Host Plant Selection by Turnip Root Flies

Deborah Bellotti

EPCC–SS94–05

Host Plant Selection by Turnip Root Flies

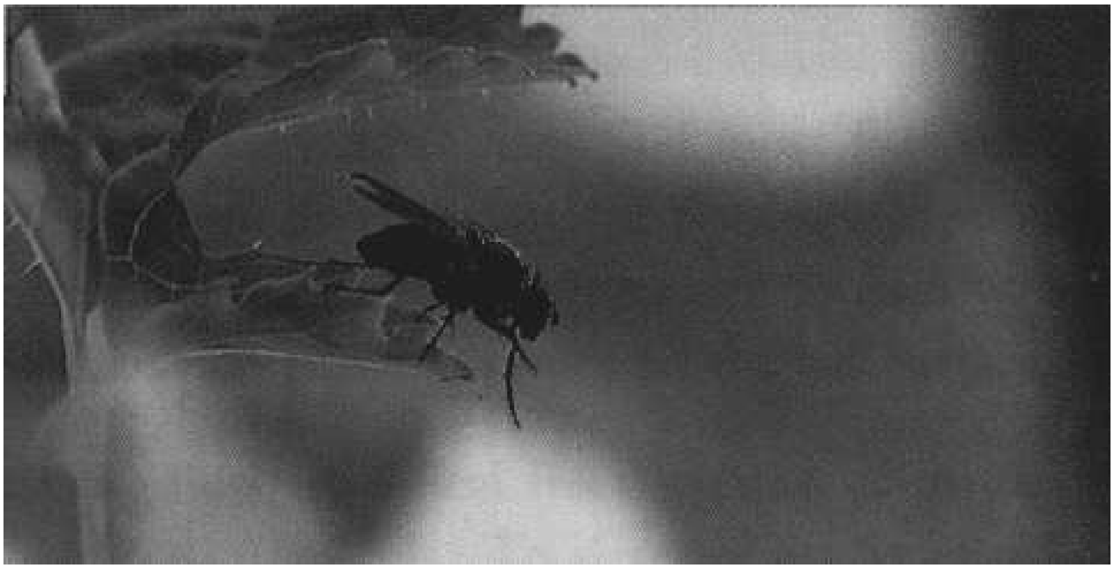
Deborah Bellotti

EPCC–SS94–05

September 1994

Abstract

Turnip root flies are plant feeding insects. The adult fly is highly mobile and oviposits (lays eggs) at the base of host plants. The eggs hatch and the larvae then feed on the roots of the brassica plants concerned. By modelling the behaviour of a series of individuals we hope to assess how the relative proportion of resistant and susceptible host plants affect the damage sustained by a crop during the course of a growing season.



1 Introduction

The title of this report highlights the agricultural nature of the problem that has been considered. Host plant selection by turnip and cabbage root flies involves the study of flies' behaviour in relation to the crop they live in. Biologists gather such behavioural information by direct observation of the population both in the field and in laboratories. From these observations, statistical representations of the flies's behaviour are produced.

Particular interest is given to the effects of flies' oviposition on a host plant and, as a result, a large amount of data describing these effects and a fly's oviposition behaviour exist. Large numbers of eggs laid on a plant will result in excessive larval feeding on the roots thus reducing yield and possibly resulting in the death of the plant. This can also lead to the spread of an insect pest across a crop causing the destruction of many plants. With a deeper understanding of flies' oviposition site selection and oviposition, a considerable amount of damage to such crops could possibly be prevented.

Hence the goal of this project : a simulation of the fly population in a field may provide a way of investigating how changes in the arrangement of plants may affect the proportion of plants damaged.

2 This Report

In this report, we discuss the following aspects of this project.

- Turnip and cabbage root fly behaviour - the biological nature of the turnip and cabbage fly behaviour.
- Fly behaviour simulation - why is this simulation useful and what features of the real system have been considered in this project.
- Implementation of the simulation - how the project has been designed and implemented
- Parallel decomposition - a detailed description of the parallel simulation
- Gathering and analysing results - what kind of results we are interested in and how their validity is proved
- Future work - a list of some of the possible enhancements of the current project

3 Turnip and Cabbage Root Fly Behaviour

Delia floralis, the turnip root fly, and *Delia radicum*, the cabbage root fly, are plant feeding insects. The site of larval development is dictated by the gravid females which are highly mobile and capable of powered flight over moderate distances. The search that an insect undertakes for an oviposition site or a food item can be categorised in a number of different ways. The most widely accepted is to divide the searching behaviour into habitat-finding, host finding, host recognition, host acceptance and host suitability. However the functional boundaries between many of these stages are difficult to distinguish and there are advantages to simply classifying them into **pre-landing** and **post-landing** behaviour [Jones, 1992].

3.1 Pre-landing behaviour

Pre-alighting behaviour of cabbage and turnip root flies has been extensively studied and it has been demostrated that only gravid females move into a host crop [Hawkes, 1976] which suggests that flies are mostly stimulated to move according to their egg load. Their flights are heavily influenced by wind and Ellis [Ellis, 1980] has also shown that there is a strong link between host preference and plant odour. In fact, gravid females in odour clouds largely move with straight fast upwind flights, in contrast with discrete plumes in which flights are slower and change direction more frequently [Nottingham, 1987].

Once a plant is found, the fly lands on it at which point host selection starts, that is the fly analyses the plant in order to decide if it is suitable for ovipositing. This selection process is the first step in the fly's post-landing behaviour.

