Online Supplement for
SLiM 4: Multispecies eco-evolutionary modeling

The American Naturalist

Benjamin C. Haller [1†]
Philipp W. Messer [1]

[1] *Department of Computational Biology, Cornell University, Ithaca, NY, USA*
[†] Corresponding author: bhaller@mac.com

Multispecies SLiM Design Specification

This is a "white paper" describing a design to extend the SLiM forward genetic simulation framework (https://messerlab.org/slim/) to be capable of simulating the evolution, and co-evolution, of multiple species. I will discuss the goals of this work, and then lay out the general approach I have adopted. After that I'll get into details: exactly how a user specifies a multispecies SLiM model, and what changes in SLiM's scripting interfaces (the Eidos classes and methods used in a SLiM script) are involved. Familiarity with SLiM is assumed.

Goals

The primary goal is that SLiM should be able to simulate the evolution of multiple species in a single model. These multiple species should be able to have completely different genetics (chromosome length, mutation types, genomic element types, mutation rates, etc.), as well as completely different scripted behaviors.

Ecological interactions between species should be possible: competition, mutualism, predation, parasitism, infection, etc. In the scripted behavior of one species, it should therefore be possible for that species to *access* state about the other species: their individual positions in space, their genetics, etc. It should also be possible for scripted behavior of one species to *modify* the state of other species: to influence their fitness, including causing mortality, or to influence their dispersal, or their mating, or whatever.

Species should be able to share an environment (spatial or nonspatial), if desired – being affected by the same environmental variables, but also, for example, competing over the same limited resources. Conversely, they should be able to occupy separate environments, with little (parapatric) or no (allopatric) interaction.

Species should be able to exist on different timescales – different frequencies/schedules for reproduction, movement, and mortality, different lifespans and generation times, and so forth. Species might also exist at different, even non-overlapping, times during simulation, perhaps arising and going extinct according to their own dynamics. However, an acceptable limitation, I think, is that all species that will be modeled must be specified up front, even if they only emerge later in a model run. SLiM models of the speciation process itself, with the stochastic, unplanned emergence of new species, would probably continue to be written as single-species models (which is already possible), since the diverging proto-species would be so similar as to not require representation as distinct species in the model; rather, they would be modeled as subpopulations of the same species with emerging (or perhaps even complete) reproductive isolation from each other.

Finally, an important goal is that existing single-species models should require few or, ideally, no changes to their code, and the syntax of multi-species models should be as similar as possible to the syntax of single-species models; the learning curve for SLiM is already steep enough! The performance impact on single-species models from these multi-species changes should also be minimized.

General approach

Repurposing SLiMSim

In SLiM 3 the class SLiMSim is used, in both Eidos script and in SLiM's underlying C++ implementation, to represent a running simulation; and in SLiM 3, that simulation is of a single species. For the multispecies redesign I have kept SLiMSim as the class representing a single species, but renamed it to Species for clarity, and then introduced a new meta-level above Species that manages multiple species as separate Species objects. This

way all of the single-species state that SLiMSim tracks remains species-specific and unchanged: all of the genetic architecture (chromosome, mutation types, genomic element types, etc.), all of the internal bookkeeping (tree-sequence recording of the ancestry of a given species, in particular), and all of the Subpopulation, Individual, Genome, and Mutation objects that comprise the population state of a simulated species (Figure 1).

In SLiM's C++ code a new class, Community, has been added that contains a vector of Species objects, one per species. Some code related to the overall simulation state has moved from SLiMSim to Community – running the phases of each simulated generation, management of script blocks and log files, and management of InteractionTypes for spatial interactions, for example. The Community object is visible in Eidos as a global constant named `community`, and provides some additional new functionality to support multiple species.

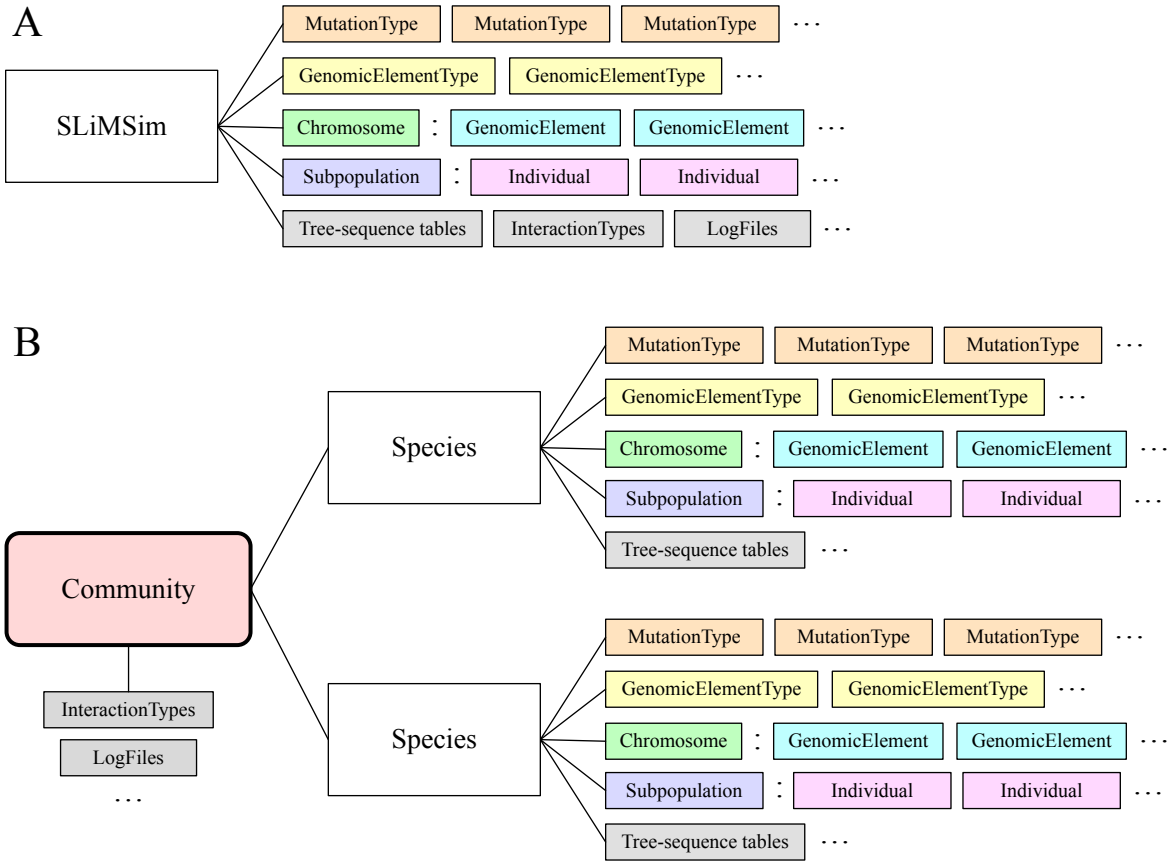

Figure 1. The basic approach of repurposing SLiMSim to represent a single species. (A) The SLiM 3 design, in which SLiMSim represents a simulation of one species. (B) The multispecies redesign, in which SLiMSim, renamed Species, represents a simulation of a single species within a single-species or multispecies model, and a new class, Community (red rounded rectangle), manages the collection of Species objects. Note that some features of SLiMSim, such as InteractionTypes and LogFiles, have moved up to Community.

This design, of separate Species objects for each species in a multispecies model, means that virtually *all* simulation state is distinct between species. One species could have tree-sequence recording enabled while another doesn't; one could use continuous space while another just uses non-spatial subpopulations; and so forth.

