  (cons primitives the-empty-env))

(define (add-binding env var val)
  (set-car! env
    (cons (list var val) (car env)))
  env)

(define (assign env var val)
  (let loop ((frame (car env)) (rest-frames (cdr env)))
    (let ((binding (assq var frame)))
      (cond (binding (set-car! (cdr binding) val))
    ((null? rest-frames)
     (error "Assignment to unbound variable" var))
    (else (loop (car rest-frames) (cdr rest-frames)))))))

(define (extend-env env params args)
  (cons (map list params args) env))

(define (lookup-var env var)
  (if (env:empty? env)
      (error "Unbound variable" var)
      (let ((bound? (assq var (car env))))
(if bound?
    (cadr bound?)
    (lookup-var (cdr env) var)))))

;;; Material Tables

(define (make-material-table pt-eqv)
  (let ((pt-table (make-empty-table pt-eqv))
(mat-table (make-empty-table eq?)))
    (vector pt-table mat-table)))

(define-integrable (table.pt mat-table) (vector-ref mat-table 0))
(define-integrable (table.mat mat-table) (vector-ref mat-table 1))
```

```
(define (add-points table material points)
  (put (table.mat table) material points))

(define (get-materials table point)
  (fold-right union '()
      (get (table.pt table) point)))

(define (get-points table material)
  (get (table.mat table) material))

(define (is-material? table point material)
  (subset? material (get-materials table point)))

;;; Backtracking tables

(define (add-backtracking-pt point instance-id prev-point)
  (let ((id-table (or (get backtracking-table point)
      (let ((new-table (make-empty-table eqv?)))
(put backtracking-table point new-table)
(list new-table)))))
    (put (car id-table) instance-id prev-point)))

(define (get-backtracking-pt point instance-id)
  (let ((id-table-lst (get backtracking-table point)))
    (and id-table-lst
 (remove-one (car id-table-lst) instance-id))))

;;; General Tables

(define (make-empty-table equ)
  (let ((table (make-vector 3 '()))
(lookup (association-procedure equ car)))

    (define (delete-key! key)
      (let ((entries (vector-ref table 0)))
(cond ((null? entries) #f)
      ((equ key (caar entries))
       (let ((result (car entries)))
 (vector-set! table 0 (cdr entries))
 (cdr result)))
      (else
       (let loop ((prev entries) (current (cdr entries)))
 (cond ((null? current) #f)
       ((equ key (caar current))
(set-cdr! prev (cdr current))
(cdar current))
       (else (loop current (cdr current)))))))))
    (vector-set! table 1 lookup)
    (vector-set! table 2 delete-key!)
    table))

(define-integrable (table.entries table) (vector-ref table 0))
(define-integrable (table.lookup table) (vector-ref table 1))
(define-integrable (table.delete-key! table) (vector-ref table 2))
(define-integrable (set-table.entries! table entries)
  (vector-set! table 0 entries))

(define (get table key)
```

```
  (let ((result ((table.lookup table) key (table.entries table))))
    (and result (cdr result))))

(define (put table key value)
  (let ((entries (table.entries table)))
    (let ((result ((table.lookup table) key entries)))
      (if result
  (set-cdr! result (cons value (cdr result)))
  (set-table.entries! table (cons (list key value) entries))))))

(define (replace table key value)
  (let ((entries (table.entries table)))
    (let ((result ((table.lookup table) key entries)))
      (if result
  (set-cdr! result (list value))
  (set-table.entries! table (cons (list key value) entries))))))

(define (remove table key)
  ((table.delete-key! table) key))

(define (remove-one table key)
  (let ((entries (table.entries table)))
    (let ((entry ((table.lookup table) key entries)))
      (and entry
   (let ((result (cadr entry))
 (new-lst (cddr entry)))
     (if (null? new-lst)
 ((table.delete-key! table) key)
 (set-cdr! entry new-lst))
     result)))))