3.2 Post-landing behaviour

The published data on post-alighting behaviour can be summarised as follows:

detection of the leaf surface the fly extends its proboscis onto the leaf surface; although the reason for the contact with the plant surface remains unknown at the present, it is clear that the fly tests the surface using "taste" sensors or feet [Stadler, 1978].

walking down if the female accepts the potential host after leaf surface analysis, it proceeds down the leaf and stem. After carrying out a circumvective walk around the base of the plant it may dig in the soil and further probing may take place.

oviposition if the female reaches this stage with all necessary stimuli, and no strong deterrents have been encountered, ovipositon takes place [Prokopy, 1983; Havukkala, 1987; Stadler, 1990].

4 The Fly Behaviour Simulation

Before describing our simulation in details it is essential to underline the usefulness of performing such a simulation. Modelling the biological system described above can be seen to be useful in several ways. By creating a fairly realistic model, the problem can be better analysed and large amounts of data can be handled cheaply and quickly which would be impossible using traditional field and laboratory methods. In particular, it is very useful to know the biological state of the plants in a crop at the end or during a simulation, or to compare data gathered after having tried varying dispositions of different kinds of vegetables in a field.

This simulation has been developed considering a model that describes the real system, although some aspects of it have been simplified to make the design more tractable. Our model consists of the following features:

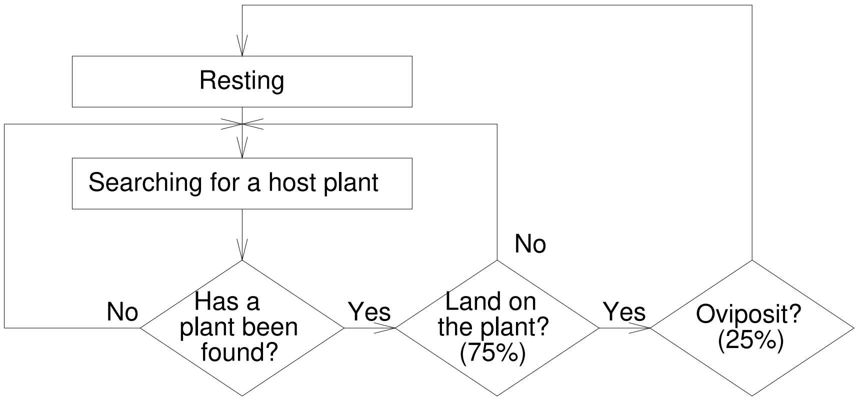


Figure 1: A graphic simple description of the fly's behaviour

pre-landing behaviour the fly's flight has been described in terms only of wind influence and with no regard to plant odour clouds in the system ; landing choice is determined probabilistically

post-landing behaviour as this part of the behaviour involves many details to be implemented, most of the stages described in it have been grouped in one bigger phase that tries to include all of them; therefore, for the time being, this is just expressed by a probability

Figure 1 depicts the very simple behavioural model implemented for the turnip root flies in our model. Notice that this is an extremely simplified model, but it captures the notion of pre- and post-landing behaviours.

5 Implementation Overview

When deciding how to design the simulation system for the turnip root fly model we identified two possible approaches:

- consider **only** this biological system, model it creating a simulating engine which can work **only** on this particular model
- consider this biological system as a member of a bigger set, say SET, model it, and create a simulating engine whose environment can accept every model of a system in SET

The former option focuses on the sentence **fly's behaviour simulation** whereas the second focuses on the word **simulation** and the bigger set mentioned above is:

SET = { biological system s | s defined by interactions between populations }

We decided that, in the interests of producing something which could be extended in many directions and which could support many different models, we would design a general purpose modelling system. Therefore our system has had to be designed in order to achieve the aim of providing generality across models and thus the work has been split into two main blocks:

- the model
- the modeller

Figure 2 shows the system design for this project. The model is written using functions from a modeller library which is a repository for useful modelling tools and simulation methods. A model control program is required to drive the simulation.

