the scenario being simulated, rather than upon performance. If a model is big and complex enough that its runtime is problematic – i.e., is measured in hours to days – the overhead due to choosing the nonWF model type will probably be small.

16.16 Alternation of generations

SLiM is generally a framework for modeling diploid organisms, but with some creative scripting that assumption can be modified. We have seen some recipes for modeling haploids, as in sections 14.9, 16.13, and 16.14. In nonWF models a similar strategy can be used to fully model the phenomenon of *alternation of generations*, the way that diploid and haploid life cycle stages generally alternate in organisms that are often thought of simply as "diploids". Many sexual animals, for example, have a multicellular diploid phase that produces a unicellular haploid phase – sperm and eggs – that then fuse, in fertilization, to produce the next diploid generation. In plants this situation is generally even more pronounced, often with a multicellular haploid phase, the gametophyte, that can be free-living and large – often larger and more obvious than the diploid sporophyte, which is often reduced. For many organisms, then, it may be important to model both the haploid and diploid phases explicitly; mutations may be expressed differently between them, selection may act differently upon them, they may migrate or disperse differently, and so forth. SLiM does not have intrinsic support for modeling this alternation of generations, but it is straightforward to implement in script in a nonWF model, as we will see in this section.

This model will be somewhat complicated, so let's start with the setup:

```
initialize()
{
    defineConstant("K", 500);     // carrying capacity (diploid)
    defineConstant("MU", 1e-7);   // mutation rate
    defineConstant("R", 1e-7);    // recombination rate
    defineConstant("L1", 1e5-1);  // chromosome end (length - 1)

    initializeSLiMModelType("nonWF");
    initializeSex("A");
    initializeMutationRate(MU);
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, L1);
    initializeRecombinationRate(R);
}
1 early()
{
    sim.addSubpop("p1", K);
    sim.addSubpop("p2", 0);
}
```

We use defined constants for several of the model parameters. The recipe here involves only neutral mutations, but extending it to other types of mutations should present no difficulties.

This is a sexual model, so we set up separate sexes with `initializeSex()`. We are not modeling sex chromosomes, but we will track the sex of individuals in both the diploid and haploid phase; sperm will be considered "male", and eggs "female", in this model.

A key point in the design of this model is that although we are modeling only a single subpopulation, we use two subpopulations in the model, `p1` and `p2`. The first, `p1`, is used to hold diploids; the second, `p2`, is used to hold the haploid sperm and eggs. This separation is not strictly necessary, but it makes the design of the model simpler, because this way we can define a

`reproduction()` callback for `p1` that reproduces the diploids (producing sperm and eggs), and a separate `reproduction()` callback for `p2` that reproduces the haploids (producing fertilized eggs that develop into diploids). For other processing of the individuals in the model, such as `fitness()` callbacks, this partitioning will also prove useful, as we will see below.

The next step is to define the `reproduction()` callbacks. Let's start with the one for `p1`:

```
reproduction(p1)
{
    g_1 = genome1;
    g_2 = genome2;

    for (meiosisCount in 1:5)
    {
       if (individual.sex == "M")
       {
          breaks = sim.chromosome.drawBreakpoints(individual);
          s_1 = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "M");
          s_2 = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "M");

          breaks = sim.chromosome.drawBreakpoints(individual);
          s_3 = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "M");
          s_4 = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "M");
       }
       else if (individual.sex == "F")
       {
          breaks = sim.chromosome.drawBreakpoints(individual);
          if (runif(1) <= 0.5)
             e = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "F");
          else
             e = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "F");
       }
    }
}
```

The definitions of `g_1` and `g_2` at the beginning are just shorthand, to keep the lines later in the callback from being so long that they wrap when shown here. As usual in nonWF models, this callback is called by SLiM once per individual in `p1`, giving the individual an opportunity to reproduce – in this model, an opportunity to produce gametes. The top-level loop causes the focal diploid individual to undergo meiosis exactly five times; this is an oversimplification, obviously, but there is no need, in most models, to generate millions of sperm. Within the loop, male individuals undergo meiosis by producing four sperm, whereas females produce just a single egg (plus three "polar bodies" that are discarded by meiosis in most sexual species, due to anisogamy; the polar bodies are not modeled here).

Gametes are produced by the `addRecombinant()` method, adding the resulting haploid individuals to `p2`. The calls to `addRecombinant()` here pass `NULL` for the genomes and breakpoints that generate the second genome of the offspring; this results in an empty second genome in the offspring, as is typical when modeling haploids in SLiM. The genomes used to generate the first genome of the offspring can be supplied as either (`g_1, g_2`) or (`g_2, g_1`); the first of the two genomes supplied is the copy strand at the beginning of recombination. Since the sperm generated use all of the genetic material from meiosis, both of those options are used (twice, because of the homologous chromosomes involved in meiosis); since egg generation produces only a single gamete, the choice of initial copy strand is randomized with a call to `runif()`.

Particularly for the sperm, since we want to generate the gametes in a realistic fashion following the rules of meiosis, we generate the recombination breakpoints ourselves and use them to

generate complementary gametes. To generate breakpoints in the standard SLiM fashion, we call the `drawBreakpoints()` method of `Chromosome`; by default this produces a set of breakpoints identical to what SLiM would generate for its own internal use in reproduction. The number of recombination breakpoints generated is chosen by `drawBreakpoints()`, by default, using the overall recombination rate defined by the model.