(define (fold-table-values! table op base-val)
  ;; op only operates on values, so combiner is homogeneous in keys.
  ;; alternative is to have op:key --> val x result --> val
  (let ((entries (table.entries table)))
    (for-each (lambda (entry)
(set-cdr! entry (list (fold-right op base-val (cdr entry)))))
      entries)
    table))

#|
(define (make-empty-table equ)
  (let ((entries '())
(lookup (association-procedure equ car)))

    (define (delete-key! key)
      (cond ((null? entries) #f)
    ((equ key (caar entries))
     (let ((result (car entries)))
       (set! entries (cdr entries))
       (cdr result)))
    (else
     (let loop ((prev entries) (current (cdr entries)))
       (cond ((null? current) #f)
     ((equ key (caar current))
      (set-cdr! prev (cdr current))
      (cdar current))
     (else (loop current (cdr current)))))))))
```

```
    (lambda (msg)
      (case msg
((GET)
 (lambda (key)
   (let ((result (lookup key entries)))
     (and result (cdr result)))))
((PUT)
 (lambda (key value)
   (let ((result (lookup key entries)))
     (if result
 (set-cdr! result (cons value (cdr result)))
 (set! entries (cons (list key value) entries))))))
((REPLACE)
 (lambda (key value)
   (let ((result (lookup key entries)))
     (if result
 (set-cdr! result (list value))
 (set! entries (cons (list key value) entries))))))
((REMOVE)
 (lambda (key)
   (delete-key! key)))
((REMOVE-ONE)
 (lambda (key)
   (let ((entry (lookup key entries)))
     (and entry
  (let ((result (cadr entry))
(new-lst (cddr entry)))
    (if (null? new-lst)
(delete-key! key)
(set-cdr! entry new-lst))
    result)))))
(else (error "Unknown Table directive"))))))

(define (get table key)
  ((table 'get) key))

(define (put table key val)
  ((table 'put) key val))

(define (replace table key val)
  ((table 'replace) key val))

(define (remove-one table key)
  ((table 'remove-one) key))

(define (remove table key)
  ((table 'remove) key))
|#
;;; Agendas
#|
(define *agenda* (make-empty-queue 'point-agenda))

(define (reset-agenda)
  (set! *agenda* (make-empty-queue 'point-agenda)))

(define (fork thunk . rest-thunks)
  (for-each (lambda (thunk)
      (queue/add! *agenda* thunk))
    (cons thunk rest-thunks)))
```

```
(define (process-next-point)
  (if (queue/empty? *agenda*)
      'done
      ((queue/remove! *agenda*))))
|#

;; new event scheduled before kth event already on agenda according to
;; geometric distribution. i.e Pr(k) = p(1-p)^(k-1)
;; ie. priority-gauge = (1-p)
;; So priority-gauge = 0 ==> new event always at front
;;    priority-gauge = 1 ==> new event always at back

(define priority-gauge 1.)

(define *agenda* (list 'agenda))

(define (reset-agenda)
  (set! *agenda* (list 'agenda)))

(define-integrable (agenda.empty? agenda)
  (null? (cdr agenda)))

(define (fork thunk . rest-thunks)
  (set-cdr! *agenda*
    (random-merge priority-gauge
  (cons thunk rest-thunks)
  (cdr *agenda*))))

(define (process-next-point)
  (if (agenda.empty? *agenda*)
      'done
      (let ((thunk (cadr *agenda*)))
(set-cdr! *agenda* (cddr *agenda*))
(thunk))))

(define (random-merge bias lst1 lst2)
  (cond ((null? lst1) lst2)
((null? lst2) lst1)
((> (random 1.) bias)
 (cons (car lst1) (random-merge bias (cdr lst1) lst2)))
(else
 (cons (car lst2) (random-merge bias lst1 (cdr lst2))))))

;;; Code for tropism compilation
#|
(define (find-similar predicate elt lst)
  (cond ((null? lst) '())
((predicate elt (car lst)) '())
((predicate (car lst) elt)
 (find-similar predicate elt (cdr lst)))
(else
 (cons (car lst)
       (find-similar predicate elt (cdr lst))))))