Notably, one species might have genetics and be capable of evolution, while another species might still be individual-based, with modeled behavior, dispersal, etc., but might not have any associated genetics, if the evolution of that species is unimportant to the simulation. In any case, this separation between the simulation state of different species is enforced throughout SLiM's Eidos interfaces: for example, if you are modeling mice and foxes, Mutation objects would be species-specific, and so a mouse mutation could not be added to a fox genome. Similarly, even if both mice and foxes need a "neutral" mutation type, each species would declare its own neutral mutation type separately, rather than sharing the same mutation type for that purpose. Such strict enforcement of the separation of state between species is not technically necessary in all cases, but a clean division between species seems best for conceptual clarity.

One place where simulation state is *not* kept separate – where logic has moved up to the Community level – is in the naming of SLiM global variables in Eidos. These variables are things like the `p1`, `p2`, … variables that represent subpopulations, the `m1`, `m2`, … variables that represent mutation types, etc. These variable names must now be unique across the whole community; if mice live in two subpopulations, `p1` and `p2`, then foxes may not use `p1` and `p2` as names for *their* subpopulations (even if they occupy the same environments, conceptually). Instead, the foxes that eat the mice in `p1` might live in `p3`, and those that eat the mice in `p2` might live in `p4`. This allows these variables to continue to be truly global, for both backward compatibility and ease of scripting; even when you are executing fox behavior code, as discussed below, you can still access the `p1` global to look at the mice that live in the same environment as the `p3` foxes.

As mentioned above, another aspect of SLiM that has moved up to Community is the execution of the generation cycle. It is usually desirable for the generation cycles of species to execute somewhat simultaneously, in an interleaved fashion, since species can affect each other's mortality, reproduction, and other behavior, and since this is typically biologically realistic – species in the real world exist in the same timestream and live out their life histories simultaneously. For this reason, the Community object now manages SLiM's Eidos script blocks – `first()`, `early()`, and `late()` events, as well as Eidos callbacks. Community now keeps a vector of all script blocks in the simulation and manages them on behalf of all of the individual species, including the registration, deregistration, and rescheduling of those script blocks. The way the generation cycles of species are interleaved will be discussed below (and note that if one does *not* wish the generation cycles of species to be interleaved, the new multispecies design also allows that, as we shall see).

Species declarations

Species have names that obey the standard naming conventions of variables in Eidos (starting with a letter or an underscore, not containing spaces, etc.); a fox/mouse model could have species named `fox` and `mouse` in Eidos. These species names would be used to create global constants; a fox/mouse model with species named `fox` and `mouse` would have global constants named `fox` and `mouse` that would refer to the Species objects for those species, just as `sim` refers to the single species defined in SLiM 3 models. The global constant `sim` would not be available in multispecies models (unless one chooses, confusingly, to name a species `sim`); it would be replaced entirely by these new species-specific globals, to avoid ambiguity regarding which species is being referenced. Single-species models that do not explicitly declare their species name still use `sim`, however; that is the default name of the single species (preserving backward compatibility).

The species implemented by a model are *declared* by the `initialize()` callbacks defined in the SLiM script. In SLiM 3, `initialize()` callbacks look like this:

```
initialize() {
    ...
}
```

In a single-species SLiM model this would continue to be the standard syntax, and would implicitly declare a species with the default name `sim`. In a multi-species SLiM model, however, this declaration syntax would be augmented by an additional *species specifier*, like so:

```
species fox initialize() {
    ...
}
```

A separate `initialize()` callback would be written to declare the "mouse" species:

```
species mouse initialize() {
    ...
}
```

Since each species is represented by a completely separate Species object, the initialization code for each species could, and usually would, be different. These `initialize()` declarations establish what species names are declared in the model. In all other contexts, using a species name that has not been so declared results in an error, preventing typos (`moose` instead of `mouse`) from causing confusion.

The one thing that is required to match between all species in a multi-species model is their model type (WF or nonWF), for practical reasons. The model type is thus now a property of the whole community, not a species-level property. That raises the question of how community-level (i.e., whole-simulation-level) initialization is done in multispecies models. To accommodate that, one uses a similar syntax:

```
species all initialize() {
    ...
}
```

This declares an `initialize()` callback that is *non-species-specific*, responsible for initializing state at the community/simulation level. In a multispecies model, your call to `initializeSLiMModelType()` must be in such a non-species-specific `initialize()` callback; calls to `initializeInteractionType()` to set up spatial interactions must also go into these callbacks, since spatial interactions are now managed at the community level. You might also wish to put configuration of simulation parameters into such a callback, particularly to set up parameters that affect the whole community. These non-species-specific `initialize()` callbacks are always run before any species-specific `initialize()` callbacks.

As mentioned above, if no species specifier is given for an `initialize()` callback then the name `sim` is assumed, and in that case, no other species may be declared (and `species all` may also not be used). In this case the `initialize()` callback contains both species-specific and non-species-specific initialization calls; the community and species levels are conflated in single-species models, both for simplicity and for backward

compatibility with SLiM 3. In most respects, though, there is very little difference between a multi-species model and a single-species model; a single-species model is mostly just a model that happens to implement only one species, which happens to be implicitly named `sim`.

The global Species constants for all species (`fox`, `mouse`, etc.) are created after all `initialize()` callbacks have finished executing (as for `sim` in single-species models now). This means that species cannot reference each other during initialization, which seems reasonable and avoids order-dependencies during the initialization phase. The new global `community` object is similarly created after all initialization is complete.

Ticks, cycles, and generations

One of the goals stated above is to allow species to run at different "speeds" – one species might reproduce much more often than another, for example. To make this easier, I have generalized SLiM 3's way of working with time. It was already unfortunate that SLiM used the term "generation" to refer to one cycle of execution in the simulation; when SLiM 3 added nonWF models, allowing overlapping generations, variation in individual ages, life history tables, and so forth, the term "generation" no longer fitted SLiM's concept of a cycle of simulation execution terribly well. The term "generation" fits even less well in a multispecies world in which different species run on different timescales. For that reason, although it does cause a minor break in backward compatibility, the term "generation" is now being retired in favor of two new terms: *ticks* and *cycles*.

A *tick* is now what a "generation" was previously: a single cycle of SLiM's internal engine, allowing reproduction, movement, fitness calculations, mortality, and so forth. Everything defined in terms of "generations" in SLiM 3, such as the times at which events and callbacks are scheduled to run, has shifted over to being defined in terms of ticks. The current tick is available from `community.tick` instead of SLiM 3's `sim.generation` (since the tick counter is global and the same for all species). Other similar API changes have been made to reflect this shift, and tree-sequence recording times are all in ticks.

A *cycle* is also, in a sense, what a "generation" was previously: the number of times that a given species has executed its cycle (starting from 1). The old API, `sim.generation`, has been removed; instead, we now have `sim.cycle` (or `fox.cycle`, `mouse.cycle`, and so forth). In the default case (such as a legacy SLiM 3 single-species model), the cycle counter starts at 1 – you could think of it as counting `initialize()` callbacks as the first cycle – and counts upwards by 1 in synchrony with the tick counter. As a result, existing model code will work without alteration; `community.tick` and `sim.cycle` will always have the exact same value in single-species models.

