Appendix A

Data Structures

A.1 Overview

The simulator is composed of many objects, which have very strong dependencies and are one of the key components in allowing us to use efficient algorithms for recomputing the set of adjacent moves. Our data structures contain a lot more information than those in previous works, but this is what allows us to use the more efficient algorithms for move selection, update and energy computation.

The following is a list of the primary objects which we will elaborate on in this section:

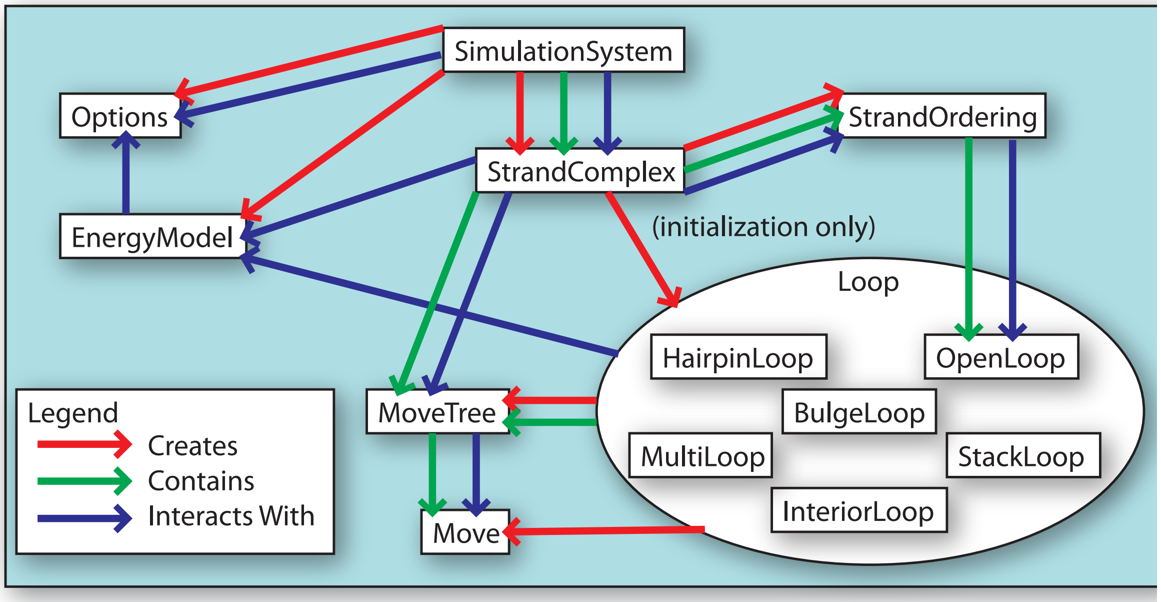

Figure A.1: Relationships between data structure components

Object Name	**Basic Function**
SimulationSystem	Controls the simulation, handles trajectory parameters and simulation setup.
StrandComplex	Summarizes the information for a particular complex, including the ordering, intra complex moves, and loop structure.
SComplexList	Stores all the complexes in the system and handles inter-complex moves.
Loop	Base class for all types of local loop objects. Contains information about the energy, adjacent loops, and has all the generic loop accessors which are implemented by derived classes.
Move	Contains information about a single move, including the affected loops, type of move and rate at which it should occur.
MoveTree	Container type for organizing the Move and MoveTree nodes for a particular unit: a loop, a complex or the system.
EnergyModel	Base class for all types of energy models. Defines all the accessors for finding the energy of particular loop structures. Derived classes implement these, and contain all the necessary parameter data.
StrandOrdering	Auxiliary class used by a Complex to handle strand ordering information and functions for checking stop states, handling output of the complex and other related tasks.
Options	Auxiliary class which contains all the simulation options. This is the object which contains all the model and simulation parameters necessary for the simulator, as well as the input DNA system. It is also used to handle the results coming from the simulator (e.g. the output). This is the only Python object included in this section.

A.2 SimulationSystem

Purpose: This object handles the main control loop of the program in addition to performing the setup and initialization duties.

Functionality:

1. Initializes the EnergyModel object based on parameters from the Options object.
2. Initializes the system state for each trajectory.
3. Handles random number generation for each step of the markov process.
4. Performs the main loop of the simulator, performing a move, updating the time and handling output duties.