(define (find-near comparator value ref lst)
  (cond ((null? lst) #f)
((comparator value (ref (car lst))) (car lst))
(else
```

```
 (let loop ((first (car lst)) (rest (cdr lst)))
   (if (null? rest)
       first
       (let ((second (car rest)))
 (cond ((comparator (ref second) value)
(loop second (cdr rest)))
       ((comparator (ref first) value) second)
       (else first)))))))
|#
;;; Code for pheromone level combination

(define (convert-results result-list)
  (g-map (lambda (result-pair)
   (cons (car result-pair)
 (list->vector (cdr result-pair))))
 result-list))

(define (compute-my-result target-names)
  (list->vector
   (g-map (lambda (name)
    (lookup-pheromone pheromone-table name
      (lambda (level) level)
      (lambda (locks) #f)))
  target-names)))

(define (missing-p? my-result)
  (let ((results (cdr my-result)))
    (let loop ((i (- (vector-length results) 1)))
      (if (>= i 0)
  (begin
    (if (vector-ref results i)
(loop (- i 1))
i))
  #f))))

#|
(define (check-my-result my-result)
  (for-all? my-result
    (lambda (pair) (cdr pair))))
|#

(define (pick-exp-random lst)
  ;; returns an element from lst with exponential PMF with base .5
  (cond ((null? lst) #f)
((null? (cdr lst)) (car lst))
((> (random 1.) .5) (car lst))
(else (pick-exp-random (cdr lst)))))

;;; Code for depositing material
#|
(define (is-material? material)
  (let ((material-list (get-var 'material)))
    (and material-list
 (subset? material material-list))))
|#
(define (material-type+ material)
  (let ((material-list (get-var 'material)))
    (cond ((not material-list)
   (set-var 'material (list material)))
```

```
  ((member material material-list)
   #f)
  (else
   (set-var 'material (cons material material-list))))
    (colour-material (get-var 'material))))

(define (colour-material material-list)
  (let ((colour-table (get-var 'colour-table)))
    (if colour-table
(let ((colour? (assm material-list colour-table)))
  (if colour?
      (color-me (cadr colour?)))))))

(define (assm el-list list-assocs)
  (cond ((null? list-assocs) #f)
((subset? (caar list-assocs) el-list)
 (car list-assocs))
(else (assm el-list (cdr list-assocs)))))

(define (subset? lst1 lst2)
  (cond ((null? lst1) #t)
((null? lst2) #f)
(else (and (memq (car lst1) lst2)
   (subset? (cdr lst1) lst2)))))

;;; Stable Insertion sort.  Result lists are small, so it's OK.

(define (g-sort lst pred)
  (define (insert! elt sorted-lst)
    (if (pair? sorted-lst)
(if (pred elt (car sorted-lst))
    (cons elt sorted-lst)
    (let loop ((prev sorted-lst) (rest (cdr sorted-lst)))
      (if (or (null? rest) (pred elt (car rest)))
  (begin
    (set-cdr! prev (cons elt rest))
    sorted-lst)
  (loop (cdr prev) (cdr rest)))))
(cons elt '())))

  (if (pair? lst)
      (let loop ((unsorted (cdr lst)) (sorted (list (car lst))))
(if (null? unsorted)
    sorted
    (loop (cdr unsorted) (insert! (car unsorted) sorted))))
      '()))

(define (g-filter predicate lst)
  (cond ((null? lst) '())
((predicate (car lst))
 (cons (car lst) (g-filter predicate (cdr lst))))
(else (g-filter predicate (cdr lst)))))

(define (union lst1 lst2)
  (cond ((null? lst1) lst2)
((member (car lst1) lst2)
 (union (cdr lst1) lst2))
(else
 (union (cdr lst1) (cons (car lst1) lst2)))))
```

```
(define (diff lst1 lst2)
  (cond ((null? lst1) '())
((member (car lst1) lst2)
 (diff (cdr lst1) lst2))
(else
 (cons (car lst1) (diff (cdr lst1) lst2)))))