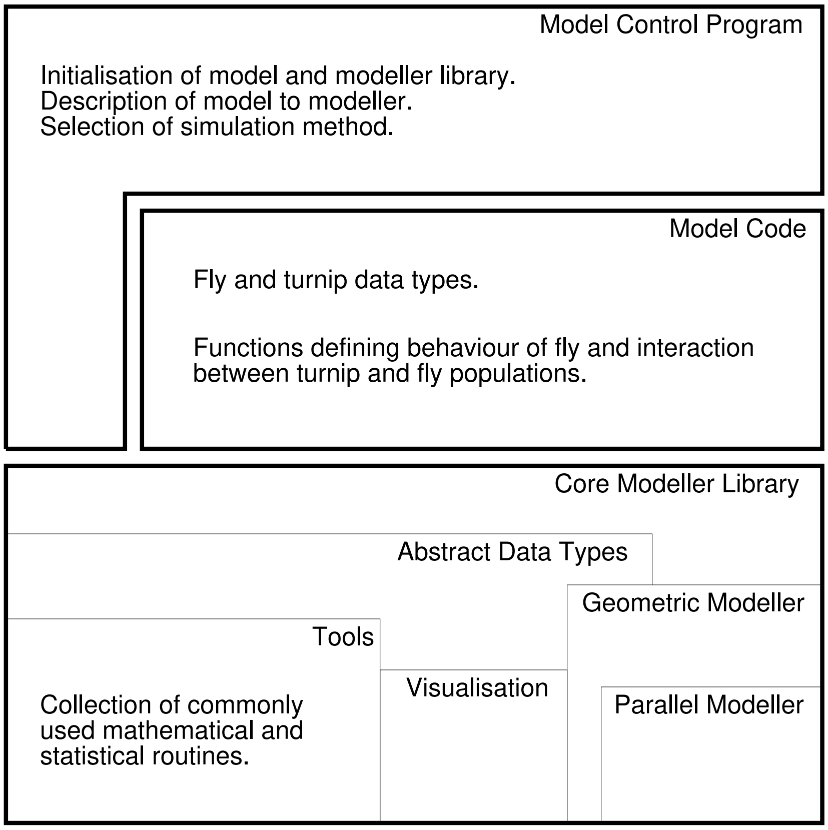


Figure 2: The system design

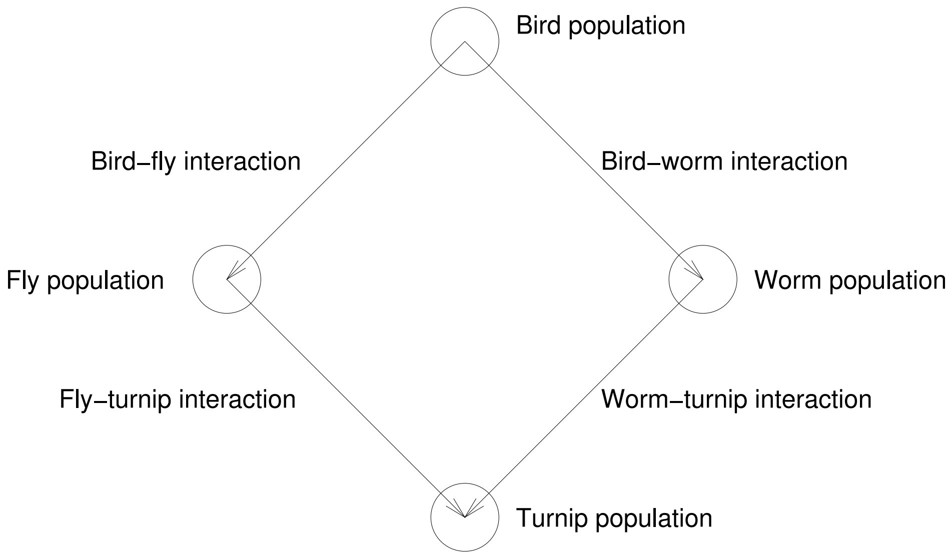


Figure 3: An interaction graph

6 Basic Modeller

The modeller has been designed to provide maximum flexibility to the model writer. Since the modeller must not depend on any particular model, a model has to be described in a general way. All systems in SET are different, but each one involves interactions between arbitrary entities, or rather populations of entities, that can be different from any other one in this set; but all consist of **interactions** and **populations**. The implementation of the modeller has been based on this common feature of systems and hence between models which describe them.

To emphasise how every system can be described in term of interactions and populations, we can consider an interaction graph, where its nodes represent populations and its edges interactions. The interaction graph in figure 3, for example, shows a biological system with four populations and four interactions. Given this representation, we can describe the modeller's implementation.

An interaction is composed of three items. There are two populations, one of which is deemed to be *active*, the other being *passive*, and there is a description of the method by which the two populations interact and behave in relation to each other.

6.1 Describing a model

The modeller provides functions to build a representation of this interaction graph in terms of populations and interactions. A model is represented by a `Model` structure which is composed of a number of `Interaction` structures which in turn are composed of two `Population` structures and two callback functions which encapsulate the interaction and behaviour processes defined between the two populations.

As far as the modeller library is concerned, the population data is completely opaque. A basic population is defined in terms of the size required to represent a single entity and the number of elements in the population. We will see later that other types of population may require more information, but this suffices for the basic model implementation.

To summarise, we have the following structures:

Population stores the size of each population entity and the number thereof.

Interaction stores references to two populations along with a function to interact an entity from the active population with one from the passive population and a function to perform the behaviour of an active element after all the necessary interactions have taken place.

Model stores a list of interactions which define the model.

Using this representation, we can build a model from any interaction graph and hence we can simulate any such model. It should be noted also that we do not forbid interactions where the active and passive elements are the same population.

6.2 Running a model

Once all the model information has been specified, the modeller can simulate the model by simply performing each of the interactions specified. In the basic modeller, a population-population interaction is simulated by calling the interaction callback function for every pair of entities (one from the active and one from the passive population) then calling the behavioural function for each member of the active population. In this way an all-with-all interaction is performed:

```
population_do_interaction ( Interaction inter )
{
    for each element i of the active population in inter
    {
       for each element j of the passive population in inter
         interaction_function(element i, element j)

       behaviour_function(element i)
    }
}
```

7 Geometric Modeller

Clearly, in many cases, the basic simulation method discussed in the previous section will be unnecessarily inefficient since in many models which incorporate spatial information (such as our turnip root fly model) many of the interactions performed by this function will be redundant. For example, in our model, a fly which is two miles away from a turnip will never interact in a meaningful manner with that turnip. This locality is depicted in figure 4.

We would like to take advantage of this locality to speed up the simulation and thereby enable simulation of larger populations. In order to retain generality in the modeller interface, however, we must ensure that the geometric information can be described in a manner which is completely independent of any particular model. We therefore extend our modeller to efficiently support a particular class of interactions with the following features:

- the active and passive populations have position in a coordinate system
- the active population moves within the coordinate system
- the passive population remains stationary
- the active population has a limited range within which it can interact with members of the passive population

To support this class of interactions, we extend the `Interaction` and `Population` structures as follows:

Passive population extends the basic `Population` structure with a grid representation. The grid is described in more detail below, but basically represents an n-dimensional array of data in which each element can be addressed by a set of n coordinates.

