How Simile works

Here is a simple description of how Simile executes a model after the diagram has been created and fully specified.

Running the model

When a modeller gives the instruction to 'run' a model, Simile converts the model diagram and equations into a c++ program which is compiled into a shared library. This library is immediately loaded into Simile itself. Simile then interacts with the library to run the model and obtain results which can then be displayed or recorded.

The important components of the program implementing a model are as follows:

- A class for each submodel in the model's hierarchy, including the top level (desktop). Each class contains a variable for each component in its submodel, and a 'get_pointer' procedure for decoding numerical references to its variables
- An 'update' procedure, which adjusts the values of the model's state variables at each time step
- An 'evaluate' procedure, which carries out all the other calculations for a given time step, including initializing state variables when the model is reset
- An array of data structures containing information about each component in the model

Simile's model interaction functions are provided by another shared library, which can be linked against by other application programs. This enables them to load shared libraries corresponding to Simile models, and to execute those models via their own user interface. The generated model code calls utility procedures in this library, as well as others in code included by the model program.

Serializing the network

The first part of the program-building process consists of creating a program variable for each primitive component in the model, and a program instruction to generate the value of each component from the values of the other components on which it depends. The variables are grouped into classes corresponding to submodels. In the case of a compartment, the value is determined by its previous value and those of the flows going in and out of it; they do not have to be updated in any particular order. For other components, the value may be derived from the values of other components calculated earlier in the same operation, so ordering is important.

The instructions that calculate these values are sorted by searching among them for those which use only compartments and other values whose instructions have already been ordered, and building the results into an ordered series of instructions depending on the results of previous instructions. These are then made into a procedure that calculates the values of all the model components, starting from those of the compartments and ensuring that no value is calculated before all those on which it depends have

been calculated.

A check is done for sets of influences forming a loop before serialization is attempted, since serialization will not be possible if there is such a circular set of influences. The ordering effect of an influence is disabled if its 'use values from this time step' property is selected, allowing models to include graphically represented cumulative and iterative algorithms.

Multiple instance submodels

During the serialization process, Simile maintains a submodel 'context'. This starts at the top level of the model (the desktop level). Instructions for each component can only be put into order in the context of the submodel containing that component. When no more instructions can be ordered at the current level (e.g., because every remaining variable in the submodel depends on a variable outside it) Simile tries to switch context to a submodel within that level. Instructions within that submodel can then be ordered. If there is nothing that can be done in any child submodel context it switches to the parent context.

The values of model components in a submodel are represented as member variables of a class which corresponds to the submodel. An instance of this class is created for each submodel instance. Sections of the code that are created in the context of a multiple-instance submodel are placed within a program loop which executes once for each instance of the submodel. The value assignments use a pointer which points to the current instance of the submodel class and is incremented each time the loop is executed. The importance of submodel context in serialization is to minimize the number of program loops that need to be created in the evaluation procedure. Array variables can be evaluated or accessed in the same context as submodels with the same dimensions; in this case the loop count is used as the array index.

Functions that convert an array into a scalar value cause an 'intermediate variable' to be created. Two instructions are added to implement the function; one, outside the context of the loop in which the array is created, to initialize the intermediate variable, and another -- inside the loop context but after the initialization and before the instruction that evaluates the equation containing the function -- to increment or otherwise alter the intermediate variable on each loop execution.

Variable-membership submodels

A variable-membership submodel is represented by a linked list of instances of the class corresponding to the submodel. For a conditional submodel, the list is set up at the start of the first program loop in which variables inside the submodel are evaluated. The list is initially empty, and each time the membership needs to be checked, the program enters a loop over all potential instances of the model (as specified by its dimensions). If an instance does already exist, it is snipped out of the list, otherwise a new one is created. The instance is then evaluated as far as the existence condition, at which point it is either deleted or inserted into the list. On subsequent visits to that submodel context, the program merely loops through all the members of the list.

In Simile, a value coming out of a variable-membership submodel is known as list value. Functions can convert this to a scalar value as described above for array values, but it cannot be assigned as a list to a model variable.

Population submodels

A population submodel is another kind of variable-membership submodel. It is also represented by a linked list, but has separate groups of instructions within the evaluate procedure to manage its membership. Before first entering the population's context, a loop is entered which deletes all instances which have a 'loss' component whose condition is satisfied. Then, new instances are appended for any individuals that have been added (including the initial membership if the model is being reset). This involves a loop over existing members if there is a reproduction channel in the population.

A consequence of this is that an individual will still exist at the end of the time step in which its 'loss' condition is satisfied, so every individual exists for at least one time step. This allows other model actions to be triggered by the loss of an individual.