What, then, of "generations"? In the context of overlapping generations, varying ages of first reproduction and mortality, etc., the biological meaning of the "generation time" is somewhat unclear and arbitrary anyway (see, e.g., Charlesworth 1994 on generation overlap); one could write Eidos code to measure it, given a particular choice of definition, but that is not what SLiM 3 meant by the term "generation", which caused confusion. Generations in SLiM 3 were merely an artificial modeling construct, and so in this redesign the removal of that term, in favor of "ticks" and "cycles", frees up "generation" to once again mean the "biological generation" – whatever one might mean by that! SLiM will still use the term sometimes, particularly in the context of WF models where one really does mean a "biological generation"; it is useful to talk of the "parental generation" versus the "offspring generation", for example. But what used to be called the "generation cycle" is now called the "tick cycle", because it runs every tick – or it can simply be called the "cycle".

To make it easy for different species to run on different timescales, it is now possible to tell SLiM that a given species should only execute in particular ticks, referred to as the ticks in which the species is *active*. Only when the cycle of a given species executes – when it is active – is the cycle counter for that species incremented. A new initialization function, `initializeSpecies()`, takes two optional parameters: `[integer$ tickModulo = 1]` and `[integer$ tickPhase = 1]`. A given species will be active every `tickModulo` ticks, beginning in tick `tickPhase`. For `tickModulo` 3 and `tickPhase` 5, for example, a species would be active, and thus execute its cycle, in ticks 5, 8, 11, 14, etc. (Figure 2). Since each species has its own `initialize()` species declaration, and thus its own call to `initializeSpecies()`, the `tickModulo` and `tickPhase` parameters are species-specific.

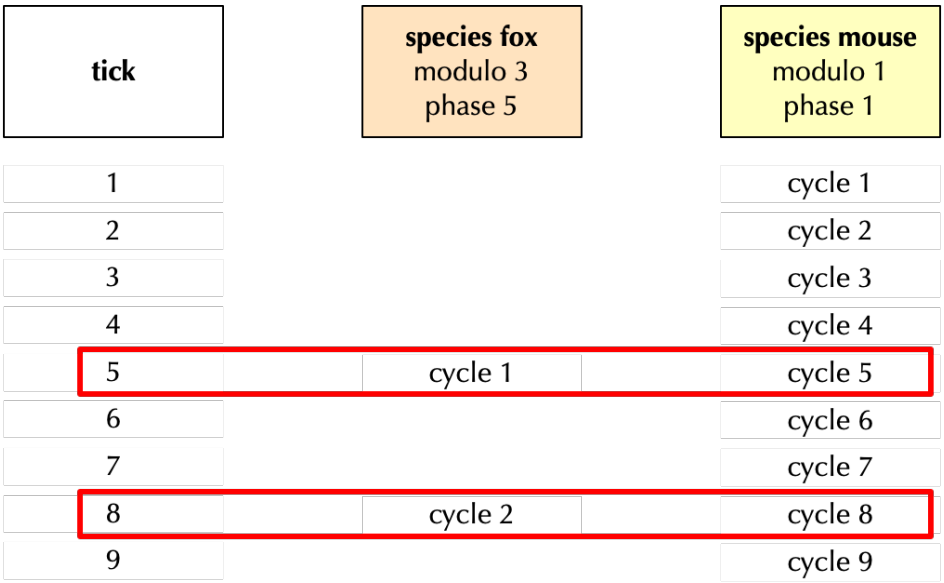

Figure 2. Scheduling of cycles in an example two-species model. The fox species, with modulo 3 and phase 5, executes its cycle only occasionally; the mouse species, with modulo 1 and phase 1, executes in every tick. In ticks 5 and 8 (red outlines), both species are active and their cycles are executed in an interleaved fashion.

Ticks are thus "objective time", like the ticking of a clock, and each species might live out its life history different as that clock ticks. Note that the biological meaning of a "tick" is up to the user; they could be days, months, seasons, years, or whatever you wish. If you have two species that run on different timescales, you would want to at least configure the model so that the timescale of your model's "ticks" match the species with the faster generation time. However, you could make the "ticking" even faster than that, if you wanted to, and then configure the timescale of each species on that global timescale. These are model design decisions; SLiM will do as you command.

If this level of control is insufficient (probably rare), a species can control exactly which ticks it is active in. This is done by using the default value of 1 for `tickModulo` and `tickPhase` so that the species is given an *opportunity* to execute in every tick. Then, as the first thing that species does in each tick, it can optionally call a new Species method, `skipTick()`. This method short-circuits the rest of the cycle for that species in that tick. This must be done first thing, to ensure that the species does no work in the current tick; the `first()` events

added in SLiM 3.7 provide a convenient opportunity. Using a `first()` event, calling `skipTick()` is very simple, in both WF and nonWF models (the `ticks all` timing designation will be explained in the next section):

```
ticks all first() {
   fox_wants_to_skip = ...;  // whatever custom logic is desired

   if (fox_wants_to_skip)
      fox.skipTick();
}
```

If `skipTick()` is called for a species in this manner, no further species-specific code will be run for that species in the current tick, the cycle for that species will not be run, the cycle counter for that species will not be incremented, and the species will not be considered to be active in the current tick. Calling `skipTick()` at a later point in the tick cycle is an error; it must be called in a `first()` event.

It's worth noting that all of these timescale mechanics – `tickModulo`, `tickPhase`, and `skipTick()` – are implemented on top of the existing `active` property of SLiMScriptBlock, which already allows particular events/callbacks to be skipped in particular ticks. Using the new timescale mechanics only does a little bit more beyond that existing facility, such as preventing the increment of the cycle counter in ticks that a species skips, and preventing the execution of *all* cycle stages for that species (even those not associated with events or callbacks).

Species-specific callbacks

SLiM supports a variety of callbacks, which are called by SLiM in order to influence the behavior of the model in some way. Most callbacks are involved in the process of reproduction, including `reproduction()`, `mateChoice()`, `modifyChild()`, `recombination()`, and `mutation()` callbacks. One type of callback, the `fitness()` callback, is involved in the process of fitness evaluation (technically, `fitness(NULL)` callbacks are a separate type, also involved in fitness evaluation – a syntactic oddity). One type, the `survival()` callback, is called to govern the mortality decisions SLiM makes for individuals. Finally, `interaction()` callbacks are called when the strengths of interactions between individuals are being evaluated. (We will skip over `initialize()` callbacks here since we already discussed them above.)

All of these types of callbacks except `interaction()` callbacks involve decisions about modeling a particular species – how it reproduces, how it defines fitness, how it undergoes selection/mortality. These callbacks in SLiM are therefore species-specific, conceptually. This is not the case for `interaction()` callbacks, which – in a multispecies model – might influence interactions between individuals of different species; they are something of a special case, therefore, and will be discussed separately at the end of this section.

The species-specific callbacks in a SLiM script are now *required* to be species-specific: they *always* specify behavior for one species. This is designated in the same manner as for `initialize()` callbacks; for example:

```
species fox 1:100 reproduction() {
   ...
}

species mouse fitness(m1) {
   ...
}
```

And so forth. In a single-species model with no explicit species declaration, a specifier of `species sim` is implied for all callbacks (and it may not be written explicitly, since the species name was not explicit).

It is required, then, in a multi-species model, to provide this explicit `species` specifier for all callbacks except `interaction()` callbacks, and two species cannot share the same callback code even if they want the same behavior. (Sharing identical callback behavior is expected to be fairly rare; even if two species are quite similar there are likely to be subtle differences in reproductive timing, fecundity, mate choice details, etc., such that they would probably want different callback implementations. But when code sharing is truly desired, it is easy to accomplish with a user-defined function containing the shared code, called by the callback of each species. The small inconvenience of this arrangement, in these rare cases, is outweighed by the conceptual clarity afforded by the policy that a species-specific callback is *always* associated with exactly one species.)