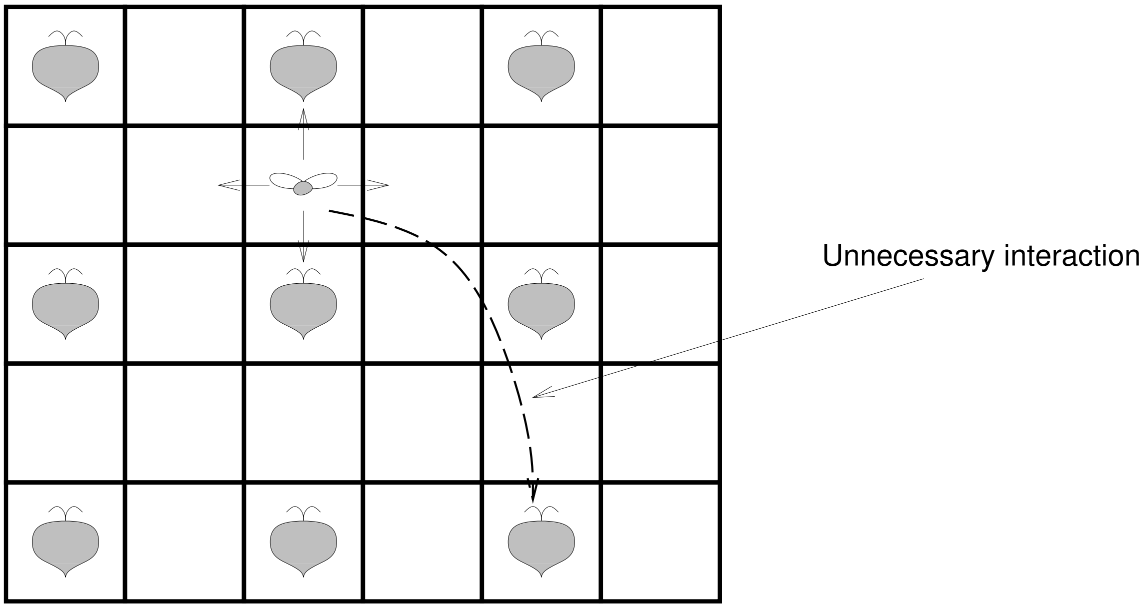


Figure 4: The world description

Active population extends the basic `Population` structure with a list based representation. Each member of the population list comprises a set of n coordinates which describe a location in the grid defined in the passive population along with a reference to a member of the population.

We also define the *search range* of the entities in the active population. This determines the distance in each dimension beyond which it makes no sense to interact with an entity from the passive population.

Interaction represents the fact that the interaction is of this particular class and that the geometric interaction scheme can be used.

7.1 The grid

We introduce the *grid* abstract data type to support these extensions in the geometric modeller. We can see from the description above that we have two basic uses for the grid type:

- to store population data (the turnips in our model)
- to define a coordinate system (for the flies in our model)

The following operations are therefore defined on the type `Grid`

grid_create : dimensions x element size –> Grid this operation allocates and initialises a new `Grid` object.

grid_lookup : Grid x coords –> element this operation retrieves the element stored at the specified coordinates.

grid_destroy : Grid –> void this operation frees any resources associated with a `Grid` object.

grid_size : Grid –> dimensions this operation retrieves the size of a grid.

In addition, we note that in our model the grid used to store the passive population (turnip) data is sparse but regularly populated — each member of the population is a fixed distance away from its neighbours. We therefore provide an operation to exploit this regularity to save space:

grid_dist_create : dimensions x distribution x element size –> Grid which allocates and initialses a new `Grid` object but stores information about the distribution of elements in the grid which can be used to allocate less space than would otherwise be required.

The `grid_lookup` operation is modified to deal with this type of grid by mapping coordinates to the appropriate entry internally. This saves the user of the grid type from being concerned about translation of coordinates.

7.2 The geometric population

In addition to a mechanism with which we can store and address information by coordinates, we require a mechanism to retain coordinate information alongside population entities but without the overhead of being able to address these entities by coordinate. This mechanism is used for the active population where all we require is the ability to map from an active entity to a set of coordinates and thus determine the range of data in the passive population within which the entity is to interact with passive entities. We therefore define the *gridpop* abstract data type with the following operations:

grid_pop_create : Grid x element size –> GridPop allocates and initialises an empty population structure in which a set of coordinates is associated with each element.

grid_pop_insert : GridPop x element x coords -> GridPop inserts a reference to an element into a population along with the coordinates of that element.

grid_pop_remove : GridPop x element -> GridPop removes an element from the population.

grid_pop_destroy : GridPop –> void releases any resources associated with the population instance.

With these operations alone, however, it is not possible to retrieve elements from the population or to traverse it as will be required by the geometric modeller. To this end we introduce the notion of a `GridPopItem` which is the encapsulation of a population element, its coordinates and information about its relationship to other population members:

grid_pop_first : GridPop -> GridPopItem retrieves the first item stored in the population.

grid_pop_next : GridPopItem -> GridPopItem given one population item, this operation returns the next.

grid_pop_data : GridPopItem -> element x coords extracts the useful information from an item description.

grid_pop_delete : GridPop x GridPopItem -> GridPop a potentially more efficient mechanism for the removal of an item from a population.

We shall see later when describing the parallel decomposition of this class of model that we must be able to dynamically insert and remove entities from an active population and thus we have chosen to implement the above operations using a linked list as opposed to a more simple array based approach. Figure 5 depicts the implementation of the *gridpop* type. Each entry in the list is doubly linked and the list structure itself has the same type as the nodes to allow for simpler insertion and deletion. A list entry contains a pointer to an structure in which a set of coordinates are stored alongside a reference to some opaque, user-level data (the `Fly` structure in our case).

At this stage it may not be immediately obvious why we manage the coordinates of each entity inside the modeller library instead of directly in the model code itself. There are several reasons for doing so, including some issues concerning the way in which the active population is handled in the parallel implementation, but our primary concern is to ensure that the modeller requires no information about the fields in the structures used by the model to describe a population entity. Since the geometric modeller requires coordinates for each active entity, this information must be stored inside the modeller in this way.