Association submodels

An association submodel is a special case of a conditional submodel. As well as its own dimensions, the procedure that determines its membership is also enclosed in loops that iterate over the memberships of its base model(s). When evaluating its existence condition, values in the base models are referenced by pointers that are updated in these loops. If an association submodel instance does exist, these pointers are stored within its data structure in the submodel's linked list.

On subsequent visits to the association submodel context, the program just loops through its instance list. References to the base model instances are handled by the pointers that were stored with it during the membership test. If a base model has an 'exclusive role' in an association, then variables in that base model which depend on values from the association model can be evaluated within the association model's context, since, if the membership meets the definition of an exclusive role, that will result in them being evaluated exactly once per time step in each base model instance.

If a role specifies that the base instance is looked up, and the association model's condition is of the appropriate form, then there is no loop over this base model's members – instead the index is calculated from the association model's condition expression when evaluating its membership, and a pointer to the base model instance is generated from that index.

Time steps

Not every value in a Simile model needs to be evaluated at every time step. So when the evaluation procedure is called, it has an argument to indicate what action is being taken. Table 1 shows what actions are defined, and what the evaluation procedure does for each of them. Each action includes all those listed below it.

Action taken	*Implementation*
Initialize	Set literal constants
Reset after loading fixed parameters	Set fixed parameters from file
Normal reset	Set compartments to initial values Set random constants
Longest time step	Set variable parameters and compartments in submodels using that time step
Shorter time step	Set variable parameters and compartments in submodels using that time step

Any other model variable is set in the first action that sets every value on which it depends. For instance, if a variable is calculated exclusively from fixed parameters, then it will be set when the model is initialized or reset after loading fixed parameters. It will not be set on an ordinary reset or during subsequent execution. In addition, no variable will be set on a shorter time step than that specified for its own submodel.

Memberships of conditional submodels can also be set in the first action that determines all the values on which they depend. When a new submodel instance is created it gets a flag saying it is new, and this has the effect that all values in that instance are set whatever action is being taken. The flag is cleared at the start of the next evaluation.

Integration methods

As well as the 'evaluation' procedure described above, an 'update' procedure is generated for the model. This is what adjusts the values of compartments according to the values calculated for their flows by the evaluation procedure. The update procedure takes a similar form to the evaluation procedure, with program loops around the groups of statements that apply to compartments in multi-instance submodels, but there is no ordering requirement, as the increments for each compartment do not depend directly on other compartments. This means that there is only ever one program loop for each submodel.

For each compartment, the 'update' procedure calls a utility procedure in the included code (the 'increment' procedure), within loops over arrays or lists of submodel instances. This procedure is passed the net sum of flows in and out of the compartment, the time step identity assigned to the submodel, and a pointer to a data structure associated with the compartment in its submodel class. The value returned by the increment procedure is added to the compartment's previous value within the update procedure.

When Euler integration is selected, the return value is simply the net sum of flows multiplied by the time step duration (calculated as the elapsed model time since compartments in submodels with that

time step identity were last updated). The compartment's associated data structure is not used.

When executing with Euler integration, Simile runs the update procedure then the evaluation procedure, then increments the model time by the shortest specified time step. This cycle is repeated until the specified execution period is completed. If the increment in model time takes it past the boundary of a longer time step than the shortest, the 'action' passed to the evaluation and update procedures is set to the identity of this time step to indicate that values that only change on this longer period should also be updated and evaluated.

Simile also allows model execution with 4^{th}-order Runge-Kutta integration. The model code is the same, but now the update and evaluation procedures are called alternately four times each model time step. The model time is incremented by half the shortest time step before the first and third calls to the evaluation procedure. Each time the update procedure is called, a global variable is set to indicate that Runge-Kutta integration is being done, and which update in the sequence is being made. This variable affects the behaviour of the increment procedure. Depending on its value, the increment procedure returns the value needed to set the compartment to what it should be for the next stage of the Runge-Kutta algorithm, or in the case of the last update of the sequence, to its final value as determined by this algorithm. The compartment's associated data structure is used to store intermediate results between different updates in the sequence. The behaviour of the evaluation procedure is the same each time it is called in the sequence, but it is starting with different, partially incremented compartment values each time.

The detail is best expressed by the code itself (procedure stage_incr in files support2.cpp and support.tcl, and procedure rk_update in shank.cpp and support.tcl, in Simile's Run directory, shank.cpp only distributed in Unix versions), and a description of the Runge-Kutta integration algorithm, such as that in “Numerical Recipes in Pascal” by Press, Flannery, Teukolsky and Vetterling (Cambridge University Press, 1989).