And what of `interaction()` callbacks? Since they are non-species-specific and are managed at the community level, their syntax is exactly parallel to the same situation for `initialize()` callbacks:

```
species all interaction(i1) {
   ...
}
```

The `species all` specifier is required for `interaction()` callbacks in multispecies models, to state explicitly that the callback is not species-specific. So, in short, every callback in a multispecies model *must* be declared with a `species` specifier; for `interaction()` callbacks it must be `species all`, whereas for other callbacks it must be species-specific. In single-species models, `species` specifiers are *never* used; they look just like SLiM 3 models have always looked.

Event timing

Events – `first()`, `early()`, and `late()` events – are a bit different from callbacks, because callbacks (except `interaction()` callbacks, as discussed above) are called only for a particular subset of individuals – the individuals in one given species. An event, on the other hand, might affect a single species or more than one – or no species at all, if it just manipulates global state. Even if a given event affects one species, it might in some sense be the "responsibility" of another species (such as an event in which foxes hunt mice; that might primarily *affect* the mice, especially if the model doesn't track individual fox food intake, but it might still be driven by *behavior* on the part of the foxes). As discussed above in *Ticks, cycles, and generations*, different species might be active (i.e., run their cycle) in different ticks; and yet, also, even if foxes are only active every third tick, you might want them to hunt in every tick. What we want for events, then, is the ability to designate whether an event ought to run only in ticks when a particular species is active, or ought to run in every tick, regardless of what species that event operates upon, or what species is "responsible" for the behavior of the event. It therefore makes sense for the syntax for designating events to be a little bit different:

```
ticks fox 1:100 first() {
   ...
}

ticks mouse early() {
   ...
}

ticks all 5: late() {
   ...
}
```

The first event will run in ticks `1:100`, but only if the `fox` species is active in a given tick. The second runs in *every* tick in which the `mouse` species is active, since no tick range is given. The third runs in every tick from 5 onward, whether any species is active or not (even in ticks in which *no* species is active). This `ticks` specifier is *required* for events in multispecies models, to avoid confusion and accidental bugs, just as `species` specifiers are mandatory in multispecies models. In single-species models that do not declare a species name explicitly, `ticks` specifiers are not allowed and `ticks sim` is assumed (providing backward compatibility), just as `species` specifiers are not allowed and `species sim` is assumed.

User-defined functions can *never* be designated in either of these ways – with a `species` specifier or a `ticks` specifier. User-defined functions can use the species-specific global constants, like `fox` and `mouse`, to refer explicitly to particular species. A user-defined function could also be passed a Species object on which to operate, if desired, to make it agnostic to the particular species implemented in a model (allowing easy reuse of user-defined functions between different species). Note that user-defined functions that reference `sim` will need to be rewritten in order to be reused in a multispecies model; they would continue to work in single-species models, though, since the `sim` symbol would still exist. User-defined functions can never be limited to running only in particular ticks; they can be called from any script block at any time.

Execution order

The trickiest design issue to resolve is the order in which things happen within a given tick. In SLiM 3, the overall order of execution within a generation was defined by the generation cycle of the model type (Figure 3).

Each of the top-level steps of the generation cycle is referred to as a "stage" (or sometimes a "phase" or a "step"). In stage 1 of WF models, and stage 2 of nonWF models, for example, `early()` events are executed. Within a stage, in SLiM 3's single-species design, particular events or callbacks are executed in the order in which they are defined in the user's SLiM script, from top to bottom. If multiple `early()` events are defined that would all run in the same tick, for example, then they run from top to bottom within the appropriate cycle stage. We will call this order of execution *definition order*; events execute in definition order.

In SLiM 4, if only a single species is running its cycle in a given tick in a multi-species model, the execution order would be exactly the same as it is now (preserving backward compatibility). When more than one species is active in a given tick (Figure 2), the same tick cycle would also be used, but the precise execution order requires some further clarification.

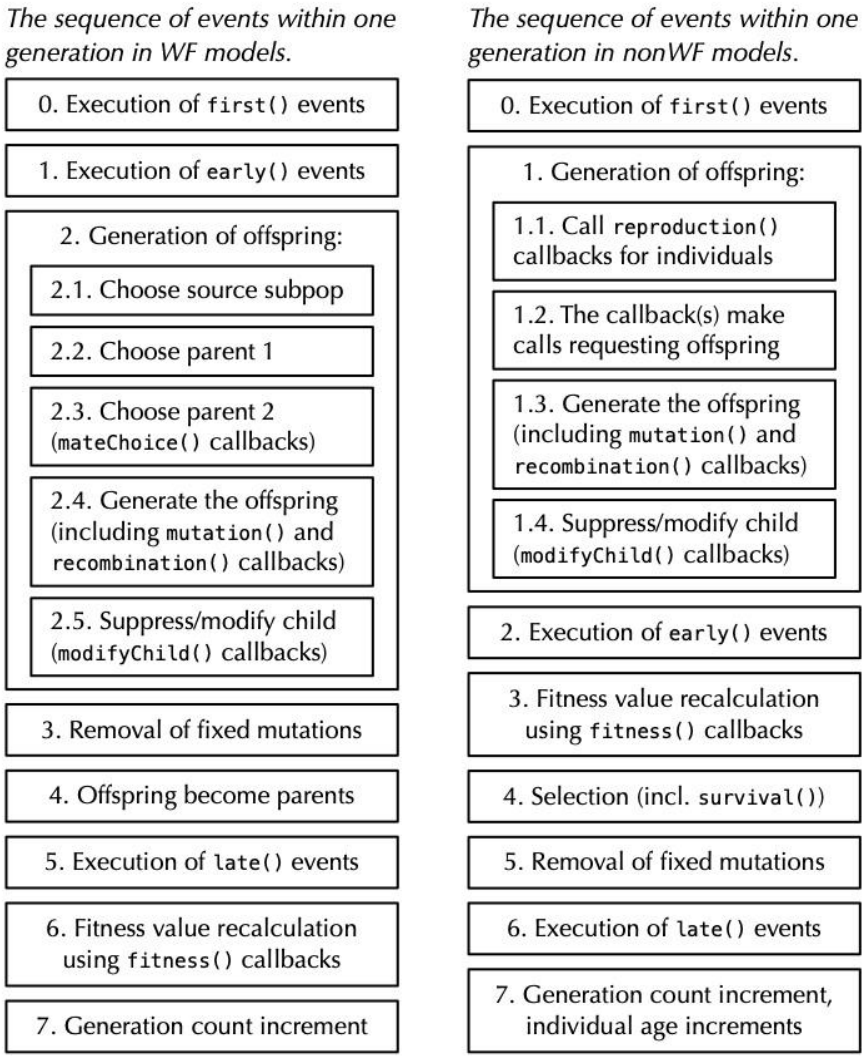

Figure 3. The SLiM 3.7 generation cycle for WF and nonWF models. This cycle would be unchanged in SLiM 4, but the execution order for multiple species within each cycle stage requires clarification.

First of all, in a given tick the cycles of species that are active are *interleaved*: all active species execute stage 1, then they all execute stage 2, etc. (If a model does not want this to be true, it can arrange for species 1 to run only in even ticks, and species 2 to run only in odd ticks, for example; so this design allows for non-interleaved execution of tick cycles if desired, by using non-intersecting tick schedules for the species.) *Within* a given stage, the order of execution depends upon the type of work being done by SLiM in that phase (Figure 4), as discussed next.