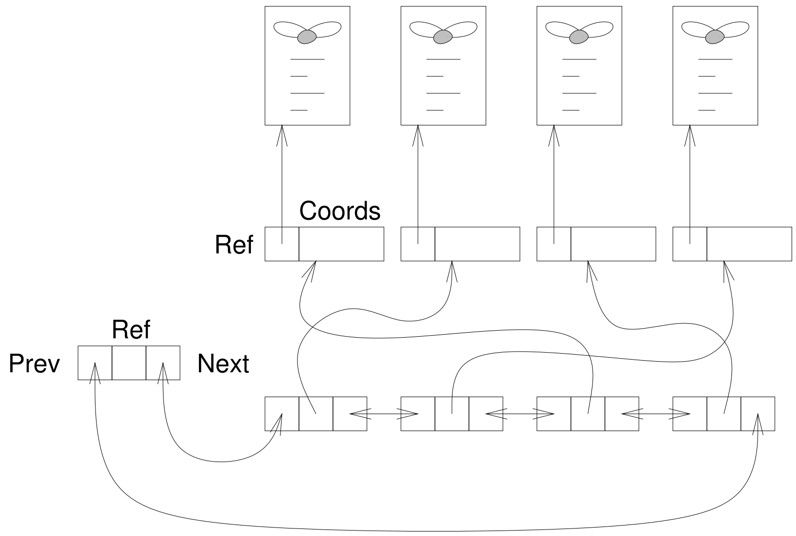


Figure 5: The world description

7.3 The Interaction

Given this representation of the geometric interaction, it only remains to describe the mechanism by which the interaction is performed. The geometric interaction function is very similar to the basic one described in the previous section:

```
population_do_grid_interaction ( Interaction inter )
{
    for each element i of the active population in inter
    {
       for each element j of the passive population in the range
        defined by the active population
       {
         interaction_function(element i, element j)
       }

       behaviour_function(element i)
    }
}
```

The inner loop of this function is now able to iterate over only those elements of the passive population which lie within the search range of the active entities. This will dramatically reduce the number of interactions which are performed and hence increase the performance of the simulation.

8 The Model

To implement our model we must represent all the behavioural and physiological information which describes the component populations of the model. In our case, we have two populations to describe with one interaction defined between them. We have therefore implemented two structures, one (`Fly`) to describe the state of a fly and the other (`Turnip`) to describe the state of a turnip. These are depicted in figure 6.

Since we wish to exploit the locality inherent in this model, we describe the fly population as a coordinate

Fly Structure

`eggLoad`	number of eggs ready for laying
`landed`	boolean flag set if the fly has landed
`nearest`	reference to nearest `Turnip` encountered so far
`nearestDistance`	distance away from nearest `Turnip`
`timeToElapse`	represents number of iterations before fly becomes active again

Turnip Structure

`eggsLaid`	Number of eggs laid on the plant

Figure 6: The model structures

list and the turnip population as a grid of `Turnip` structures. The turnip grid is defined with a distribution since the turnips are laid out in a regular fashion.

A fly may be in one of several states:

- flying
- landing
- laying an egg
- resting

We can describe the behaviour of a fly with a transition diagram or flow chart such as that shown in figure 1 and this can be implemented in our model by way of a simple function consisting of several conditional statements. Before this behavioural function can be called, however, we must interact the fly with each of the relevant turnips to determine which, if any, turnip will influence the behaviour the fly. In our case, this reduces to selecting the closest turnip from those with which a fly is interacted.

The transitions shown in figure 1 are modelled purely in terms of probabilities derived from statistical models generated from field and laboratory study of the turnip and cabbage root fly. When the fly, or more generally, when an active population member changes state according to one of these transitions, two situations can arise:

- the fly moves
- the physiological state of the fly changes

In the former case the model calls a modeller function to update the fly's coordinates. Recall that the coordinates of an entity represented by a coordinate list are controlled by the modeller. Although it would be possible for the model to change these coordinates directly, in order to support the parallel implementation described later, all changes to coordinates must be made via the modeller.

In the latter case the model can directly update the appropriate attribute's value in the member's structure. Notice, however, that the state of a turnip may also change in this case since the fly may lay an egg. In order to support the parallel implementation, it is vital that fly should not change the state of a turnip if it has moved during the same call to the behavioural function. This is because movement of a fly in the model may precipitate movement of the fly's description from one process to another.

The behaviour function which describes the transition diagram is called **fly_behaviour**. It updates the fly and turnip states according to choices made by the fly:

```
fly_behaviour ( fly, fly_coords )
{
   if time to elapse is zero
   {
      if fly has landed
      {
```

```
            if fly wants to lay an egg
            {
               lay an egg
            }
            else
            if has found a turnip (the closest) within its range
            {
               if fly wants to land
               {
                   land( fly,fly_coords,closest_turnip_coords)
               }
            }
            else
            if fly wants to move
            {
               move( fly,fly_coords )
            }
         }
       }
       else decrease time
   }
```

Notice here that we introduce the notion of time into our model with a simple counter mechanism. The functions which decide what action a fly is to take (e.g. wants to land) set the time to elapse field in the fly structure to represent the number of time units which are required for the operation of deciding whether to land and actually landing. Every time the behaviour is called, this time to elapse field is decremented until it reaches zero at which point the next part of the behaviour takes place. In this way, all of a fly's actions take a time relative to a basic operation which, in this case, is the time taken to perform a single flight.

There is one behavioural rule in particular which has caused some consternation in the implementation of this model and that is that no more than a certain number of flies may exist in a single site. We have considered trying to implement this rule both in the modeller and in the model. At one level, this rule can be seen to be introducing a new interaction in our interaction graph between the fly population and itself and therefore it may seem sensible to provide support for it in the modeller. On the other hand, the interaction is indirect and could therefore also be implemented directly in the model.

To implement this rule, we have taken advantage of the work done to provide a geometric modeller and we have used another grid to describe the world in which the flies operate. Each entry in this grid is a single integer value representing the number of flies occupying a particular point in space. This highlights the usefulness of implementing complicated modeller constructs in terms of simple abstract data types which can then be reused by the model programmer.