(define (flatten lst)
  (cond ((null? lst) '())
((pair? (car lst))
 (append (car lst) (flatten (cdr lst))))
(else
 (cons (car lst) (flatten (cdr lst))))))

(define (flat-map p . args)
  (fold-right append '() (apply map p args)))

(define (every-other lst)
  (cond ((null? lst) '())
((null? (cdr lst)) lst)
(else (cons (car lst) (every-other (cddr lst))))))

(define (between? lo hi v)
  (or (<= lo v hi) (>= lo v hi)))

(define (compose f g)
  (lambda (x) (f (g x))))

(define (for-each-index start stop inc p)
  (if (>= inc 0)
      (let loop ((i start))
(if (<= start i stop)
    (begin
      (p i)
      (loop (+ i inc)))))
      (let loop ((i start))
(if (>= start i stop)
    (begin
      (p i)
      (loop (+ i inc)))))))

(define (map-indexes start stop inc p)
  (if (>= inc 0)
      (let loop ((i start))
(if  (<= start i stop)
     (cons (p i)
   (loop (+ i inc)))
     '()))
      (let loop ((i start))
(if  (>= start i stop)
     (cons (p i)
   (loop (+ i inc)))
     '()))))

(define (vector:for-each p v1 . vs)
  (let ((n (vector-length v1)))
    (cond ((null? vs)
   (for-each-vector-element v1 p))
```

```
  ((null? (cdr vs))
   (let ((v2 (car vs)))
     (let loop ((i 0))
       (if (< i n)
   (begin
     (p (vector-ref v1 i) (vector-ref v2 i))
     (loop (fix:+ i 1)))))))
  ((null? (cddr vs))
   (let ((v2 (car vs))
 (v3 (cadr vs)))
     (let loop ((i 0))
       (if (< i n)
   (begin
     (p (vector-ref v1 i) (vector-ref v2 i)
(vector-ref v3 i))
     (loop (fix:+ i 1)))))))
  (else
   (let loop ((i 0))
     (if (< i n)
 (begin
   (apply p
  (vector-ref v1 i)
  (map (lambda (v*) (vector-ref v* i))
       vs))
   (loop (fix:+ i 1)))))))))

(define (map-vector f v1 . vs)
  (let ((n (vector-length v1)))
    (let ((result (make-vector n #f)))
      (cond ((null? vs)
     (vector-map v1 f))
    ((null? (cdr vs))
     (let ((v2 (car vs)))
       (let loop ((i 0))
 (if (< i n)
     (begin
       (vector-set! result i
    (f (vector-ref v1 i)
       (vector-ref v2 i)))
       (loop (fix:+ i 1)))
     result))))
    ((null? (cddr vs))
     (let ((v2 (car vs))
   (v3 (cadr vs)))
       (let loop ((i 0))
 (if (< i n)
     (begin
       (vector-set! result i
    (f (vector-ref v1 i)
       (vector-ref v2 i)
       (vector-ref v3 i)))
       (loop (fix:+ i 1)))
     result))))
    (else
     (let loop ((i 0))
       (if (< i n)
   (begin
     (vector-set!
      result i
```

```
      (apply f
     (vector-ref v1 i)
     (map (lambda (v*) (vector-ref v* i))
  vs)))
     (loop (fix:+ i 1)))
   result)))))))

(define (vector-acc combiner base vec)
  (let loop ((i (- (vector-length vec) 1)) (result base))
    (if (< i 0)
result
(loop (- i 1) (combiner (vector-ref vec i) result)))))

(define (flo:list->vector lst)
  ;; converts a list to a floating vector
  (let ((len (length lst)))
    (let ((result (flo:vector-cons len)))
      (let loop ((i 0) (lst lst))
(if (null? lst)
    result
    (begin
      (flo:vector-set! result i (exact->inexact (car lst)))
      (loop (+ i 1) (cdr lst))))))))

(define (find-min&max v k)
  (let ((len (vector-length v)))
    (let loop ((i 1)
       (lo (vector-ref v 0))
       (hi (vector-ref v 0)))
      (if (= i len)
  (k lo hi)
  (loop (+ i 1)
(min lo (vector-ref v i))
(max hi (vector-ref v i)))))))