If the stage primarily involves events – `first()`, `early()`, or `late()` – then those events will execute in *definition order*, executing top-to-bottom in the order in which they are defined in the script. This allows a model to do event-type work in whatever order it wishes. For example, within one event stage, an event might move all the mice, then another event might move all the foxes (hunting for mice), then an event might look for fox/mouse intersections and have the foxes eat the mice they find, then an event might handle seasonal environmental changes, then another event might move foxes that have eaten back toward their dens while decreasing the fitness

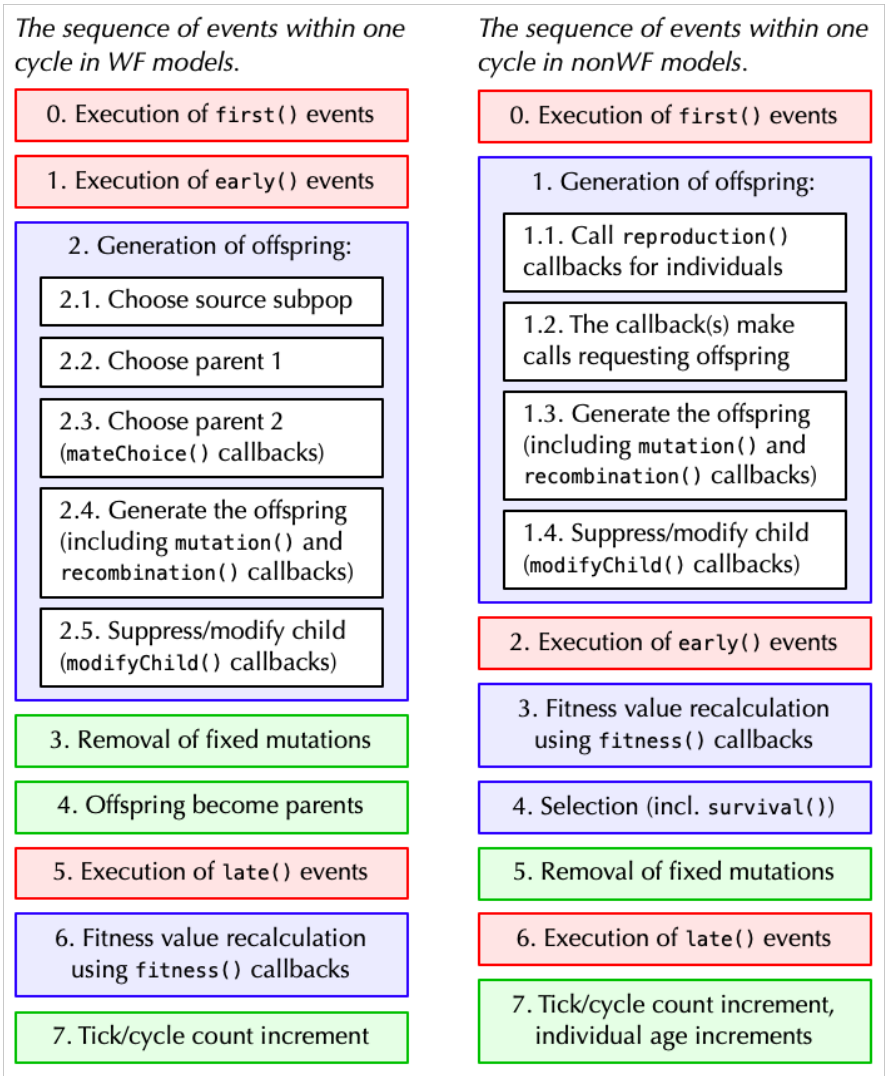

Figure 4. The new SLiM 4 multispecies tick cycle, for WF and nonWF models. Phases colored red – the `first()`, `early()`, and `late()` event stages – execute all active events in *definition order*, from top to bottom in the user's script; this allows precise ordering of events. Stages colored blue – involving reproduction or fitness calculations, and potentially involving execution of non-event callbacks – execute one species at a time, in *species declaration order*, the order in which the species themselves are declared in the user's script, from top to bottom. The execution order of stages colored green – involving no events or callbacks – is undefined at present; SLiM will execute these in whatever order is convenient, since the effects of these stages on each species are independent and their ordering thus has no user-visible effect.

of foxes that failed to eat, then an event might handle density-dependent fitness for the mice that survived, and so forth. These events would simply be defined in top-to-bottom order to produce the desired order of execution. In any given tick some of these events might not run, if they are designated with a `ticks` specifier as running only in the ticks when a given species is active (as governed by `tickModulo` and `tickPhase`); perhaps the event that moves foxes back to their dens would be designated `ticks fox`, for example, because that movement would only happen in ticks in which the foxes are reproducing. The event in which foxes eat mice, on the other hand, might be designated `ticks all`, which would mean that foxes would eat mice in every tick, even if they

were not executing their full cycle in that tick – not reproducing, recalculating fitness, undergoing mortality, etc. The designer of a model would need to think carefully about these timing choices, if different timescales were used for different species; many simple models would probably just have every species active in every tick.

If the stage involves species-specific callbacks (reproduction, fitness calculations, survival) then the code for that stage for different species will be run consecutively, in *species declaration order*: for example, foxes would do *all* of their reproduction, then mice would do *all* of their reproduction, and so forth. The order in which the species do their work, within a single such stage, is determined by the order in which they were declared via their `initialize()` callbacks, from top to bottom: if you want a particular species to go first in the execution of these types of stages, simply declare that species first. Within the execution of a stage for each species, the relevant callbacks (if more than one `reproduction()`, `fitness()`, or `survival()` callback is defined for that species, for example) will be run in top-to-bottom *definition order*, exactly as is the case now, preserving backward compatibility.

These rules set out a well-defined order of execution for each tick. I don't anticipate this commonly needing to be more flexible – e.g., sometimes wanting foxes to reproduce before mice, and other times wanting mice to reproduce before foxes, depending upon the tick – so I don't think the syntax needs to jump through hoops to allow finer-grained control of the timing of within-tick execution. In fact, models wanting *really* fine control over execution order could always adopt a more continuous-time approach, with a `first()` event in every tick that would decide on one single thing that would occur in that tick (a single individual reproducing, a single individual dying, etc.), and limiting the species in the model to just fulfilling that single task in that tick; such a "Moran model" style of design would be possible, although it would probably run quite slowly. More likely, models might go a little way toward such a design, allowing the custom execution order they need while not being quite so extreme in their approach as to only execute a single task in each tick. Anyway, I think such designs are doable within this framework, for users that really need it, but this is a marginal issue. If the timing rules set out here prove to be insufficient, some sort of additional syntax for adjusting timing could be introduced later on, but I don't expect to need to do so.

Spatial interactions

Species in a multi-species model will typically interact with each other in some way, and SLiM will need to support the implementation of such interactions with its spatial interaction engine, InteractionType. This is a complex topic, and I originally proposed a very different design than the one I have now arrived at. Implementation of the final design is now complete, and actually involves no substantive change to the list of InteractionType methods present in SLiM 3, summarized in Table 1. However, the functionality of those existing methods has been considerably extended, as will be discussed below.

For the sake of discussion in the remainder of this section, let's imagine a model of five species of Darwin's Finches on an island in the Galapagos.