Data Members:

1. EnergyModel: contains a reference to the energy model being used for the system. This data is also used to initialize a static member of the Loop class, so that all loops have a direct reference to the correct energy model, but we need to store it here so that it can be properly freed later.

2. SComplexList: the current secondary structure state of the system is stored in this object, which needs to be reinitialized to the starting state for each trajectory.

Discussion: The system object, while containing the main control loop for the program, is very limited in the scope of what it does, handling only the main initialization duties and the random number generation component of each step of the markov process. This lets us focus on handling all of the different simulation modes (and thus output options) at this point without having to know about any of the secondary structure information in the system.

This is one of two C++ objects that are closely connected to the Python side of the simulator, as we need to both access the parameters and inputs for a particular simulation system as well as provide output back across that interface.

A.3 StrandComplex

Purpose: Represents a single connected complex of strands. Things that take place at that level are handled here if they don't involve the specific ordering of the strands. This includes the initial generation of the loop objects representing the complex as well as handling joining moves between two complexes.

Functionality:

1. Contains code that generates the entire Loop-based structure from the sequence and dot-paren structure input. Only used when the trajectory is initialized.

2. Performing base pair creation moves between two complexes is handled here, joining the appropriate data together.

3. Performing base pair deletion moves which cause a complex to break into two connected structures is done here.

Data Members:

1. StrandOrdering: contains information about the strands/sequences used in the complex, the cyclic permutation corresponding to the current structure and links to the actual structure.
2. MoveTree: tree structure collecting all the moves corresponding to loops within the complex.

Discussion: Originally this object also contained all the information about the current sequence and dot paren representation of the structure, but it made more sense to split those into a seperate object when it became apparent that all the functions which dealt with those used a very similar linked list approach. Because of this, there are really only three main functions here which don't involve the ordering in some way: joining a complex, breaking a complex, and initializing a complex.

The initialization of loop structure for a particular complex is easily the most complicated algorithm here, and is discussed in Section B.2. Why not handle the loop creation in another location? The simple answer is that at some point, there needs to be a function which can translate from the sequence and structure for a complex into the loop structure, and since those are only relevent at the complex level, it can't be any lower. Loop structure only knows very local bits of the sequence and has no concept of the dot-paren structure, and while the strand ordering does involve both the sequence and structure, the main functions contained there deal with modifications to that ordering and subsequent changes to the output representation, and not really anything to do with the actual structures underneath.

A.4 SComplexList

Purpose: Contains all the complexes in the current state of the system. Handles computation of join move rates between complexes, and controls move choice selection between join moves and moves internal to each complex.

Functionality:

1. Computes the total join rate between complexes, which requires information about the external bases (unpaired bases appearing in an open loop) present in each complex.

2. Handles control flow for picking which complex the next move takes place within, or computing which join move took place.

3. Handles high level code for checking a stop state, specifically when a stop state involves multiple complexes being present, we need to check for the simultaneous presence of all of them, without overlapping.

Data Members:

1. StrandComplex: linked list of complexes, along with the current energy and total rate for each complex, so that they do not have to be recomputed when a move doesn't involve that complex.

2. Join Rate: total flux for all join moves between complexes. If a move is performed within a complex, this total only needs to be recalculated when that move involves an OpenLoop.

Discussion: Exists mainly as a way of seperating the simulation system from the actual strand complexes. This layer handles summarizing the rates from each complex, as well as everything about the join moves. The join moves are calculated very simply by figuring out the total number of combinatorial choices for pairing bases. While this is done under the assumption that a single base pair is sufficient to join two complexes, it could easily be extended to require two adjacent bases to pair, by changing the definitions of exterior bases and the resulting combinatorial choice.

Note our bimolecular rate method's effects are focussed mainly here: the join move rate only involves the numble of possible pairs plus the energy model's base join move rate. If we changed the rate method this may have to be handled at a much lower level so that we can include energy computations on the resulting open loops. Currently we don't have to do that, as the energy contributions due to those loops are in the reverse rate (when breaking a complex apart), as at that point we will have detailed information about the secondary structure. If we tried to use those energies in computing the join move rates, we would have to have *much* more information about the secondary structure of each complex at this level, which would be a lot more computationally intensive. However, for some types of energies, it's conceivable that a similar redefinition of the exterior bases would work, like what we mentioned above for forming two bases at a time.