Now let's look at the `reproduction()` callback for `p2`, which contains haploid gametes:

```
reproduction(p2, "F")
{
    mate = p2.sampleIndividuals(1, sex="M", tag=0);
    mate.tag = 1;

    child = p1.addRecombinant(individual.genome1, NULL, NULL,
      mate.genome1, NULL, NULL);
}
```

This callback is defined only for females – i.e., eggs. Each eggs gets to "reproduce" – be fertilized – to produce a new diploid organism in `p1`. In this model a random sperm is chosen to fertilize each egg, but one could easily implement phenomena such as sperm competition here to make the choice non-random. We mark sperm that have been used to fertilize an egg with a `tag` value of `1`, so that they will not be used again; when we draw a random sperm, we specify in the call to `sampleIndividuals()` that the sperm chosen must have a `tag` value of `0`, indicating that it has not already been used. Once the fertilizing sperm has been selected, it is tagged with a value of `1`, and the diploid zygote is generated with a call to `addRecombinant()`. The call to `addRecombinant()` here supplies only a single genome for each of the offspring genomes, with `NULL` for the breakpoint vectors; this makes the offspring's genomes a clonal copy of the corresponding genomes from the gametes. Normally, new mutations would be generated and added by SLiM during this clonal replication; we will fix that momentarily.

These callbacks implement the generation of gametes and then the fusion of gametes to produce diploid zygotes; but a little additional machinery is needed, which we implement in an `early()` callback that cleans up after reproduction and sets up for the next reproduction event:

```
early()
{
    if (sim.generation % 2 == 0)
    {
      p1.fitnessScaling = 0.0;
      p2.individuals.tag = 0;
      sim.chromosome.setMutationRate(0.0);
    }
    else
    {
      p2.fitnessScaling = 0.0;
      p1.fitnessScaling = K / p1.individualCount;
      sim.chromosome.setMutationRate(MU);
    }
}
```

In even-numbered generations the top half of this event will execute; in odd-numbered generations the bottom half will execute. In an even-numbered generation, at the point that `early()` events are called, `p1` will have just generated gametes. This recipe assumes non-overlapping generations, so here we kill off the diploids by setting their `fitnessScaling` to `0.0`; `p1` will be emptied out completely. Next, we set the `tag` values of all of the gametes in `p2` to `0`; this marks all of the sperm as unused, in preparation for the way the `reproduction(p2)` callback uses

the `tag` field. Finally, we set the mutation rate to `0.0`; we do not want new mutations to be generated by SLiM during fertilization, so we need to disable mutation temporarily.

In odd-numbered generations, gametes have just undergone fertilization, filling `p1` up with new diploid offspring. We therefore kill off the haploid gametes by setting their `fitnessScaling` to `0.0`; `p2` will be emptied out completely. Since each egg produces a zygote, we will have way too many diploids; we will be far above carrying capacity. The next line thus implements density-dependent selection on `p1`, as usual in nonWF models; note that no such density-dependence was imposed upon the gametes in this model. Finally, we set the mutation rate back up to the defined constant `MU`, since we want new mutations to arise during gamete production.

With this design, the population will flip back and forth between `p1` and `p2` as it flips between diploidy and haploidy, and the mutation rate will flip on and off as well. It would be straightforward to implement overlapping generations of diploids in this model; one could even delve into more esoteric ideas such as sperm storage. Fitness effects could differ in the haploid and diploid phases, by implementing `fitness()` callbacks that apply only to `p1` or `p2`.

All that is left to finish off the model is a termination event, which here is trivial:

```
1000 late()
{
    sim.simulationFinished();
}
```

This approach to modeling the alternation of generations may be overkill in many practical situations. This model runs much more slowly than the equivalent model of only the diploid phase; for one thing, it is generating a population of gametes that is more than ten times larger than the population of diploids, every generation. Other strategies for modeling life cycle complexity may be usable instead; section 16.8, for example, presents a model of pollen flow between subpopulations of plants, which is simple to model without getting down to the details of modeling individual pollen grains and the sperm cells they produce as separate entities. Additional biological realism should generally be incorporated into a model only when there is reason to believe that it matters – that it would affect the results of the model. In some cases, however – such as when one wishes to have selection operate in the haploid phase – the additional biological realism of modeling alternation of generations may be useful.

Even more esoterically, one could use the same basic concepts to develop models of mating systems such as haplodiploidy; all that is really needed is to set up rules of reproduction that move the genomes around from individual to individual in the correct way using `addRecombinant()`, which is designed to be as flexible as possible in order to accommodate these sorts of purposes. Partitioning the population according to genetics – here, diploids versus haploids – is a useful trick in many scenarios. Indeed, such artificial partitioning can be very useful in other contexts too, such as storing non-reproducing juveniles separately from reproductive adults, or storing sympatric but reproductively isolated groups separately. Since the spatial interaction engine of InteractionType evaluates interactions on a per-subpopulation basis (see chapter 15), it can also be useful to partition the population according to "interaction groups" – sets of individuals that interact with each other, but not with the individuals in other interaction groups.

16.17 Meiotic drive

In section 12.3 we saw a model of a gene drive, a genetic construct that drives itself higher in frequency by copying itself from one chromosome to the other. We implemented that mechanism with a `modifyChild()` callback that copied the drive allele from one child genome to the other in each newly generated offspring. In this section, we will look at a different type of drive: *meiotic*