Spatial query type	Individual–Individual interactions	Individual-Point interactions
Distance measurement to all individuals	`distance()`	`distanceFromPoint()`
Distance measurement to interactiong individuals	`interactionDistance()`	–
Counting all neighbors	`neighborCount()`	`neighborCountOfPoint()`
Counting interacting neighbors	`interactingNeighborCount()`	–
Getting all neighbors	`nearestNeighbors()`	`nearestNeighborsOfPoint()`
Getting interacting neighbors	`nearestInteractingNeighbors()`	–
Interaction strength measurement	`strength()`	–
Total interaction strength of interacting neighbors	`totalOfNeighborStrengths()`	–
Population density of interacting neighbors	`localPopulationDensity()`	–
Drawing proportional to interaction strength	`drawByStrength()`	–

Table 1. Spatial queries supported in SLiM 4; this list of methods is much the same as in SLiM 3.7 (`neighborCount()` and `neighborCountOfPoint()` are new, but that is unrelated to the multispecies redesign). However, the functionality of these methods has been extended to accommodate queries between subpopulations and between species (see text). Note that the `distanceFromPoint()` method was named `distanceToPoint()` in SLiM 3; it has been renamed, and now takes its arguments in the opposite order, for greater consistency.

Indirect interactions

In one type of model, the interactions between these finch species would be indirect, mediated by the environment. The model might keep a sort of spatial map that has a count of the number of seeds of different types, in grid squares across the landscape, and let the finches forage for seeds according to their individual seed-size preferences and foraging abilities. Their competition would be indirect and emergent from this feeding behavior, as seeds of particular types run out and some finches go hungry. This type of model would be easily implemented with existing tools – in particular, by defining one or more global matrices of seed counts at different grid points, and modifying those global matrices as the finches feed. A model of the effects on other species of beaver dam-building, or of afforestation, could work similarly. Such a model might use InteractionType only for within-species mate searches.

Direct interactions (aggregated count)

In another type of model, the interactions would be more direct, but aggregated – based upon local density, rather than separately handled individual-to-individual interactions. Here, an individual of one finch species might assess the total negative fitness effect exerted upon it by all other nearby finches, in an aggregate estimate of the

strength of competition it is experiencing. In the simplest form, this would involve simply counting the number of interacting individuals within an interaction radius, and assigning a total strength of competition based upon that count. In a single-species model, this would most efficiently be implemented using the existing InteractionType method `interactingNeighborCount()`, which returns the number of *exerters* that interact with a focal *receiver* individual. In SLiM 3, that method will find only exerters of the interaction within the maximum interaction distance of a focal receiver in the *same subpopulation*. That is the fundamental limitation that needed to be lifted for multispecies support. The old method signature looked like this:

```
- (integer)interactingNeighborCount(object<Individual> individuals)
```

In SLiM 4, this method now allows the exerters to be in a different subpopulation from the receivers. There are limitations upon this flexibility, for efficiency; for a given call to `interactingNeighborCount()`, all of the receivers must be in the same subpopulation, and all of the exerters must be in the same subpopulation – but those do not have to be the same. It is assumed that the receivers and exerters exist in the same coordinate system; the spatial bounds of their subpopulations need not be identical, but an (*x*, *y*) position in one subpopulation must refer to the same actual point in space in the other subpopulation. The new method signature looks like this:

```
- (integer)interactingNeighborCount(object<Individual> receivers,
        [No<Subpopulation>$ exerterSubpop = NULL])
```

The old `individuals` parameter has been renamed `receivers`, to emphasize the role that those individuals play in the interaction. More importantly, a new parameter has been added, `exerterSubpop`. This parameter is optional, and its default value, `NULL`, falls back on the old SLiM 3 functionality of counting exerters within the same subpopulation (and thus the same species) as the receivers; this preserves backward compatibility. But now a different subpopulation, potentially in a different species, can instead be passed in, and the query will then be run counting exerters in that different subpopulation that are near the spatial location of each receiver.

This is the basic approach throughout the new API design. For methods that took only a receiver or receivers, like `interactingNeighborCount()`, a new optional `exerterSubpop` parameter has been added that allows cross-subpopulation and cross-species queries while preserving backward compatibility. For methods that already took a vector of exerters – the `strength()` method, for example – those exerters are now permitted to be in a different subpopulation, and thus a different species, than the receiver.

InteractionType has always required that one call a method named `evaluate()` to "evaluate" a subpopulation before interaction queries involving that subpopulation can be done. This remains true now; one is simply required to evaluate *both* subpopulations involved in the interaction you want to query, the receiver subpopulation and the exerter subpopulation. As in SLiM 3, this takes a snapshot of the spatial positions of all of the individuals in the evaluated subpopulation, and those snapped positions are the positions used by InteractionType to respond to queries; if individuals change their spatial position after `evaluate()` is called, the InteractionType will not be affected. This is a nice feature in some ways, providing a consistent baseline state for interaction evaluation even if one wishes to move individuals around in response to the interactions they are receiving; but it is also necessary, from a software architecture standpoint, to make the underlying algorithms efficient, since it allows spatial data structures to be build once and reused many times.

Direct interactions (aggregated interaction strength)

A third type of model makes this aggregated strength of competition depend not only upon the number of nearby individuals, but upon the strength of competition each of those nearby individuals exerts, depending upon their spatial proximity and perhaps also their phenotypic similarity or other factors. This is the type of competition illustrated in recipes in the SLiM manual, such as in section 15.2 and later chapter 15 recipes. With a multi-species model such as the finch model, an individual of one species might feel the strongest competition from individuals of its own species, and different, lesser levels of competition from the other two finch species (i.e., competition strength depending upon spatial proximity and species identity, without assessing individual phenotype); or competition strength might be phenotype-based (based on beak size, probably, in our finch example), more accurately representing competitive effects when the phenotypic distributions within each species might be somewhat broad and overlapping. In existing single-species models these types of competition would typically be implemented with the vectorized `totalOfNeighborStrengths()` method of InteractionType, which tallies up interaction strengths calculated between interacting pairs of individuals, or with `localPopulationDensity()`, which similarly tallies up interactions but then divides by the available habitat in some sense (the integral of the interaction function clipped to spatial bounds, in fact) to provide a metric of the "interaction density" exerted upon receivers.

As before, in SLiM 3 these methods could evaluate interactions only within a single subpopulation, but now they allow the receivers and the exerters to live in different subpopulations, and thus different species. This approach raises an additional issue, however, since now we are totaling up interaction strengths, not just counting interacting individuals: how can the interaction strength be modified by factors such as phenotypic similarity? One might, similarly, wish interaction strengths to depend upon the genetics of coat color for mice versus the learned "search image" of a fox used to looking for mice of a particular color; or many other such factors.

In SLiM 3, such considerations *could* be coded as an additional spatial dimension (see section 15.7 of the manual), which is elegant and efficient when it is possible, but when it is not, one must write an `interaction()` callback. This callback type allows the interaction strength between a focal receiver/exerter pair to be tailored according to their genetics, their phenotypes, their learned behaviors, or anything else you wish to write in your Eidos script.

In SLiM 4, this mechanism remains essentially the same. The only difference is that now, when an `interaction()` callback is called, the receiver and the exerter might be in different subpopulations or even different species. For that reason, the `subpop` pseudo-parameter that used to be passed in to `interaction()` callbacks has been removed (breaking backward compatibility, but in a very minor way). Instead, the callback should use `receiver.subpopulation` and `exerter.subpopulation` as needed, and recognize that the two might not be the same.