(define (permute lst)
  (let ((vec (list->vector lst)))
    (let ((size (vector-length vec)))
      (let loop ((i 0))
(if (< i size)
    (let ((j (random size)))
      (let ((tmp (vector-ref vec j)))
(vector-set! vec j (vector-ref vec i))
(vector-set! vec i tmp)
(loop (+ i 1))))
    (vector->list vec))))))

;;;; Display related utilities

(define (make-window width height #!optional coords)
  (let ((type (graphics-type-name (graphics-type #f))))
    (let ((w
   (case type
     ((X) (make-graphics-device
   'x #f
   (x-geometry-string #f #f width height)))
     ((WIN32) (make-graphics-device 'win32 width height))
     (else (error "Unsupported graphics type: " type)))))
      (graphics-operation w 'set-foreground-color "black")
```

```
      (graphics-operation w 'set-background-color "#cccccc")
      (graphics-clear w)
      (if (not (default-object? coords))
  (apply graphics-set-coordinate-limits w coords))
      w)))

(define (pixels->units win)
  (with-values
      (lambda () (graphics-device-coordinate-limits win))
    (lambda (left bot right top)
      (with-values
  (lambda () (graphics-coordinate-limits win))
(lambda (xlo ylo xhi yhi)
  (values (/ (- xhi xlo) (+ (- right left) 1))
  (/ (- yhi ylo) (+ (- bot top) 1))))))))

(define (mark-+ size)
  ;; mark point (x,y) with a '+' of given size in window win.
  ;; size is in pixels
  (lambda (win)
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((xsize (* .5 (* size x-units)))
      (ysize (* .5 (* size y-units))))
  (lambda (x y)
    (graphics-draw-line win (- x xsize) y (+ x xsize) y)
    (graphics-draw-line win x (- y ysize) x (+ y ysize)))))))))

(define (mark-x size)
  ;; mark point (x,y) with an 'x' of given size in window win.
  ;; size is in pixels
  (lambda (win)
    (with-values
(lambda () (pixels->units win))
      (lambda (x-units y-units)
(let ((xsize (* .5 (* size x-units)))
      (ysize (* .5 (* size y-units))))
  (lambda (x y)
    (graphics-draw-line win (- x xsize) (- y ysize)
(+ x xsize) (+ y ysize))
    (graphics-draw-line win (- x xsize) (+ y ysize)
(+ x xsize) (- y ysize)))))))))

(define (number->grey v vmin vmax)
  ;; assume 256 levels of grey
  (if (= vmax vmin)
      "black"
      (let ((intensity (min (max 0
 (round->exact (/ (* 256 (- v vmin))
  (- vmax vmin))))
    255)))
(let ((int-str (string-pad-left (number->string intensity 16)
2 #\0)))
  (string-append "#" int-str int-str int-str)))))

;;; Procedures to generate list of coordinates of a disc of specified radius.

(define (make-fill-quadrant-coords size)
```

```
  ;; returns a vector of y coordinates.  All values should be integers.
  ;; vector is indexed by x coordinate.
  (let ((xs (make-vector size #f)))
    (let x-loop ((x 0))
      (if (= x size)
  xs
  (let y-loop ((y 0)
       (yhi (round->exact (sqrt (- (* size size) (* x x)))))
       (ys '()))
    (if (= y yhi)
(begin
  (vector-set! xs x (reverse ys))
  (x-loop (+ x 1)))
(y-loop (+ y 1) yhi (cons y ys))))))))