9 The Control Program

In order to perform the simulation, a model control program must be written. This is typically the `main` function of the model system. This function is responsible for initialising the modeller library and describing the constituents of the model in a suitable form. It is at this point that decisions are made by the model programmer as to how the model should be simulated.

For example, our model control program may be conditionally compiled according to whether the model is to be simulated using the basic, geometric or some other simulation method:

```
Population   *turnip;
Population   *fly;
```

```
  Interaction  *inter;

#if   defined(BASIC)
  /*--- use basic modeller */
  turnip = get_population(NUM_TURNIPS, sizeof(Turnip));
  fly    = get_population(NUM_FLIES,   sizeof(Fly));

#elif defined(GEOMETRIC)
  /*--- use geometric modeller */

#define DIMS    2
  static int    gridSize[DIMS] = { 100, 100 };
  static int    turnDist[DIMS] = {   5,   5 };
  Grid         *turnWorld;

  turnWorld = grid_dist_create(DIMS, gridSize, sizeof(Turnip), turnDist);
  turnip    = get_grid_population(turnWorld);
  fly       = get_gridpop_population();

  set_population_search_range(fly, FLY_SEARCH_RANGE);
#elif defined(PARALLEL)
  ...

#endif

  inter     = get_interaction(fly, turnip, ft_interaction, fly_behaviour);
```

Our aim is that the changes required to cause a model to be simulated using a different mechanism be isolated in this part of the code and that the code describing the model need not change. Of course, in changing from using a basic to geometric modeller or vice versa, this is not possible since the model is making use of the geometric information presented to it by the modeller, but it is possible in moving from a serial to parallel implementation of the geometric modeller as we will see later.

10 Parallel Decomposition

So far we have seen how we can extend the modeller library to support geometric models efficiently. We now turn our attention to how we can implement a geometric modeller in parallel. Our aim in doing so is to provide a system which can either be used to decrease the turn-around time for a model or with which we can simulate models which have previously been too large or too slow to model in serial.

We have considered several methods of parallelising the modeller and several have been prototyped, but most failed in retaining the generality of the modeller and in providing a scalable solution. In deciding how to parallelise this code, we first considered what features could potentially be decomposed:

Trivial parallelism the first option to parallelisation with all such codes is trivial parallelism where several copies of the serial program are run in parallel with different initial conditions. Our serial implementation has been designed in such a way as to allow this trivial parallelism to be exploited since it can be parameterised by command line arguments and dump output in named files.

Although this is the most efficient way of generating large amounts of data from the same model with different initial conditions it does not allow us to simulate populations larger than those which can reside within the memory of a single processor.

Fly population the fly population consists of a potentially large number of members. To reduce execution time we might consider decomposing the set of all flies across the available processes. In doing so,

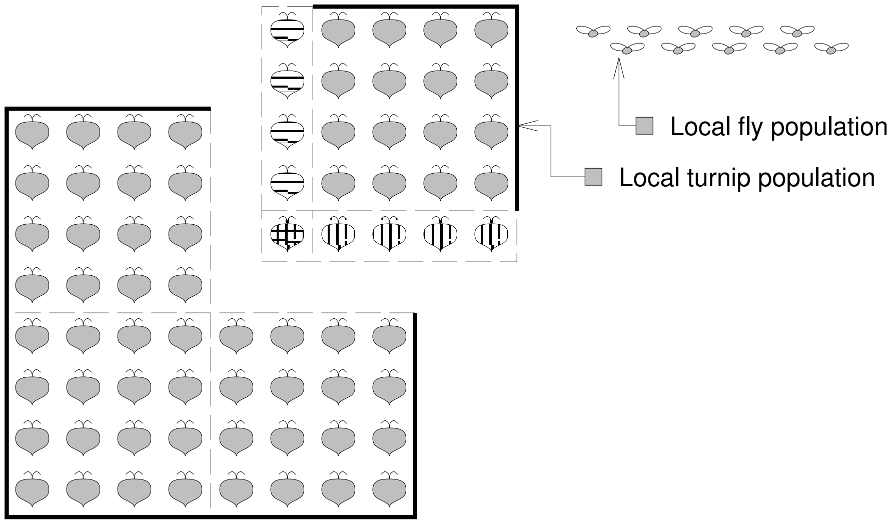


Figure 7: The world description

however, we would have to give each process a copy of the turnip population with which the flies could interact. At the end of the simulation, the copies of the turnip population are combined (e.g. by adding the number of eggs laid on a turnip for every turnip in each copy of the population) to produce our result.

This scheme is simple to implement and requires no interaction between processes during the execution of the simulation. However, it suffers from two drawbacks:

- It is impossible to implement the rule whereby no more than a certain number of flies are permitted to occupy the same position in space since the flies are distributed across processes.
- The scalability problem is not addressed since we are still limited to turnip populations which can fit into the memory of a single process.

Although the latter of these is unlikely to cause a problem with the turnip root fly simulation where the average turnip population size is expected to be kept to a reasonable limit, for other, more complicated models this may be an unacceptable failing of the system.

Turnip population Instead of decomposing the fly population, we might consider splitting the turnips up amongst the available processes. In this scenario, each process would get a portion of the turnip world. Since flies are active creatures and change their position regularly, if each process was given a copy of the entire fly population, it would still have to interact and call the behaviour function for each of the flies to keep track of which are positioned inside the local part of the grid and which are not. This leads us to the final decomposition strategy.

Turnip and fly populations Here we decompose the turnip array in a regular fashion across the available processes and we allocate each fly to a particular process according to the coordinates of each fly. For example a fly at position (0, 0) is allocated to the process which has been allocated the section of the turnip population in which position (0, 0) falls.