Direct interactions (individual to individual)

Finally, there are models where individuals interact very directly with specific other individuals, rather than just feeling a "total interaction strength". In single-species models this often involves mate choice; you don't just want to assess the level of mating interest for a focal individual, you want to find that focal individual a mate. It could also involve, say, predation: you might not want to just assess the overall predation risk felt by a mouse, you might want to find the specific fox that eats that mouse. For this sort of task, methods such as `nearestNeighbors()`, `nearestInteractingNeighbors()`, or `drawByStrength()` are typically

used. Given a receiver, these provide, respectively, *all* individuals within the interaction radius, only the *exerters* within the interaction radius (excluding individuals of one sex, for example, if the interaction is sex-specific), or just *one* individual chosen from among the available exerters weighted by their interaction strength. These APIs are all focused on individual–individual interactions.

As before, for SLiM 4 these methods have been extended to allow the search for neighbors or exerters to be done in a different subpopulation or species than the receiver, and interaction() callbacks can now modify the strength of interaction for receiver/exerter pairs that reside in different subpopulations or species.

How was this implemented?

This sounds like a very simple change, but an extensive redesign of the internals of InteractionType was necessary to achieve it. Originally I thought that this approach would simply not be possible to implement efficiently, which is why the original plan for the InteractionType redesign was so different. As it turns out, the performance after this redesign is remarkably close to SLiM 3.7's performance – often essentially identical, occasionally slightly faster. For those who are interested, I will explain here how this redesign works under the hood; those who are not interested can skip this section.

In SLiM 3, InteractionType uses two spatial data structures to respond to spatial queries. One is called a *k*-d tree; it is designed to efficiently search for individuals that are near to a given spatial location. This is what SLiM uses to find exerters that are near the location of a given receiver, and this part of InteractionType did not need to change significantly for SLiM 4; now we just feed the spatial location of a receiver in one subpopulation into the *k*-d tree built for a different subpopulation.

The other spatial data structure in SLiM 3 is called a *sparse array*; it is designed to efficiently cache the results from interaction calculations. If you have N individuals in a subpopulation, you have $N \times N$ potential interactions, and if you wish to save the results of those interaction calculations, naively you could store them in an $N \times N$ array. That is actually what SLiM 2 did, but for large population sizes it used a tremendous amount of memory; it did not scale as it needed to. For SLiM 3, the $N \times N$ array was replaced by the sparse array. The benefit of that is that if each of the N individuals interacts only with a small set of nearby individuals – M individuals on average, let's say – then the sparse array occupies only $N \times M$ memory rather than $N \times N$ memory. For $M << N$, this is a huge win, and it worked pretty well for SLiM 3.

But limitations soon became clear. One is that even for $M << N$, the memory usage of the sparse array could be very large for sufficiently large N; if we want to get to a place where we can model populations sizes of many millions or perhaps even a few billions, $N \times M$ memory usage remains extremely painful. The other limitation is that it really doesn't scale well to multispecies models with interactions between subpopulations. To see why, picture the data structures needed for our five-species finch model. To hold the cached interactions between receivers of species 1 and exerters of species 2 (we could notate that as 1←2), we would need a sparse array of size $N \times M$, where N is the total size of species 1 and M is the average number of exerters in species 2. If we want the reciprocal interactions – receivers of species 2 and exerters of species 1 (2←1) – that's another $N \times M$ sparse array (the species identity of N and M being flipped). And we probably also need within-species interactions, 1←1 and 2←2. So already, for two species, the memory usage has quadrupled; for a single-species model, a single $N \times M$ sparse array would have sufficed. Now if we want to add a third finch species, we will need nine sparse arrays: 1←2, 2←1, 1←3, 3←1, 2←3, 3←2, 1←1, 2←2, and 3←3. For five finch species, we will need 25 sparse arrays, if I have counted them up correctly. This is a problem.

So the sparse array can no longer be relied upon to serve our needs. The first redesign I envisioned was to make all queries between subpopulations be point-based and not use the sparse array design at all. In other words, you would take the spatial location of a receiver, and use a point-based call to ask InteractionType "what exerters are near this spatial location, and what interaction strength would they exert upon a receiver there?" However, this shift towards point-based calls had major drawbacks. One is that the receiver individual drops out of the picture; this would mean that `interaction()` callbacks would no longer know the identity of the receiver, for example, making it impossible to tailor interaction strengths according to receiver–exerter genetics and so forth. Another drawback is that it means that interaction information is no longer cached at all, even temporarily within a given query; this would make the implementation of some of these point-based queries much more complicated, and probably less efficient too. A third drawback is that we still have at least the sparse arrays for each within-subpopulation interaction: 1←1, 2←2, etc. These need to be kept because we don't want to lose existing functionality and efficiency for simple models. So for a five-finch-species model we still have five $N \times M$ sparse arrays, which is still painful. And the last drawback is an explosion of new methods in InteractionType for all these new point-based queries, making the class quite confusing and difficult to both use and maintain.

So that was the wrong design, but I scratched my head for a long time trying to think of a better one. The key realization is that the sparse array, for everything that InteractionType does, is always accessed one row at a time: we're always looking at the list of exerters for one focal receiver at a time. We only need to cache that sparse interaction information for a single row at a time – and thus we only need to build it for one row, at any given time. This allows us to switch from a design revolving around sparse arrays of size $N \times M$ to a design revolving around sparse vectors of size M. When we need the row of the sparse array for a given receiver, we build it, use it, and throw it away. In memory, at any given time, we have only the single sparse vector of length M that we need to service the portion of the query we are currently working on. Conceptually, we are still using a design based upon sparse arrays; but in practice, the memory usage for those sparse arrays disappears almost completely. And because of this, it no longer matters how many sparse arrays we have, for a given model; if our five-species finch model uses 25 of them, the memory usage is still essentially zero. And it avoids needing to add a whole new wave of point-based query methods to InteractionType, and it scales easily.

The only concern was whether all of this building and throwing away of temporary sparse vectors would turn out to be much slower than building a more permanent sparse array that could be reused repeatedly. That has turned out to really not be an issue, however. The reason is that a given interaction for a given receiver is almost always queried only once before the InteractionType needs to be re-evaluated (e.g., because individuals have died or been born). So in the old design we would build the whole sparse array, use each row of it once (if even that), and then throw it away. In the new design, we build and throw away each row individually, as needed. The total amount of computation is essentially the same; or if some rows are never used (because interactions for some receivers are never queried), the new design can be even *more* efficient.

The one place where we do lose efficiency, in some cases, is in *reciprocality*. A reciprocal interaction is one which has the same strength if the receiver and exerter roles are switched; if A exerts a strength of x upon B, then B exerts a strength of x upon A. Such reciprocal interactions are commonplace; if the competition strength between two finches depends upon their spatial proximity and their relative beak sizes, for example, then the strength of competition one finch exerts upon another might be defined to be equal to the strength the other exerts upon the one. In principle, one could leverage this fact to calculate the interaction strength just once, cache it, and reuse it when the reciprocal interaction is queried. This was actually the motivation behind the original $N \times N$ array design used in SLiM 2; that made it very easy to cache and reuse reciprocal interactions. And when interaction

strengths are slow to calculate – when `interaction()` callbacks are active, in particular – that design did result in a substantial speedup for models involving reciprocal interactions. However, that benefit was mostly lost when we switched to sparse arrays in SLiM 3, because inserting the reciprocal interaction into the sparse array often took more time than it saved; the sparse array data structure is just not optimized for random-access insertion in the way that an $N \times N$ array is. So in SLiM 3 the reciprocality property was only used when `interaction()` callbacks were active, and even then it was only a win when those callbacks were fairly slow. In SLiM 4 we lose the ability to cache reciprocal interaction strengths at all, because there is literally no data structure to cache them in any more – there is only the sparse vector for the interaction we are presently evaluating. This does mean that models that use reciprocal interactions with `interaction()` callbacks will often run more slowly than in SLiM 3 – potentially up to twice as slowly, in the worst case where runtime is completely dominated by the callback overhead. Such is life.