(define (fill-circle-coords radius)
  ;; x, y, size are in pixels
  ;; returns a list of pairs of numbers (x . y).  They correspond to
  ;; the coordinates that ought to be filled to visibly draw a representation
  ;; for the point (x,y)
  (let* ((diameter (round->exact (+ (* 2 radius) 1)))
 (size (ceiling->exact (/ diameter 2)))
 (quadrant (make-fill-quadrant-coords size)))
    (cond ((= diameter 1) (list (cons 0 0)))
  ((odd? diameter)
   (append
    (list (cons 0 0))
    (flatten
     (map (lambda (y) (list (cons 0 y) (cons 0 (- y))))
  (cdr (vector-ref quadrant 0))))
    (flatten
     (map-indexes 1 (- size 1) 1
  (lambda (index)
    (list (cons index 0) (cons (- index) 0)))))
    (flatten
     (map-indexes 1 (- size 1) 1
  (lambda (index)
    (flatten
     (map (lambda (y)
    (list (cons index y)
  (cons (- index) y)
  (cons index (- y))
  (cons (- index) (- y))))
  (cdr (vector-ref quadrant index))))))))
  (else
   (flatten
    (map-indexes 0 (- size 1) 1
 (lambda (index)
   (flatten
    (map (lambda (y)
   (list (cons index y)
 (cons (- -1 index) y)
 (cons index (- -1 y))
 (cons (- -1 index) (- -1 y))))
 (vector-ref quadrant index))))))))))

(define x-color:name->rgb
  ;; returns the rgb components for an X color name
  ;; looks in /usr/X11R6/lib/X11/rgb.txt for mapping
```

```
  ;; format is <r> <g> <b>#\TAB #\TAB<name>#\newline
  (let ((rgb-file "/usr/X11R6/lib/X11/rgb.txt")
(cache '()))
    (define (search-file name)
      (with-input-from-file
  rgb-file
(lambda ()
  (read-line) ;throw away first line
  (let loop ((line (read-line)))
    (cond ((eof-object? line)
   (error "X Server unable to translate X Color" name))
  ((eqv? (substring? "#" line) 0)
   (loop (read-line)))
  ((substring? name line)
   (let ((str-port (string->input-port line)))
     (let* ((r (read str-port))
    (g (read str-port))
    (b (read str-port)))
       ;; throw away 2 tabs
       (read-char str-port) (read-char str-port)
       (if (string=? name (read-line str-port))
   (list r g b)
   (loop (read-line))))))
  (else
   (loop (read-line))))))))
    (lambda (name)
      (let ((cache-result (assoc name cache)))
(if cache-result
    (cadr cache-result)
    (let ((result (search-file name)))
      (set! cache (cons (list name result) cache))
      result))))))

(define (x-color:hex->rgb hex-string)
  (let ((r (string->number (substring hex-string 1 3) 16))
(g (string->number (substring hex-string 3 5) 16))
(b (string->number (substring hex-string 5 7) 16)))
    (list r g b)))

(define (x-color->rgb colour)
  (if (not (string? colour))
      (error "X Colors must be specified as strings" colour))
  (if (eqv? (substring? "#" colour) 0) ; check if starts with # character
      (x-color:hex->rgb colour)
      (x-color:name->rgb colour)))

;;; Routines for compatibility with ECOLI implementation

(define-integrable make-entry cons)
(define-integrable entry/tag car)
(define-integrable entry/value cdr)
```

**point-set.scm**

```
;;; -*- Scheme -*-

(declare (usual-integrations))

(define vector->set vector->list)
(define (set->list s) s)
(define elt->set list)
(define empty-set '())

(define (set-elt? elt s)
  (memv elt s))

(define (set+ s1 s2)
  (cond ((null? s1) s2)
((set-elt? (car s1) s2)
 (set+ (cdr s1) s2))
(else (set+ (cdr s1) (cons (car s1) s2)))))