When a fly moves from one position to another the modeller library determines whether the new coordinates are outside the bounds of the area or volume which it is responsible for and, if so, sends the details of the fly to the owner of the new coordinates. It is for this reason that we require the coordinates of a fly to be in the control of the modeller library.

figure 7 shows this parallel decomposition.

Having decided on a decomposition scheme it is important that the interface be kept as general as possible in keeping with the other operations in the modeller library. We therefore implemented a new grid_rd_create operation which builds a grid in much the same way as described above, but which takes an extra parameter describing the set of processes across which the grid is decomposed in a regular fashion. We did not need to change the fly population initialisation since we can assume that the flies are to be decomposed across the same set of processes as the turnips.

To implement the regular decomposition we have used PUL-RD utility [Chapple, 1992]. Message passing is performed using the MPI interface [Forum, 1994].

10.1 The interaction function

The interaction function used in the parallel modeller is identical to that used for the geometric modeller. This is possible since in the parallel modeller each process is a copy of the serial implementation working on a reduced data set. However, after each call to the interaction function, the modeller library may have to swap information with other processes depending on the movements and behaviour of the populations making up the interaction.

10.2 Decomposition of the grid

PUL-RD provides a library of routines with which an n-dimensional data set can be decomposed across a set of processes in a regular manner. Once decomposed in this way, PUL-RD also provides operations to exchange information at the boundary of each of the sub-blocks of the global data set. The PUL-RD programming methodology requires that an instance of PUL-RD is initialised then opened. A PUL-RD instance is referred to by way of a *handle*.

PUL-RD assumes that for particular point in a decomposed array several neighbouring points are required to perform any calculation required at that point. For this purpose, a stencil describing which neighbouring elements are required for an arbitrary point in the data set must be passed to the call to open a PUL-RD instance. From this stencil PUL-RD can calulate exactly how large an overlap area there must be for each block.

For example, suppose we have a two dimensional array decomposed using PUL-RD and that for each point we want to calculate the average of all the immediate neighbouring points. Our stencil would then be a square of nine points each set to RD_TRUE. With this stencil PUL-RD can deduce that every sub-block of data must be extended with a halo of width one in all directions.

We have extended our grid abstract data type to incorporate PUL-RD support. The Grid object has been extended to contain a PUL-RD handle when required and the following new operations are provided:

grid_rd_create : dimensions x element size x communicator –> Grid initialises and opens a PUL-RD instance describing the decomposition of the grid data set with the given dimensions. The communicator argument is an MPI object describing, amongst other things, a set or *group* of processes.

grid_rd_dist_create performs a similar operation but compacts the array according to the distrubution of the elements therein.

All of the existing query operations can be applied to the object returned from grid_rd_create.

The grid_rd_create operation, since it calls rd_open to decompose the data, must specify a stencil for the decomposition. The stencil is defined by the accesses which are performed for an arbitrary point. In our case, therefore, we can see that the stencil is defined by the search range of a fly. With this stencil in place we can be assured that for every fly within the bounds of the block allocated to a process, all of the turnips within interaction range will be addressable directly by the modeller library.

Having decomposed our grid in this manner we must decide when the data in the overlap areas must be updated. This choice is determined by the type of interaction required by the model:

- Members of the active population do not need any information about a member of the passive population other than its location when performing an interaction. In this case the modeller does not need to keep the information stored in each passive population slot up to date on each iteration and therefore requires only a single swap of overlap regions when the grid is decomposed.

- Members of the active population require information stored inside the passive population membets when interacting. In this case, the information in the overlap region must be updated after each iteration and therefore the `rd_swap` function must be called once after each call to the grid interaction function.

Clearly, the performance of the latter approach is likely to be worse than that of the former. In our model, we notice that the interaction defined between a fly and a turnip only determines on which turnip to land according to location and does not require to look inside the fields the turnip description. Hence we can use the former approach for our model, but we note that in general, we may have to support the latter.

10.3 Message passing

We decompose the active (fly) population according to the coordinates of its members. Therefore, each process is potentially allocated a different number of active population members. Furthermore, during the simulation, members of this population may migrate out of the region owned by one process and into that of another which means that the size of the individual populations stored in each modeller process varies with time during the simulation. For this reason, the active population is represented as a list to allow dynamic growth, shrinkage and efficient removal of individual elements.

Whenever a member of the population is moved by the model, the new coordinates are checked and, if out of bounds the current state of the active member is copied into a buffer for transmission at the end of the interaction loop. The buffer into which it is copied is determined by the owner of the coordinates to which the fly is heading. We maintain an array of n buffers where n is the number of processes in the group across which we have decomposed the grid. PUL-RD provides an operation to map from a set of coordinates to the member which owns the associated grid point.

MPI provides an operation to pack a buffer with data in a network indepdendent format. When an active entity moves out of bounds therefore, we use this function to append the information to the end of the appropriate buffer. This raises the question of how the modeller, to which the active entity description is completely opaque, can pack the data in such a way that the MPI data typing mechanism can be used to ensure correct data marshalling.

We could simply have the modeller pack the data as raw bytes and leave data conversion up to the user, but this seems to defeat the purpose of the rich MPI typing mechanisms so instead we permit the registration of an MPI type which describes the structure of a member of the active population. In this way we can retain the opacity of the active population structure in the modeller and still support the seamless translation of data on heterogeneous platforms.

Since the number of flies departing from or arriving at a process is completely determined by the model simulation, the best we can do is place an upper limit on the possible number of arrivals and departures. Before beginning the simulation, each process initiates a single non-blocking receive into a buffer which is large enough to accomodate the upper limit of arrivals.

After each interaction loop this non-blocking communication is tested for completion. If a message containing active entities from a remote process has arrived the test will succeed and the flies can be unpacked from the buffer and inserted in the list of active entities. This process is repeated until the test fails at which point the interaction loop continues.