In summary, then, by removing the sparse array from InteractionType and switching to dynamically generated sparse vectors, we can easily and efficiently address the common needs of multi-species models for spatial interactions between individuals of different species.

No-genetics species

In SLiM 3, one virtually always wants to simulate genetics of some kind; that's the whole point. But with multispecies models in SLiM 4, that may no longer be the case. One might want to use SLiM simply as a framework for "pure ecology" models; there is no reason why it can't be used for that! One might also want to write a model in which some species have genetics and can evolve, whereas other species are present in the model only for their ecological effects and do not need to evolve. We'll use the term "no-genetics species" for such species. In SLiM 3, one would need to make a set of boilerplate initialization calls to set up a no-genetics species:

```
initialize()
{
   initializeMutationRate(0.0);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 0);
   initializeRecombinationRate(0.0);
}
```

This just sets up an empty chromosome (of length 1, since there is no legal way to set a chromosome length of zero in SLiM 3), with mutation and recombination rates of zero. Since this is expected to be much more common in SLiM 4, a streamlined way to do this has been added. A no-genetics species can now be declared like this:

```
initialize()
{
}
```

Of course other initialization code might be present in the callback; the point is that all of the genetics initialization boilerplate can be omitted. This actually provides a zero-length chromosome, too, so it is functionally superior to the old no-genetics style.

Note that these two styles cannot be mixed; there is no intermediate point between "no-genetics" and "genetics". Either you leave out all the boilerplate and declare a no-genetics species, or – as in SLiM 3 – you are required to specify the mutation and recombination rates, and set up a chromosome with at least one genomic element (and therefore, at least one genomic element type and mutation type).

API modifications

This section lists the specific API modifications I have proposed above, as well as some others that have become part of the design but were not discussed above. The list includes a few SLiM 4 changes unrelated to the multispecies work, just so people know what's changing in SLiM's Eidos APIs. Now that development work is being done, I have divided this list into items that have been completed and items that have not.

Completed:

- Add a new class, Community, in both C++ and Eidos, and rename SLiMSim to Species; Species gets most of what was in SLiMSim, but Community gets logfiles, tick execution management, script block management, InteractionType management, and a few other non-species-specific things
- Add a `community` global constant to provide access to the Community shared object
- Add a `tick` property to Community to get the current tick counter
- Add an `allSpecies` property to Community to get a vector of all Species objects; also add `allMutationTypes`, `allGenomicElementTypes`, `allScriptBlocks`, `allSubpopulations`, etc., to Community to get vectors of all of those objects across all species
- Add a new optional species-initialization function, `initializeSpecies()`, with an `avatar` parameter that sets a string (typically a single emoji, like a fox or a mouse) and a `color` parameter that sets a color, both used to represent the species in SLiMgui
- Add `[tickModulo = 1]` and `[tickPhase = 1]` to `initializeSpecies()` for control of which ticks the species is "active" in (i.e., executes the tick cycle in)
- Add `tickModulo` and `tickPhase` properties to Species to access those values
- Add a `cycle` property to Species to get the current cycle counter for that species; remove the `generation` property
- Add a `skipTick()` method to Species to skip running in a given tick for a given species; this sets `active = F` for the species, sets `active = 0` for all callbacks designated for that species with `species <name>`, sets `active = 0` for all events designated to run only when that species is active with `ticks <name>`, and prevents the cycle counter for that species from incrementing at the end of the tick, just as if the species had never been scheduled to run in the current tick at all
- Add a read-only `string$ name` property to Species to get the species name (`"fox"`, `"mouse"`, etc., or `"sim"` by default in a single-species model), and a `string$ description` property for a longer description; these get persisted in tree-sequence top-level metadata
- Add the `species <name>` syntax for designation of species-specific callbacks, including `initialize()` callbacks (which define the existence of particular species names); this includes the `species all` syntax for non-species-specific `initialize()` callbacks and `interaction()` callbacks

- Require that `initializeSLiMModelType()` and `initializeInteractionType()` be called from a `species all initialize()` callback (in multispecies models), since this simulation configuration is non-species-specific
- Add the `ticks <name>` syntax for designation of events that run only when a given species is active; this includes the `ticks all` syntax for events that should run in every tick
- Add Community methods `subpopulationsWithIDs()`, `mutationTypesWithIDs()`, `speciesWithIDs()`, etc.; these provide a way to quickly look up objects by their `id` property, to make the cross-references that might exist in a multispecies model faster/easier to handle
- Add a new property, `species`, to many classes (SLiMEidosBlock, MutationType, Subpopulation, etc.) to get the species that they belong to, for whatever purpose the user might have
- Change the times for tree-sequence recording from generations to ticks; change the `originGeneration` property of Mutation and Substitution to `originTick`; change the `fixationGeneration` property of Substitution to `fixationTick`; change the tree-sequence metadata schemas as necessary to adjust to these changes; and so forth
- Change the `generationStage` property (now on Community) to `cycleStage`
- Add a new `addTick()` method to LogFile; change the `addGeneration()` and `addGenerationStage()` methods of LogFile to `addCycle()` and `addCycleStage()` respectively; note that the names for these columns in the logged output also changes, from `generation` and `gen_stage`, to `cycle` and `cycle_stage`
- The `simulationFinished()` method on Species should be legal to call only in single-species models (to preserve backward compatibility); Community adds a new `simulationFinished()` method that ends the whole simulation
- Remove the default of `early()` for events; `early()` must now be written explicitly, there is no default (this removes some excessive complication from the event declaration syntax now that `first()` events and `ticks` specifiers have been added)
- Switch InteractionType from using a "sparse array" to using "sparse vectors", under the hood, enabling interactions to be evaluated across subpopulations and even across species
- Modify InteractionType interfaces to emphasize the (now much more important) distinction between "receivers" and "exerters", to allow an "exerter subpop" to be specified for cross-subpop and cross-species interactions, and so forth
- Change the `evaluate()` method of InteractionType to require receiver and exerter subpopulations to be specified explicitly with `subpops`, since the mechanics of evaluation have become more complex and should be made explicit for clarity; remove the `immediate` parameter since immediate evaluation of interactions is no longer supported
- Relax initialization requirements to allow "no-genetics" species to skip the initialization boilerplate for genetics, providing an empty chromosome with mutation and recombination rates of zero
- SLiMgui improvements: show both the tick and cycle; display multiple species with avatars; add a "species tab bar" to select the displayed species; implement auto-fix capabilities to ease the transition to SLiM 4 where possible; make graph windows species-specific, bound to the active species when they are created; add a "multispecies population size ~ time" graph to see multispecies population dynamics in a single plot; etc.

- Tweak the output formats, in the header line, for `outputFull()`, `outputMutations()`, and `outputFixedMutations()` to include both the tick and generation
- Remove some obsolete/deprecated APIs: the `inSLiMgui` property of SLiMSim; the `originGeneration` parameter to `addNewMutation()` and `addNewDrawnMutation()`; obsolete pseudo-parameters for callbacks such as `genome1` and `genome2`, `childGenome1` and `childGenome2`, and so forth (this information can all be accessed through the corresponding Individual object)