The other key thing handled here is checking for the existence of a stop state. A stop state can involve the secondary structure of multiple complexes, so we have to start handling it at this level, and it's important that we try to optimize the computation as much as possible, as the simulator goes through a huge number of states, and only a small portion of those are going to be stop states. At this level several shortcuts are implemented, such as only checking for a stop state when the number of complexes currently present would allow it, and halting the checking for a particular stop state when there aren't enough unchecked complexes in the current state to match the remaining ones in the stop state.

A.5 Loop

Purpose: Base class for one of the two (ad hoc) polymorphic types used in the simulator. Secondary structure can be broken down into a tree consisting of connected loops. The particular type of loop does not matter in this structure, but does matter for calculating the energy and possible moves available to that loop. Thus this class defines all the methods which can be used to interact with generic loops, and the specific implementations handle the details.

This class, along with the six derived classes (StackLoop, HairpinLoop, BulgeLoop, InteriorLoop, MultiLoop and OpenLoop) composes the largest percentage of the code in the program, currently around 40%. This is due to the need for each derived class needs to handle move generation and energy calculation in a specific way, as well as delete moves which need to be able to handle each of the possible combinations of two derived types.

Base Class Functionality:

1. Compute delete move rates for arbitrary pairs of derived loop types.
2. Perform delete moves for arbitrary pairs of derived loop types.
3. Perform deletion moves which cause a complex to split (OpenLoop-OpenLoop delete moves).

Derived Class Functionality:

1. Energy Computation: passes appropriate data members to the energy model for calculation.
2. Move Generation: computes rates for all internal base pair creation moves, adds generated moves to move tree data structure for that loop.
3. Move Evaluation: builds the resulting loop structure after performing a base pair creation move.
4. (OpenLoop only) Exterior bases: computes the counts of each base type accessible in that open loop for a complex join.
5. (OpenLoop only) Complex joins: create the resulting pair of connected open loops from the pair of open loops involved in a complex join move.

Data Members (general for most derived classes):

1. Loop: list of adjacent loops, in a fixed 5' to 3' ordering such that base pairs can be associated with the correct adjacent loop.
2. sequence: base sequence for each strand participating in the loop. Usually just two, but can be any number for Multi and Open loop types.
3. lengths: lengths of each base sequence present in the loop.
4. energy: stored value from calling the energy model functions, to avoid recomputation.
5. MoveContainer: container for all creation and deletion moves involving this loop. Used to make deleting this set of moves from a larger structure a simple task.

Discussion: Algorithms: The per-step move generation algorithm (Section B.4) and the move update algorithm (Section B.5) are implemented in a distributed manner in the derived Loop objects. The advantage for these algorithms are in the local properties. We only need to know for creation moves the information about the particular loop involved, and for deletion moves the two loops adjacent to the base pair being deleted. This means that creation moves can be handled within each loop type cleanly, using only the internal data to that loop to generate the two new loops which are to replace it. Deletion moves are handled in a similar way by creating a friend function of the base Loop class which handles each case of a pair of loop types.

Output Reconstruction: Though each loop object only stores a very small portion of the total sequence, it is straightforward (and indeed, must be possible) to use only those pieces to reconstruct the actual strand sequences and dot paren structures. In practice, this reconstruction, especially that of the dot paren structure for use in output, would need to be done each time we wanted to output a state, which for full trajectory output would be every step. Thus there are functions to perform this reconstruction within the class, but instead of using those at each step, the information about which base(s) were affected in a move are passed back by the relevant move update function so that the StrandOrdering object can keep the output structure up to date for use both in trajectory mode and for stop state checking.

Deletion Move Duplication: Note that a particular deletion move occurs twice, once in each loop adjacent to the base pair involved. The resulting rates are then halved, such that the total rate is correct, and the deletion move is performed identically no matter which loop's copy of the move is chosen. This means that each loop needs to be able to reset its deletion moves independently of the creation moves within that loop, and handling this cleanly is one difficulty with how we build the overall set of moves for the complex. The alternative, however, would be to store a deletion move relative to only one of the two loops adjacent, which would make the recomputation of deletion moves caused by performing creation moves to be excessively complicated. Since the total number of deletion moves per loop is small (usually two), calculating and storing the deletion moves twice is a small effect on the complexity of move generation, and is much easier to implement and more efficient for move update.