There is clearly potential for the processes in this decomposition to become badly out of synchronisation. This could occur, for example, when one process has less active members to process than others or when it has less members to communicate. It is initially unclear whether this synchronisation is important or not since the active members of our active population do not interact with each other and only have limited indirect interaction through the passive population.

If we discover that we require the synchronisation, then we can introduce extra mechanisms to ensure that all members of the active population which are in transit have been received before the next iteration begins. One possible scheme for doing so would be to have each process maintain an array of size n where n is the number of processes. A process would set entry i to one if it sends some active members to process i, otherwise entry i would remain zero. At the end of each interaction loop this array would be reduced using a sum operation and the results scattered across the processes with the result that every process would receive an indication of how many messages had been sent to it during the interaction loop.

11 Gathering and Analysing results

Due to lack of time, no meaningful results have been forthcoming from this project at this time. This section explains what results we hope to gather in the future and how they will be analysed and verified against observed data.

The model is implemented in terms of probabilities which in turn are implemented in terms of a random number generator which produces a series of numbers according to an initial seed. By varying this seed therefore, we can produce a large number of different simulations and average the results across these different runs.

11.1 What constitutes a result

Before gathering and analysing results, we must determine what data to extract from the simulation. Initially the data we are most interested in is the number of eggs which have been laid in a crop since this determines the percentage of crop which is ultimately lost due to pest damage. The simplest form of result from the simulation is therefore the average, maximum and minimum number of eggs laid on a plant.

11.2 Visualising results

In order that the behaviour of the flies can be both better understood and verfied as sensible fly behaviour, we have implemented a very simple visualisation tool in which the flight paths of individual flies are represented as lines and levels of fly activity and egg laying are represented using simple bar graphs (one per grid site).

12 Future work

There are several potential extensions to this work. Each extension either relates to the model or to the modeller library:

Modeller extensions the modeller library can be seen as a repository for useful simulation methods and tools. The modellers described in this document represent useful tools for the simulation of our particular model, but many other models (e.g. the ant farming model developed as part of the previous year's summer scholarship programme [Rambaut, 1993]) will not benefit from the optimisations discussed here. The modeller could therefore be extended to support other types and classes of interaction more efficiently.

More model-independent tools could also be added to the modeller library. For example, the existing visualisation system could be extended to provide more detailed visualisation of individual population elements or global conditions of populations (e.g. running total of eggs laid in our case). Extending the visualiser in this way would not only simplify verification of models, but could prove useful as a teaching tool

In the interests of using the modeller library as a tool to predict good crop planting strategies, we might extend the modeller with the ability to automatically search for good crop distributions either using exhaustive or other techniques such as a genetic algorithm or hill-climbing.

Model extenstions Many aspects of the root fly model have been simplified or ignored in the present implementation. There is a large amount of published behavioural data for both turnip and cabbage root flies and the model could be extended ad infinitum with more and more detailed behavioural models. It would be very interesting to see at what point during this refinement process that meaningful results begin to emerge.

13 Conclusions

We have developed a system which allows clear expression and simulation of any model which can be described as an interaction graph. We have shown how to optimise the simulation for a particular class of models and discussed briefly how other classes may be dealt with in a similar way. Our aim has been to provide a system in which a model can be developed entirely on a serial computer then ported to a parallel one with only one or two extra lines of code required and, for our model at least, this goal has been acheived.

References

[Chapple, 1992] Simon R Chapple. PUL-RD prototype user guide. Technical Report EPCC-KTP-PUL-RD-PROT-UG, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[Ellis, 1980] P.R. et al Ellis. The relationship between cabbage root fly egg laying and volatile hydrolysis products of radish. *Annals of Applied Biology,95*, 1980.

[Forum, 1994] Message Passing Interface Forum. Document for a standard message-passing interface. Technical report, University of Tennessee, March 1994. Available on **netlib**.

[Havukkala, 1987] I Havukkala. Odour source finding behaviour of the turnip root fly,*delia floralis*. *Journal of Applied Entomology,104*, 1987.

[Hawkes, 1976] T.H. Hawkes, C. Coaker. Behavioural responses to host-plant odours in adult cabbage root fly,(*erioischia brassicae*). *Symposium Biology Hungary,16*, 1976.

[Jones, 1992] R.E. Jones. Search behaviour: strategies and outcomes. *Proceedings of the 8th International Symposium on Insect-Plant Relationships*, 1992.

[Nottingham, 1987] T.H. Nottingham, S.F. Coaker. Changes in flight track angle of cabbage root fly,*delia radicum*,in diffuse clouds and discrete plumes of the host plant volatile allylisothiocyanate. *Entomologica Experimentalis et Applicata,43*, 1987.

[Prokopy, 1983] R.J. et al Prokopy. Visual detection of host plants by cabbage root flies. *Entomologica Experimentalis et Applicata,34*, 1983.

[Rambaut, 1993] Andrew Rambaut. A parallel simulation of the foraging behaviour of the wood ant *formica aquilonia*. Technical report, Edinburgh Parallel Computing Centre, University of Edinburgh, September 1993.

[Stadler, 1978] E. Stadler. Cheremoreception of host plant chemicals by females of *delia (hylemya) brassicae*. *Entomologica Experimentalis et Applicata,24*, 1978.

[Stadler, 1990] R. Stadler, E. Schoni. Oviposition behaviour of cabbage root fly,*delia radicum* (l.),influenced by host plant extracts. *Journal of Insect Behaviour,3(2)*, 1990.

Deborah Bellotti is a student in Computer Science at the University of Genoa, Italy. After graduation in 1995 she hopes to take up a postgraduate course.
This project was supervised by Alasdair Bruce (EPCC) and Richard Hopkins (SAC).