(define (set- s1 s2)
  (cond ((null? s1) '())
((set-elt? (car s1) s2)
 (set- (cdr s1) s2))
(else (cons (car s1) (set- (cdr s1) s2)))))
```

# Bibliography


[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs, 2nd ed.* MIT Press, Cambridge, MA, 1996.

[2] Stephen Adams. A high level simulator for gunk. *Internal Publication, Amorphous Computing Project*, October 1997.

[3] Lars Valerian Ahlfors. *Complex Analysis: An Introduction to the Theory of Analytic Functions of One Complex Variable.* McGraw Hill, 1979.

[4] Michael Artin. *Algebra.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[5] F. Aurenhammer. Power diagrams: Properties, algorithms and applications. *SIAM Journal of Computing*, 16(3):78–96, 1987.

[6] Yaneer Bar-Yam. *Dynamics of Complex Systems.* Addison-Wesley, Reading, MA, 1997.

[7] Andrew A. Berlin. *Towards Intelligent Structures: Active Control of Buckling.* PhD thesis, MIT, May 1994.

[8] David Betounes. *Partial Differential Equations for computational science: with Maple and vector analysis.* Springer-Verlag, New York, 1997.

[9] Carl B. Boyer and Uta C. Merzbach. *A History of Mathematics.* John Wiley & Sons, Inc, New York, 1991.

[10] Chandler. Calculation of number of relay hops required in randomly locatied radio network.". *Electronics Letters*, 25(24), Nov 1989.

[11] B. Chopard and M. Droz. Cellular automata model for the diffusion equation. *Journal of Statistical Physics*, 64(3/4):859–892, Aug 1991.

[12] Geoffrey M. Cooper. *The Cell: A Molecular Approach.* Sinauer Associates, Sunderland, MA, 1997.

[13] Daniel Coore. Establishing a coordinate system on an amorphous computer. MIT/LCS/TR 737, MIT, 545 Technology Sq., Cambridge, MA 02139, January 1998. Proceedings of 1998 MIT Student Workshop on High-Performance Computing in Science and Engineering.

[14] Daniel Coore and Radhika Nagpal. Implementing reaction-diffusion on an amorphous computer. MIT/LCS/TR 737, MIT, 545 Technology Sq., Cambridge, MA 02139, January 1998. Proceedings of 1998 MIT Student Workshop on High-Performance Computing in Science and Engineering.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill Book Company, Cambrideg, MA; New York, 1990.

[16] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[17] Herbert Edelsbrunner. The union of balls and its dual shape. In *Proceedings of the 9th Annual Symposium on Computational Geometry*, pages 218–231, California, May 1993. ACM.

[18] Stephanie Forrest. Emergent computation: Self-organizing, collective, and cooperative phenomena in natural and artificial computing networks. *Physica D*, 42:1–11, 1990.

[19] John Gerhart and Marc Kirschner. *Cells, Embryos, and Evolution: Toward a Cellular and Developmental Understanding of Phenotypic Variation and Evolutionary Adaptability*. Blackwell Science, Malden, MA, 1997.

[20] David Gifford and Franklyn Turbak. 6.821 book draft. MIT Internal Publication for Programming Languages course, September 1994.

[21] S. R. Hall, E. F. Crawley, J. P. How, and B. Ward. Hierarchic control architecture for intelligent structures. *Journal of Guidance, Control and Dynamics*, 14(3), May-June 1991.

[22] Chris Hanson. MIT Scheme reference manual. MIT/AI/TR 1281, MIT, 545 Technology Sq., Cambridge, MA 02139, January 1991.

[23] John H. Holland. *Emergence: From Chaos to Order*. Addison-Wesley, Reading, MA, 1997.

[24] J. P. How and S. R. Hall. Local control design methodologies for a hierarchic control architecture. *Journal of Guidance, Control and Dynamics*, 15(3), May-June 1992.

[25] Limin Hu. Topology control for multihop packet radio networks. *IEEE Transactions on Communications*, 41(10):1474–1481, October 1993.

[26] A. Kalay, H. Parnas, and E. Shamir. Neuronal growth via hybrid system of self-growing and diffusion based grammar rules: I. *Bulletin of Mathematical Biology*, 57:205–227, 1995.