Complex Join/Break: The complex join and break moves are handled at the lowest level within this object. Complex breaks are implemented in the base class, as they need to rebuild the two open loops adjacent to the breaking base pair into disconnected open loops. Joins are handled within the OpenLoop object, forming the resulting connected open loops. While it would be convenient to handle these all within a particular object, they affect the data structure at many different levels and so there is no convenient place to handle all the appropriate changes to the strand ordering, the underlying open loops, the new resulting complex(es) and the moves available to each.

A.6 Move

Purpose: Represents a single creation or deletion move. Contains the necessary information for the appropriate Loop to perform the move.

Functionality:

1. Directs control to the correct loop to handle the actual creation/deletion necessary to perform the move.

Data Members:

1. affected loops: links to the loops affected by this move (i.e. the ones that created it).
2. move rate: rate at which the move should occur, for use by the move choice algorithm.
3. type information: used by the affected loops to perform this move

Discussion: This is the simplest data structure that's important enough to actually mention. It's really just used as data storage by the loop's move generation algorithms, so that the move choice algorithms can traverse the moves directly without having to go through the local loop structure, and once the move is chosen transfer control into the affected loop to perform the choice.

This is the basic unit that's being allocated and deallocated in large quantities at each step, and so one possibility for improving speed is to keep a pool of pre-allocated Move objects that are passed out rather than constructed from memory, and returned to the pool instead of deallocated. The allocation/deallocation hasn't appeared to be a major factor at this point though, so it's likely that other methods are a better choice for improving the simulation speed.

A.7 MoveTree

Purpose: Holds all the Node objects for a particular logical unit, such as a Loop, StrandComplex, or SimulationSystem. Implements the addition of nodes and deletion of nodes in an efficient way. Can itself be used as a base type to be organized into higher level move trees.

Note that this object is no longer fully implemented and has been deprecated in favor of moving to C++ Standard Template Library containers for handling moves. Specifically, we currently use an array data structure as our move container for each logical unit, with a move selection algorithm that is linear in the number of moves; as discussed in section B.7, move selection is not typically the rate limiting step in simulations, so the non optimal data structure does not result in a significant performance drop.

We present the following information as a look into the original construction of these data structures and algorithms.

Functionality:

1. Contains a tree structure of node objects (implemented as either Moves or MoveTrees).
2. Can singly add a node efficiently.
3. Can be constructed with serial addition of Move objects efficiently.
4. Can delete a single node efficiently while leaving the tree well balanced.
5. Performs the move choice algorithm, taking the random number and efficiently selects a node. This node then is either the Move we wish to perform, or a nested MoveTree from which we must continue the algorithm.

Data Members:

1. Tree root: pointer to the root node in the tree.
2. Tree total rate: total rate of all nodes in the contained tree.

Discussion: While we could implement a container for the Move type objects in many other ways, it is convenient to use a tree for efficiency reasons. While the worst case complexity of our simulator doesn't change, the average case can be affected be the choice of data storage for these objects, as we will discuss further in section B.7.

The ideal structure for us to use would actually be a Huffman encoding tree based on the relative rates of each move/container stored in the tree. However, maintaining this encoding for a rapidly updating selection of moves is just as inefficient as using a list structure, so while it would be ideal for our choice operations, we would have just moved the time complexity into the update steps.

Finally, we have many different choices of methods to implement our trees in a balanced fashion. Since we do not need a search operation as the elements are not sorted, and are instead accessed by the move choice algorithm using their relative rate, we can implement any simple balanced tree, as long as the insert and delete operations are logarithmic with the total size of the tree. We do need a special algorithm for constructing a tree from a loop's set of moves, as performing these insertions on an empty tree would add an extra logarithmic factor to the requirement. This is easy to implement by maintaining a static list as the tree gets built, and make a single pass to connect the structure and sum the totals once we have all the moves required.