[27] M. Katchalski and H. Last. On geometric graphs with no two edges in convex position. *Discrete & Computational Geometry*, 19(3):399–404, 1998.

[28] L. Kleinrock and J. Silvester. Optimum transmission radii for packet radio networks or why six is a magic number. In *Proceedings of the National Telecommunications Conference*, pages 4.3.1–4.3.5, New York, 1978.

[29] Regina Klimmek and Frank Wagner. A simple hypergraph min cut algorithm. TR B 96-02, Universität Berlin, Germany, March 1996.

[30] Y. S. Kupitz and M.A. Parles. Extremal theory for convex matchings in convex geometric graphs. *Discrete and Computational Geometry*, 15:195–220, 1996.

[31] Leslie Lamport and Nancy Lynch. Distribute computing: Models and methods. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, II*, pages 1157–1199. Elsevier Science Publishers B.V., Amsterdam, 1990.

[32] K. J. Lee, W. D. McCormick, Q. Ouyang, and H. L. Swinney. Pattern formation by interacting chemical fronts. *Science*, 261:192–194, Jul 1993.

[33] Kevin K. Lin. Coordinate independent computations on differential equations. Master's thesis, MIT, August 1997.

[34] Nancy Lynch. *Distributed Algorithms.* Morgan Kauffman Publishers, San Francisco, CA, 1994.

[35] C. R. F. Maunder. *Algebraic Topology.* Dover, New York, 1996.

[36] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, MA, 1988.

[37] Patrick Moran and Marcus Wagner. Introducing alpha shapes for the analysis of path integral Monte Carlo results. In *Proceedings of Visualization*, pages 52–59, 1994.

[38] Randolph Nelson. *Channel Access Protocols for Multi-Hop Broadcast Packet Radio Networks.* PhD thesis, UCLA, July 1982.

[39] J. Pach and P. K. Agarwal. *Combinatorial Geometry.* Wiley Interscience, New York, 1995.

[40] J. Pach, F. shahrokhi, and M. Szegedy. Applications of the crossing number. *Algorithmica*, 261:111–117, 1996.

[41] Roger. H. Pain. *Mechanisms of Protein Folding.* Oxford University Press, Inc, New York, 1994.

[42] J. E. Pearson. Complex patterns in a simple system. *Science*, 261:189–192, Jul 1993.

[43] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer Verlag, New York, 1990.

[44] F. Shahrokhi, L. A. Szekely, and I. Vrt'o. Drawings of graphs on surfaces with few crossings. *Algorithmica*, 16:118–131, Sep 1996.

[45] Steven S. Skiena. *The Algorithm Design Manual.* Springer Verlag, New York, 1998.

[46] J. M. W. Slack. *From Egg to Embryo: Regional Specification in Early Development, 2nd ed.* Cambridge University Press, Cambridge, UK, 1991.

[47] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1988.

[48] Jr. Thomas F. Knight and Gerald J. Sussman. Cellular gate technology. In *Proceedings of the First International Conference on Unconventional Models of Computation*, Auckland, NZ, Jan 1998.

[49] M. Q. Vahidi-Asl and J. C. Wierman. First passage percolation on the voronoi tessellation and delaunay triangulation. In *Random Graphs. Proceedings of the 1987 Random Graphs Conference on Poznán, Poland.*, pages 341–359, New York, 1990.

[50] M. Q. Vahidi-Asl and J. C. Wierman. A shape result for the first-passage percolation on the voronoi tessellation and delaunay triangulation. In *Random Graphs. Volume 2*, pages 247–263, New York, 1992.

[51] P. Valtr. On geometric graphs with no $k$ pairwise parallel edges. *Discrete & Computational Geometry*, 19(3):461–469, 1998.

[52] Ron Weiss, George Homsy, and Thomas F. Knight. Toward in vivo digital circuits. In *Dimacs Workshop on Evolution as Computation*, Princeton, NJ, Jan 1999. DIMACS.

[53] Arthur T. White. *Graphs, Groups and Surfaces.* North-Holland, Amsterdam, Holland, 1973.

[54] S. Wolfram. Computation theory of cellular automata. *Communications in Mathematical Physics*, 96:15–57, 1984.

[55] Kaizhi Yue and Ken A. Dill. Inverse protein folding: Designing polymer sequences. *Proceedings of the National Academy of Sciences, USA*, 89:4163–4167, May 1992.