A.8 EnergyModel

Purpose: Computes the energy of particular loop types. Computes rates for moves, based on energy model parameters and rate options. This is a base class, used to implement specific instances of energy models, such as the NUPACK energy model.

Functionality:

1. Calculates the energy for each of the six specific types of loop, based on the energy model parameters used and other energy model options.
2. Calculates the move rate based on the starting and ending state's energy, type of move and other energy model options.
3. Reads in the energy model's parameter set from a file.
4. Transforms the energy model parameter set for non-standard temperatures.

Data Members:

1. Energy model parameters: stores all the parameters for the energy model. This includes dG and dH parameters so that we can calculate dG's for different temperatures.

Discussion: Data Size: Though we list the energy model parameters as a single data member, it should be noted that this is the largest object in terms of data size. There are a huge number of parameters involved in calculating the loop energies, and extra data needs to be stored in order to calculate the energy at different temperatures. Collecting these parameters in a single place is the main reason why the energy is not calculated directly within each loop type, as the storage and extra functionality are better placed in a single external object which can be quickly called.

Scope: Note that this object is used to calculate the energy of specific loops in a secondary structure. Previous libraries for computing the energy only deal with (at the input level) the energy of an entire structure, and as such were not suitable for use here, as the computation of energy for a single loop is the basic operation which we use a large number of times for every step of the simulator.

Different Energy Models: The EnergyModel object is actually a base class, providing a standard interface for calculating the energy of a specific loop type. We implement two different derived classes, ViennaEnergyModel and NupackEnergyModel, based on the Vienna and Nupack parameter sets, respectively. These are particular implementations of the general nearest neighbor energy model and use subtly different parameter files. Other variations on the standard energy model can be used here as necessary, as long as they have the same basic function of being able to calculate the energy of a loop component.

A.9 StrandOrdering

Purpose: Handles the functions within a complex that involve the ordering of the strands within the complex. These include most output functions, reordering of strands that happens when a complex is joined to another, and returning the exterior bases in all the OpenLoops in the structure.

Functionality:

1. Maintains a linked list of the strands within the complex and the implied cyclic permutation needed.

2. Performs circular permutations on the strand ordering, necessary when joining two complexes.

3. Maintains the current sequence and dot-paren structures on demand for output use.

4. Performs stop structure checks to see if the strands in the ordering could match the given set of strand ids and their ordering.

5. Assists SComplex in building the initial loop structure, and maintains links to all the OpenLoops contained within the complex.

Data Members:

1. Strand List: doubly linked list of strands, each containing the following:
    (a) Strand ID.
    (b) Strand Sequence (output version - ASCII).
    (c) Strand Sequence (binary encoded version, used internally).
    (d) Current Dot-paren structure.
    (e) OpenLoop pointer associated with the strand.

2. Exterior base count (updated when open loops change).

Discussion: When we join two complexes together, we need to ensure that the permutation of the strands used for output corresponds to the cyclic permutation which does not contain any crossed chords (this is guaranteed by the Representation Theorem, Section D.1). Maintaining the ordering such that this is guaranteed is very straightforward, we just need to take the cyclic permutation of each set of strands such that the strands being joined together are on an 'edge'. This type of operation happens often when dealing with the structure output and multi-complex moves (joining or breaking).

This object also maintains links into the actual loop structure, used for retreiving the exterior base counts. These get updated automatically when moves are used that affect

OpenLoops. This is also used when we need to actually perform a join move, to help choose which pair of bases is actually being joined, since we only compute the total rate of all join moves combinatorially and don't actually enumerate all the options until one is chosen.

A.10 Options

Purpose: Collects all the information about input, including simulation options, energy model options, associated files and output information.

NOTE: this is a Python object and included here due to the relative dependence on the data contained within at the highest simulation levels (SimulationSystem and EnergyModel). Full documentation on the input and output routines is in the Multistrand documentation, generated automatically for the Python side of the simulator.

Functionality:

1. Maintains simulation options and variables for use by SimulationSystem.

2. Returns energy model options for use by the EnergyModel.

3. Provides interface methods for reporting output to the user.

Discussion: Collects all the options used in the simulator in one data structure. Accessed by EnergyModel and SimulationSystem to retreive the appropriate options when neeeded. Contains the default settings for all options which are not required to be set in the input.